Artificial Intelligence
Prof. Dr. Jürgen Dix

Computational Intelligence Group
TU Clausthal
Clausthal, SS 2011
**Time and place:** Tuesday and Wednesday 10–12 in Multimediahörsaal (Tannenhöhe)

**Exercises:** See schedule (7 exercises in total).

**Website**

http://www.in.tu-clausthal.de/index.php?id=cig_ki11

Visit regularly!

There you will find important information about the lecture, documents, exercises et cetera.

**Organization:** Exercise: Dr. N. Bulling, F. Schlesinger, J. Dehnert

**Exam:** Oct. 6, 2011, 10am.
History

This course evolved over the last 15 years. It was first held in Koblenz (WS 95/96, WS 96/97), then in Vienna (Austria, WS 98, WS00), Bahia Blanca (Argentina, SS 98, SS 01) and Clausthal (SS 04–11).

Chapters 1–4, 9 are based on the seminal book of Russel/Norvig: Artificial Intelligence.

Many thanks to Nils, Tristan, and Wojtek for the time they invested in crafting slides and doing the exercises. Their help over the years improved the course a lot.
Lecture Overview

1. Introduction (2 lectures)
2. Searching (3 lectures)
3. Supervised Learning (3 lectures)
4. Neural Nets (1 lecture)
5. Knowledge Engineering: SL (3 lectures)
6. Hoare Calculus (2 lectures)
7. Knowledge Engineering: FOL (2 lectures)
8. Knowledge Engineering: Provers (2 lectures)
9. Planning (1 lecture: overview)
1. Introduction

Introduction
- What Is AI?
- From Plato To Zuse
- History of AI
- Intelligent Agents
Content of this chapter (1):

**Defining AI:** There are several definitions of **AI**. They lead to several scientific areas ranging from **Cognitive Science** to **Rational Agents**.

**History:** We discuss some important philosophical ideas in the last 3 millennia and touch several events that play a role in later chapters (**syllogisms of Aristotle**, **Ockhams razor**, **Ars Magna**).
### Content of this chapter (2):

**AI since 1958:** AI came into being in 1956-1958 with John McCarthy. We give a rough overview of its successes and failures.

**Rational Agent:** The modern approach to AI is based on the notion of a Rational Agent that is situated in an environment. We discuss the PEAS description and give some formal definitions of agency, introducing the notions of run, standard- vs. state- based agent.
1.1 What Is AI?
“The exciting new effort to make computers think . . . *machines with minds*, in the full and literal sense” (Haugeland, 1985)

“[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . .” (Bellman, 1978)

“The art of creating machines that perform functions that require intelligence when performed by people” (Kurzweil, 1990)

“The study of how to make computers do things at which, at the moment, people are better” (Rich and Knight, 1991)

“‘The study of mental faculties through the use of computational models’” (Charniak and McDermott, 1985)

“The study of the computations that make it possible to perceive, reason, and act” (Winston, 1992)

“A field of study that seeks to explain and emulate intelligent behavior in terms of computational processes” (Schalkoff, 1990)

“The branch of computer science that is concerned with the automation of intelligent behavior” (Luger and Stubblefield, 1993)

### Table 1.1: Several Definitions of AI
1. Cognitive science

2. ”Socrates is a man. All men are mortal. Therefore Socrates is mortal.”
(Famous syllogisms by Aristotle.)

(1) Informal description ⇝
(2) Formal description ⇝
(3) Problem solution

(2) is often problematic due to under-specification

(3) is deduction (correct inferences): only enumerable, but not decidable
3. Turing Test:

http://cogsci.ucsd.edu/~asaygin/tt/ttest.html

http://www.loebner.net/Prizef/loebner-prize.html

- Standard Turing Test
- Total Turing Test

Turing believed in 1950:

In 2000 a computer with $10^9$ memory-units could be programmed such that it can chat with a human for 5 minutes and pass the Turing Test with a probability of 30%.
4. In item 2. correct inferences were mentioned.

Often not enough information is available in order to act in a way that makes sense (to act in a provably correct way).

\[ \text{Non-monotonic logics.} \]

The world is in general under-specified. It is also impossible to act rationally without correct inferences: reflexes.
To pass the total Turing Test, the computer will need:

- **computer vision** to perceive objects,
- **robotics** to move them about.

The question *Are machines able to think?* leads to 2 theses:

- **Weak AI thesis**: Machines can be built, that *act as if they were intelligent*.
- **Strong AI thesis**: Machines that act in an intelligent way *do possess cognitive states*, i.e. *mind*.
The Chinese Chamber

was used in 1980 by Searle to demonstrate that a system can pass the Turing Test but disproves the Strong AI Thesis. The chamber consists of:

- **CPU**: an English-speaking man without experiences with the Chinese language,
- **Program**: a book containing rules formulated in English which describe how to translate Chinese texts,
- **Memory**: sufficient pens and paper

Papers with Chinese texts are passed to the man, which he translates using the book.

**Question**

Does the chamber understand Chinese?
1.2 From Plato To Zuse
450 BC: Plato, Socrates, Aristotle

Sokr.: "What is characteristic of piety which makes all actions pious?"
Aris.: "Which laws govern the rational part of the mind?"

800 : Al Chwarizmi (Arabia): Algorithm

1300 : Raymundus Lullus: Ars Magna

1350 : William van Ockham: Ockham’s Razor
"Entia non sunt multiplicanda praeter necessitatem."
1596–1650: R. Descartes: 
*Mind = physical system*
Free will, dualism

1623–1662: B. Pascal, W. Schickard:
Addition-machines
1646–1716: G. W. Leibniz: Materialism, uses ideas of Ars Magna to build a machine for simulating the human mind

1561–1626: F. Bacon: Empirism

1632–1704: J. Locke: Empirism

”Nihil est in intellectu quod non antefuerat in sensu.”

1711–1776: D. Hume: Induction

1724–1804: I. Kant:

”Der Verstand schöpft seine Gesetze nicht aus der Natur, sondern schreibt sie dieser vor.”
1805: **Jacquard**: Loom

1815–1864: **G. Boole**: 
Formal language,
Logic as a *mathematical* discipline

1792–1871: **Ch. Babbage**: 
**Difference Engine**: Logarithm-tables
**Analytical Engine**: with addressable memory, stored programs and conditional jumps
Figure 1.1: Reconstruction of Babbage’s difference engine.
1848–1925: G. Frege: **Begriffsschrift**

2-dimensional notation for PL1

Figure 1.2: A formula from Frege’s Begriffsschrift.
1862–1943: D. Hilbert:
  Famous talk 1900 in Paris: 23 problems
  23rd problem: The Entscheidungsproblem

1872–1970: B. Russell:
  1910: Principia Mathematica
  Logical positivism, Vienna Circle (1920–40)
Idea of *truth* in formal languages

1906–1978: K. Gödel:  
*Completeness theorem* (1930)  
*Incompleteness theorem* (1930/31)  
*Unprovability of theorems* (1936)

1912–1954: A. Turing:  
*Turing-machine* (1936)  
*Computability*

1903–1995: A. Church:  
*λ-Calculus*, *Church-Turing-thesis*
1938/42: First operational programmable computer: Z₁
Z₃ by K. Zuse (Deutsches Museum)
with floating-point-arithmetic.

*Plankalkül*: First high-level programming language

Figure 1.3: Reconstruction of Zuse’s Z3.
1940: First computer "Heath Robinson" built to decipher German messages (Turing)
1943: "Collossus" built from vacuum tubes


1952: IBM: \textit{IBM 701}, first computer to yield a profit (Rochester et al.)
1948: First stored program computer *(The Baby)*
Tom Kilburn (Manchester)
Manchester beats Cambridge by 3 months

Figure 1.4: Reconstruction of Kilburn’s baby.
First program run on **The Baby** in 1948:

![Reconstruction of first executed program on The Baby](image)

**Figure 1.5:** Reconstruction of first executed program on The Baby.
1.3 History of AI
The year 1943:

McCulloch and W. Pitts drew on three sources:

1. **physiology** and function of neurons in the brain,
2. **propositional logic** due to Russell/Whitehead,
3. Turing’s **theory of computation**.

### Model of artificial, connected neurons:

- Any computable function can be computed by some network of neurons.
- All the logical connectives can be implemented by simple net-structures.
The year 1951:

Minsky and Edwards build the first computer based on neuronal networks (Princeton)

The year 1952:

A. Samuel develops programs for checkers that learn to play tournament-level checkers.
The year 1956:

Two-month workshop at Dartmouth organized by McCarthy, Minsky, Shannon and Rochester.

Idea:

Combine knowledge about automata theory, neural nets and the studies of intelligence (10 participants)
Newell und Simon show a reasoning program, the Logic Theorist, able to prove most of the theorems in Chapter 2 of the Principia Mathematica (even one with a shorter proof).

But the Journal of Symbolic Logic rejected a paper authored by Newell, Simon and Logical Theorist. Newell and Simon claim to have solved the venerable mind-body problem.
The term **Artificial Intelligence** is proposed as the name of the new discipline.

**Logic Theorist** is followed by the **General Problem Solver**, which was designed from the start to imitate human problem-solving protocols.
The year 1958: Birthyear of AI

McCarthy joins MIT and develops:

1. **Lisp**, the dominant AI programming language
2. **Time-Sharing** to optimize the use of computer-time
3. **Programs with Common-Sense.**

**Advice-Taker**: A hypothetical program that can be seen as the first complete AI system. Unlike others it embodies general knowledge of the world.
The year 1959:

H. Gelernter: *Geometry Theorem Prover*
The years 1960-1966:

McCarthy concentrates on knowledge-representation and reasoning in formal logic (Robinson’s Resolution, Green’s Planner, Shakey).

Minsky is more interested in getting programs to work and focusses on special worlds, the Microworlds.
- **SAINT** is able to solve integration problems typical of first-year college calculus courses

- **ANALOGY** is able to solve geometric analogy problems that appear in IQ-tests
Blocksworld is the most famous microworld.

Work building on the neural networks of McCulloch and Pitts continued. *Perceptrons* by Rosenblatt and *convergence theorem*:

**Convergence theorem**

An algorithm exists that can adjust the connection strengths of a perceptron to match any input data.
Summary:
Great promises for the future, initial successes but miserable further results.

The year 1966: All US funds for AI research are cancelled.

Inacceptable mistake

The spirit is willing but the flesh is weak.

was translated into

The vodka is good but the meat is rotten.
The years 1966–1974: A dose of reality

Simon 1958: ”In 10 years a computer will be grandmaster of chess.”

Simple problems are solvable due to small search-space. Serious problems remain unsolvable.

Hope:

Faster hardware and more memory will solve everything!

NP-Completeness, S. Cook/R. Karp (1971), $P \neq NP$?
The year 1973:

The **Lighthill report** forms the basis for the decision by the British government to end support for AI research.

Minsky’s book **Perceptrons** proved limitations of some approaches with fatal consequences:

Research funding for neural net research decreased to almost nothing.
The years 1969–79: Knowledge-based systems

General purpose mechanisms are called weak methods, because they use weak information about the domain. For many complex problems it turns out that their performance is also weak.

Idea:

Use knowledge suited to making larger reasoning steps and to solving typically occurring cases on narrow areas of expertise.

Example DENDRAL (1969)

Leads to expert systems like MYCIN (diagnosis of blood infections).
’73: PROLOG
’74: Relational databases (Codd)
81-91: Fifth generation project
’91: Dynamic Analysis and Replanning Tool (DART) paid back DARPA’s investment in AI during the last 30 years
’97: IBM’s Deep Blue
’98: NASA’s remote agent program
Something to laugh about: In 1902 a German poem was translated into Japanese. The Japanese version was translated into French. At last this version was translated back into German, assuming that it was a Japanese poem.

The result:

Stille ist im Pavillon aus Jade,
Krähnen fliegen stumm
Zu beschneiten Kirscbäumen im Mondlicht.
Ich sitze
und weine.

What is the original poem?
1.4 Intelligent Agents
Definition 1.1 (Agent a)

An agent \(a\) is anything that can be viewed as *perceiving* its environment through *sensor* and *acting* upon that environment through *effectors*.
Definition 1.2 (Rational Agent, Omniscient Agent)

A *rational agent* is one that does the *right thing* (*Performance measure* determines how successful an agent is).

A *omniscient agent* knows the actual outcome of his actions and can act accordingly.

**Attention:**

A rational agent is in general not omniscient!
Question
What is the right thing and what does it depend on?

1. **Performance measure** (as objective as possible).
2. **Percept sequence** (everything the agent has received so far).
3. **The agent’s knowledge** about the environment.
4. **How** the agent can act.
Definition 1.3 (Ideal Rational Agent)

For each possible percept-sequence an ideal rational agent should do whatever action is expected to maximize its performance measure (based on the evidence provided by the percepts and built-in knowledge).

Note the performance measure is something outside the agent. It allows to compare agents performances.
Mappings:

set of percept sequences $\rightarrow$ set of actions

can be used to describe agents in a mathematical way.

Remark:

Internally an agent is

agent = architecture + program

AI is engaged in designing agent programs
<table>
<thead>
<tr>
<th>Agent Type</th>
<th>Performance Measure</th>
<th>Environment</th>
<th>Actuators</th>
<th>Sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medical diagnosis system</td>
<td>Healthy patient, minimize costs, lawsuits</td>
<td>Patient, hospital, staff</td>
<td>Display questions, tests, diagnoses, treatments, referrals</td>
<td>Keyboard entry of symptoms, findings, patient’s answers</td>
</tr>
<tr>
<td>Satellite image analysis system</td>
<td>Correct image categorization</td>
<td>Downlink from orbiting satellite</td>
<td>Display categorization of scene</td>
<td>Color pixel arrays</td>
</tr>
<tr>
<td>Part-picking robot</td>
<td>Percentage of parts in correct bins</td>
<td>Conveyor belt with parts; bins</td>
<td>Jointed arm and hand</td>
<td>Camera, joint angle sensors</td>
</tr>
<tr>
<td>Interactive English tutor</td>
<td>Maximize student’s score on test</td>
<td>Set of students, testing agency</td>
<td>Display exercises, suggestions, corrections</td>
<td>Keyboard entry</td>
</tr>
</tbody>
</table>

Table 1.2: Examples of agents types and their **PEAS** descriptions.
A simple agent program:

\[
\text{function } \text{SKELETON-AGENT}(\text{percept}) \text{ returns action} \\
\text{static: memory, the agent’s memory of the world} \\
memory \leftarrow \text{UPDATE-MEMORY}(\text{memory, percept}) \\
action \leftarrow \text{CHOOSE-BEST-ACTION}(\text{memory}) \\
memory \leftarrow \text{UPDATE-MEMORY}(\text{memory, action}) \\
\text{return } action
\]
In theory everything is trivial:

function TABLE-DRIVEN-AGENT(percept) returns action

static: percepts, a sequence, initially empty

table, a table, indexed by percept sequences, initially fully specified

append percept to the end of percepts

action ← LOOKUP(percepts, table)

return action
An agent example – taxi driver:

<table>
<thead>
<tr>
<th>Agent Type</th>
<th>Performance Measure</th>
<th>Environment</th>
<th>Actuators</th>
<th>Sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Taxi driver</td>
<td>Safe, fast, legal, comfortable trip, maximize profits</td>
<td>Roads, other traffic, pedestrians, customers</td>
<td>Steering, accelerator, brake, signal, horn, display</td>
<td>Cameras, sonar, speedometer, GPS, odometer, accelerometer, engine sensors, keyboard</td>
</tr>
</tbody>
</table>

Table 1.3: **PEAS** description of the environment for an automated taxi
Some examples:

1. **Production rules**: If the driver in front hits the breaks, then hit the breaks too.

```plaintext
function SIMPLE-REFLEX-AGENT(percept) returns action
static: rules, a set of condition-action rules

state ← INTERPRET-INPUT(percept)
rule ← RULE-MATCH(state, rules)
action ← RULE-ACTION[rule]
return action
```
A first mathematical description

At first, we want to keep everything as simple as possible.

Agents and environments

An agent is **situated** in an environment and can **perform** actions

\[ A := \{ a_1, \ldots, a_n \} \quad \text{(set of actions)} \]

and **change** the state of the environment

\[ S := \{ s_1, s_2, \ldots, s_n \} \quad \text{(set of states)} \]
How does the environment (the state $s$) develop when an action $a$ is executed?

We describe this with a function

$$\text{env} : S \times A \rightarrow 2^S.$$ 

This includes non-deterministic environments.
How do we describe agents?

We could take a function action : S → A.
Question:
How can we describe an agent, now?

Definition 1.4 (Purely Reactive Agent)
An agent is called purely reactive, if its function is given by

\[
\text{action} : S \rightarrow A.
\]
This is too weak!

Take the whole history (of the environment) into account:

\[ S_0 \rightarrow a_0 \quad S_1 \rightarrow a_1 \quad \ldots \quad S_n \rightarrow a_n \quad \ldots \]

The same should be done for env!
This leads to agents that take the whole sequence of states into account, i.e.

\[ \text{action : } S^\ast \rightarrow A. \]

We also want to consider the actions performed by an agent. This requires the notion of a run (next slide).
We define the run of an agent in an environment as a sequence of interleaved states and actions:

**Definition 1.5 (Run \( r, R = R^{act} \cup R^{state} \))**

A run \( r \) over \( A \) and \( S \) is a finite sequence

\[
\begin{align*}
    r : s_0 \rightarrow a_0 & \rightarrow s_1 \rightarrow a_1 \ldots s_n \rightarrow a_n \ldots
\end{align*}
\]

Such a sequence may end with a state \( s_n \) or with an action \( a_n \): we denote by \( R^{act} \) the set of runs ending with an action and by \( R^{state} \) the set of runs ending with a state.
Definition 1.6 (Environment, 2. version)

An **environment** $Env$ is a triple $\langle S, s_0, \tau \rangle$ consisting of

1. the set $S$ of states,
2. the initial state $s_0 \in S$,
3. a function $\tau : R^{act} \rightarrow 2^S$, which describes how the environment changes when an action is performed (given the whole history).
Definition 1.7 (Agent system $a$)

An agent $a$ is determined by a function

$$\text{action} : R^{state} \rightarrow A,$$

describing which action the agent performs, given its current history.

An agent system is then a pair $a = \langle \text{action}, Env \rangle$ consisting of an agent and an environment.

We denote by $R(a, Env)$ the set of runs of agent $a$ in environment $Env$. 
Definition 1.8 (Characteristic Behaviour)

The characteristic behaviour of an agent \( a \) in an environment \( Env \) is the set \( R \) of all possible runs \( r : s_0 \rightarrow a_0 \, s_1 \rightarrow a_1 \, \ldots \, s_n \rightarrow a_n \, \ldots \) with:

1. for all \( n \): \( a_n = \text{action}(\langle s_0, a_0 \ldots, a_{n-1}, s_n \rangle) \),
2. for all \( n > 0 \): \( s_n \in \tau(s_0, a_0, s_1, a_1, \ldots, s_{n-1}, a_{n-1}) \).

For deterministic \( \tau \), the relation “\( \in \)” can be replaced by “\( = \)”.

Prof. Dr. Jürgen Dix

Clausthal, SS 2011
Important:

The formalization of the characteristic behaviour is dependent on the concrete **agent type**. Later we will introduce further behaviours (and corresponding **agent designs**).
Definition 1.9 (Equivalence)

Two agents $a$, $b$ are called behaviourally equivalent wrt. environment $Env$, if

$$R(a, Env) = R(b, Env).$$

Two agents $a$, $b$ are called behaviourally equivalent, if they are behaviourally equivalent wrt. all possible environments $Env$. 
So far so good, but...

What is the problem with all these agents and this framework in general?

Problem

All agents have **perfect information** about the environment!
(Of course, it can also be seen as feature!)
We need more realistic agents!

Note

In general, agents only have **incomplete/uncertain** information about the environment!

We extend our framework by **perceptions**:

**Definition 1.10 (Actions A, Percepts P, States S)**

\[ A := \{ a_1, a_2, \ldots, a_n \} \text{ is the set of actions.} \]
\[ P := \{ p_1, p_2, \ldots, p_m \} \text{ is the set of percepts.} \]
\[ S := \{ s_1, s_2, \ldots, s_l \} \text{ is the set of states} \]
Sensors don’t need to provide perfect information!

Agent

Sensors

What the world is like now

Condition–action rules

What action I should do now

Effectors

Environment
How can agent programs be designed?

There are four types of agent programs:

- Simple reflex agents
- Agents that keep track of the world
- Goal-based agents
- Utility-based agents
First try

We consider a purely reactive agent and just replace states by perceptions.

Definition 1.11 (Simple Reflex Agent)

An agent is called simple reflex agent, if its function is given by

\[
\text{action} : P \rightarrow A.
\]
A very simple reflex agent

function SIMPLE-REFLEX-AGENT(percept) returns action

static: rules, a set of condition-action rules

state ← INTERPRET-INPUT(percept)
rule ← RULE-MATCH(state, rules)
action ← RULE-ACTION[rule]
return action
A simple reflex agent with memory

function Reflex-Agent-With-State(percept) returns action

static: state, a description of the current world state

rules, a set of condition-action rules

state ← Update-State(state, percept)
rule ← Rule-Match(state, rules)
action ← Rule-Action[rule]
state ← Update-State(state, action)
return action
As before, let us now consider sequences of percepts:

**Definition 1.12 (Standard Agent $\alpha$)**

A *standard agent* $\alpha$ is given by a function

$$\text{action} : P^* \rightarrow A$$

together with

$$\text{see} : S \rightarrow P.$$ 

An agent is thus a pair $\langle \text{see}, \text{action} \rangle$. 
Definition 1.13 (Indistinguishable)

Two different states $s, s'$ are **indistinguishable** for an agent $a$, if $\text{see}(s) = \text{see}(s')$.

The relation “indistinguishable” on $S \times S$ is an **equivalence** relation.

What does $|\sim| = |S|$ mean?

And what $|\sim| = 1$?

The characteristic behaviour has to match with the agent design!
Definition 1.14 (Characteristic Behaviour)

The characteristic behaviour of a standard agent $\langle\text{see}, \text{action}\rangle$ in an environment $Env$ is the set of all finite sequences $p_0 \rightarrow a_0 \ p_1 \rightarrow a_1 \ \ldots \ p_n \rightarrow a_n \ \ldots$

where

$p_0 = \text{see}(s_0),$

$a_i = \text{action}(\langle p_0, \ldots, p_i \rangle),$

$p_i = \text{see}(s_i), \text{ where } s_i \in \tau(s_0, a_0, s_1, a_1, \ldots, s_{i-1}, a_{i-1}).$

Such a sequence, even if deterministic from the agent’s viewpoint, may cover different environmental behaviours (runs):

$s_0 \rightarrow a_0 \ s_1 \rightarrow a_1 \ \ldots \ s_n \rightarrow a_n \ \ldots$
Instead of using the whole history, resp. \( P^* \), one can also use **internal states**

\[ I := \{ i_1, i_2, \ldots, i_n, i_{n+1}, \ldots \} \].

**Definition 1.15 (State-based Agent \( a_{state} \))**

A **state-based** agent \( a_{state} \) is given by a function

\[ \text{action} : I \rightarrow A \]

together with

\[ \text{see} : S \rightarrow P, \]

and \[ \text{next} : I \times P \rightarrow I. \]

Here \( \text{next}(i, p) \) is the successor state of \( i \) if \( p \) is observed.
1.4 Intelligent Agents

**Agent**

- **State**
- **How the world evolves**
- **What my actions do**
- **Condition–action rules**

**Environment**

- **Sensors**
- **What the world is like now**
- **What action I should do now**
- **Effectors**
Definition 1.16 (Characteristic Behaviour)

The characteristic behaviour of a state-based agent $a_{state}$ in an environment $Env$ is the set of all finite sequences

$$(i_0, p_0) \rightarrow_{a_0} (i_1, p_1) \rightarrow_{a_1} \cdots \rightarrow_{a_{n-1}} (i_n, p_n), \ldots$$

with

- $p_0 = \text{see}(s_0)$,
- $p_i = \text{see}(s_i)$, where $s_i \in \tau(s_0, a_0, s_1, a_1, \ldots, s_{i-1}, a_{i-1})$,
- $a_n = \text{action}(i_{n+1})$,
- $\text{next}(i_n, p_n) = i_{n+1}$.

Sequence covers the runs $r : s_0 \rightarrow_{a_0} s_1 \rightarrow_{a_1} \cdots$ where

- $a_j = \text{action}(i_{j+1})$,
- $s_j \in \tau(s_0, a_0, s_1, a_1, \ldots, s_{j-1}, a_{j-1})$,
- $p_j = \text{see}(s_j)$.
Are state-based agents more expressive than standard agents? How to measure?

**Definition 1.17 (Environmental Behaviour of \( a_{state} \))**

The *environmental behaviour* of an agent \( a_{state} \) is the set of possible runs covered by the characteristic behaviour of the agent.
Theorem 1.18 (Equivalence)

Standard agents and state-based agents are equivalent with respect to their environmental behaviour.

More precisely: For each state-based agent $a_{\text{state}}$ and next storage function there exists a standard agent $a$ which has the same environmental behaviour, and vice versa.
3. Goal based agents:

Will be discussed in **Planning** (Chapter 9).
Intention:
We want to tell our agent what to do, without telling it how to do it!

We need a utility measure.
How can we define a utility function?

Take for example

\[ u : S \rightarrow \mathbb{R} \]

Question:

What is the problem with this definition?

Better idea:

\[ u : \mathbb{R} \rightarrow \mathbb{R} \]

Take the set of runs \( \mathbb{R} \) and not the set of states \( S \).

How to calculate the utility if the agent has incomplete information?
4. Agents with a utility measure:

The utility measure can often be simulated through a set of goals. It would be helpful, if the utility is close to the performance measure.
Example 1.19 (Tileworld)

(a)  

(b)  

(c)  

\[ H \]

\[ \uparrow \]

\[ T \]

\[ \square T \]

\[ \square T \]

\[ \square T \]

\[ \square T \]

\[ \square T \]

\[ \square T \]

\[ \square T \]

\[ \square T \]
Question:
How do properties of the environment influence the design of an agent?
Some important properties:

**Fully/partially observable:** If the environment is not completely observable the agent will need *internal states*.

**Deterministic/stochastic:** If the environment is only partially observable, then it may appear stochastic (while it is deterministic).

**Episodic/nonepisodic:** Percept-action-sequences are independent. The agent’s experience is divided into *episodes.*
**Static/dynamic:** The environment can change while an agent is deliberating. An environment is *semidynamic* if it does not change with the passage of time but the performance measure does.

**Discrete/continuous:** If there is a limited number of percepts and actions the environment is discrete.

**Single/multi agents:** Is there just one agent or are there several interacting with each other.
### Table 1.4: Examples of task environments and their characteristics

<table>
<thead>
<tr>
<th>Task Environment</th>
<th>Observ.</th>
<th>In/Determ.</th>
<th>Episodic</th>
<th>Static</th>
<th>Discrete</th>
<th>Agents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chess with a clock</td>
<td>Fully</td>
<td>Strategic</td>
<td>Seq.</td>
<td>Semi</td>
<td>Discr.</td>
<td>Multi</td>
</tr>
<tr>
<td>Taxi driving</td>
<td>Part.</td>
<td>Stochastic</td>
<td>Seq.</td>
<td>Dyn.</td>
<td>Cont</td>
<td>Multi</td>
</tr>
<tr>
<td>Medical diagnosis</td>
<td>Part.</td>
<td>Stochastic</td>
<td>Seq.</td>
<td>Dyn.</td>
<td>Cont</td>
<td>Single</td>
</tr>
<tr>
<td>Image-analysis</td>
<td>Fully</td>
<td>Determ.</td>
<td>Epis.</td>
<td>Semi</td>
<td>Cont</td>
<td>Single</td>
</tr>
<tr>
<td>Refinery controller</td>
<td>Part.</td>
<td>Stochastic</td>
<td>Seq.</td>
<td>Dyn.</td>
<td>Cont</td>
<td>Single</td>
</tr>
<tr>
<td>Interactive tutor</td>
<td>Part.</td>
<td>Stochastic</td>
<td>Seq.</td>
<td>Dyn.</td>
<td>Discr.</td>
<td>Multi</td>
</tr>
</tbody>
</table>
2. Searching

- Problem Formulation
- Uninformed search
- Best-First Search
- A* Search
- Heuristics
- Limited Memory
- Iterative Improvements
- Online Search
Searching: Search Algorithms are perhaps the most basic notion of AI. Almost any problem can be formulated as a search problem.

(Un-) informed: We distinguish between uninformed and informed search algorithms. In the latter case, there is information available to guide the search. Often, this results in algorithms that are exponentially better than uninformed ones.
Content of this chapter (2):

**A***: The **A*** algorithm is one of the fundamental informed search algorithms in AI. We discuss several variants based on **Tree-Search** or **Graph-Search** and discuss their correctness and completeness. We also consider **Heuristics**.  

**Memory**: We discuss several variants which use only limited memory: **IDA***, **RBFS**, and **SMA***.  

**Extensions**: Finally, we conclude with a few words about **iterative improvements** (genetic algorithms or simulated annealing) and **online algorithms** (where actions and search are interleaved). We present the **LRTA*** algorithm.
Wanted: Problem-solving agents, which form a subclass of the goal-oriented agents.

Structure: Formulate, Search, Execute.
Formulate:

- **Goal**: a set of states,
- **Problem**: States, actions mapping from states into states, transitions

**Search**: Which *sequence of actions* is helpful?

**Execute**: The resulting sequence of actions is **applied** to the initial state.
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
inputs: percept, a percept
static: seq, an action sequence, initially empty
        state, some description of the current world state
        goal, a goal, initially null
        problem, a problem formulation

state ← UPDATE-STATE(state, percept)
if seq is empty then do
    goal ← FORMULATE-GOAL(state)
    problem ← FORMULATE-PROBLEM(state, goal)
    seq ← SEARCH(problem)
    action ← FIRST(seq)
    seq ← REST(seq)
return action

Table 2.5: A simple problem solving agent.

When executing, percepts are ignored.
2.1 Problem Formulation
We distinguish four types:

1. **1-state-problems**: Actions are completely described. Complete information through sensors to determine the actual state.

2. **Multiple-state-problems**: Actions are completely described, but the initial state is not certain.

3. **Contingency-problems**: Sometimes the result is not a fixed sequence, so the complete tree must be considered.

   (⇝ Excercise: Murphy’s law, ⇝ Chapter 9. Planning)

4. **Exploration-problems**: Not even the effect of each action is known. You have to search in the world instead of searching in the abstract model.
Table 2.6: The vacuum world.
Definition 2.1 (1-state-problem)

A **1-state-problem** consists of:

- a set of **states** (incl. the *initial state*)
- a set of *n* **actions** (*operators*), which – applied to a state – leads to an other state:

  \[
  \text{Operator}_i : \text{States} \rightarrow \text{States}, \ i = 1, \ldots, n
  \]

We use a function **Successor-Fn**: \( S \rightarrow 2^{A \times S} \). It assigns each state a set of pairs \( \langle a, s \rangle \): the set of possible actions and the state it leads to.

- a set of **goal states** or a *goal test*, which – applied on a state – determines if it is a goal-state or not.
- a **cost function** \( g \), which assesses every path in the state space (set of reachable states) and is usually *additive*. 
What about multiple-state-problems? 
(⇝ excercise)

How to choose actions and states?
(⇝ abstraction)

**Definition 2.2 (State Space)**

The **state space** of a problem is the **set of all reachable states** (from the initial state). It forms a directed graph with the states as nodes and the arcs the actions leading from one state to another.
2 Searching

2.1 Problem Formulation

Start State

Goal State

Table 2.7: The 8-puzzle.
Table 2.8: The 8-queens problem.
Table 2.9: State Space for Vacuum world.

Missionaries vs. cannibals
Table 2.10: Belief Space for Vacuum world without sensors.
Real-world-problems:

- travelling-salesman-problem
- VLSI-layout
- labelling maps
- robot-navigation
2.2 Uninformed search
Table 2.11: Map of Romania

Choose, test, expand  RBFS
Principle: Choose, test, expand.

Search-tree

(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu
function TREE-SEARCH(problem, strategy) returns a solution, or failure
initialize the search tree using the initial state of problem
loop do
  if there are no candidates for expansion then return failure
  choose a leaf node for expansion according to strategy
  if the node contains a goal state then return the corresponding solution
  else expand the node and add the resulting nodes to the search tree

Table 2.12: Tree Search.
Important:

**State-space versus search-tree:**

The search-tree is *countably infinite* in contrast to the *finite* state-space.

- a **node** is a *bookkeeping data structure* with respect to the problem instance and with respect to an algorithm;
- a **state** is a *snapshot* of the world.
Definition 2.3 (Datatype Node)

The datatype node is defined by state (∈ S), parent (a node), action (also called operator) which generated this node, path-costs (the costs to reach the node) and depth (distance from the root).

Important:

The recursive dependency between node and parent is important. If the depth is left out then a special node root has to be introduced.

Conversely the root can be defined by the depth: root is its own parent with depth 0.
Figure 2.6: Illustration of a node in the 8-puzzle.
Instantiating Tree-SEARCH

Design-decision: Queue

Tree-SEARCH generates nodes. Among them are those that are-to-be-expanded later on. Rather than describing them as a set, we use a queue instead.
The fringe is the set of generated nodes that have not yet been expanded.

Here are a few functions operating on queues:

Make-Queue(Elements)  Remove-First(Queue)
Empty?(Queue)        Insert(Element, Queue)
First(Queue)            Insert-All(Elements, Queue)
function Tree-Search(problem, fringe) returns a solution, or failure

fringe ← Insert(Make-Node(INITIAL-State[problem]), fringe)
loop do
  if EMPTY?(fringe) then return failure
  node ← REMOVE-FIRST(fringe)
  if GOAL-TEST[problem] applied to STATE[node] succeeds
     then return SOLUTION(node)
  fringe ← INSERT-ALL(EXPAND(node, problem), fringe)

function Expand(node, problem) returns a set of nodes

successors ← the empty set
for each ⟨action, result⟩ in SUCCESSOR-Fn[problem](STATE[node]) do
  s ← a new NODE
  STATE[s] ← result
  PARENT-NODE[s] ← node
  ACTION[s] ← action
  PATH-COST[s] ← PATH-COST[node] + STEP-COST(node, action, s)
  DEPTH[s] ← DEPTH[node] + 1
  add s to successors
return successors

Table 2.13: Tree-Search
Question:

Which are interesting requirements of search-strategies?

- completeness
- time-complexity
- space complexity
- optimality (w.r.t. path-costs)

We distinguish:

Uninformed vs. informed search.
Definition 2.4 (Completeness, optimality)

A search strategy is called

- **complete**, if it *finds a solution*, provided there exists one at all.
- **optimal**, if whenever it produces an output, this output is *an optimal solution*, i.e. one with the smallest path costs among all solutions.

Is any optimal strategy also complete? Vice versa?
Breadth-first search: “nodes with the smallest depth are expanded first”,

Make-Queue: add new nodes at the end: FIFO

Complete? Yes.
Optimal? Yes, if all operators are equally expensive.

Constant branching-factor $b$: for a solution at depth $d$ we have generated\(^1\)(in the worst case)

\[ b + b^2 + \ldots + b^d + (b^{d+1} - b) \text{-many nodes.} \]

Space complexity = Time Complexity

\(^1\) Note this is different from “expanded”. 
Figure 2.7: Illustration of Breadth-First Search.
### Table 2.14: Time versus Memory.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
<th>Time</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1 millisecond</td>
<td>100 bytes</td>
</tr>
<tr>
<td>2</td>
<td>111</td>
<td>.1 seconds</td>
<td>11 kilobytes</td>
</tr>
<tr>
<td>4</td>
<td>11,111</td>
<td>11 seconds</td>
<td>1 megabyte</td>
</tr>
<tr>
<td>6</td>
<td>$10^6$</td>
<td>18 minutes</td>
<td>111 megabytes</td>
</tr>
<tr>
<td>8</td>
<td>$10^8$</td>
<td>31 hours</td>
<td>11 gigabytes</td>
</tr>
<tr>
<td>10</td>
<td>$10^{10}$</td>
<td>128 days</td>
<td>1 terabyte</td>
</tr>
<tr>
<td>12</td>
<td>$10^{12}$</td>
<td>35 years</td>
<td>111 terabytes</td>
</tr>
<tr>
<td>14</td>
<td>$10^{14}$</td>
<td>3500 years</td>
<td>11,111 terabytes</td>
</tr>
</tbody>
</table>
Uniform-Cost-Search: “Nodes \(n\) with lowest path-costs \(g(n)\) are expanded first”

Make-Queue: new nodes are compared to those in the queue according to their path costs and are inserted accordingly.

Complete? Yes, if each operator increases the path-costs by a minimum of \(\delta > 0\) (see below).

Worst case space/time complexity: \(O(b^{1+\left\lfloor \frac{C^*}{\delta} \right\rfloor})\), where \(C^*\) is the cost of the optimal solution and each action costs at least \(\delta\)
If all operators have the same costs (in particular if $g(n) = \text{depth}(n)$ holds):

**Uniform-cost search**

Uniform-cost search = Breadth-first search.

**Theorem 2.5 (Optimality of Uniform-cost search)**

If $\exists \delta > 0 : g(\text{succ}(n)) \geq g(n) + \delta$ then: Uniform-cost search is optimal.
Depth-first search: “Nodes of the greatest depth are expanded first”,

Make-Queue: LIFO, new nodes are added at the beginning

If branching-factor $b$ is constant and the maximum depth is $m$ then:

- **Space-complexity:** $b \times m$,
- **Time-complexity:** $b^m$. 
Table 2.15: Illustration of Depth-First-Search.
**Depth-limited search:** “search to depth \( d \).”

```
function DEPTH-LIMITED-SEARCH( problem, limit ) returns a solution, or failure/cutoff
return RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[ problem ]), problem, limit )

function RECURSIVE-DLS(node, problem, limit) returns a solution, or failure/cutoff
cutoff_occurred? ← false
if GOAL-TEST[ problem ](STATE[node]) then return SOLUTION(node)
else if DEPTH[node] = limit then return cutoff
else for each successor in EXPAND(node, problem) do
    result ← RECURSIVE-DLS(successor, problem, limit)
    if result = cutoff then cutoff_occurred? ← true
    else if result ≠ failure then return result
if cutoff_occurred? then return cutoff else return failure
```

Table 2.16: Depth-Limited-Search.
In total: $1 + b + b^2 + \ldots + b^{d-1} + b^d + (b^{d+1} - b)$ many nodes.

- **Space-complexity:** $b \times l$,
- **Time-complexity:** $b^l$.

Two different sorts of failures!
Iterative-deepening search: “depth increases”

Table 2.17: Illustration of Iterative-Deepening-Search.
function \textsc{Iterative-Deepening-Search}(\textit{problem}) \textbf{returns} a solution, or failure
inputs: \textit{problem}, a problem

\begin{verbatim}
for \textit{depth} \leftarrow 0 \textbf{to} \infty \textbf{do}
    \textit{result} \leftarrow \textsc{Depth-Limited-Search}(\textit{problem}, \textit{depth})
    \textbf{if} \textit{result} \neq \text{cutoff} \textbf{then return} \textit{result}
\end{verbatim}

Table 2.18: Iterative-Deepening-Search.
How many nodes?

\[ d \times b + (d - 1) \times b^2 + \ldots + 2 \times b^{d-1} + 1 \times b^d. \]

Compare with Breadth-first search:

\[ b + b^2 + \ldots + b^d + (b^{d+1} - b). \]
Attention:
Iterative-deepening search is faster than Breadth-first search because the latter also generates nodes at depth $d + 1$ (even if the solution is at depth $d$).

The amount of revisited nodes is kept within a limit.
Figure 2.8: Bidirectional search.
2 Searching
2.2 Uninformed search

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>Yes(^1)</td>
<td>Yes(^1,2)</td>
<td>No</td>
<td>No</td>
<td>Yes(^1)</td>
<td>Yes(^1,4)</td>
</tr>
<tr>
<td>Time</td>
<td>(O(b^{d+1}))</td>
<td>(O(b^{\lceil C^*/\varepsilon \rceil}))</td>
<td>(O(b^d))</td>
<td>(O(b^l))</td>
<td>(O(b^d))</td>
<td>(O(b^{d/2}))</td>
</tr>
<tr>
<td>Space</td>
<td>(O(b^{d+1}))</td>
<td>(O(b^{\lceil C^*/\varepsilon \rceil}))</td>
<td>(O(bm))</td>
<td>(O(bl))</td>
<td>(O(b^d))</td>
<td>(O(b^{d/2}))</td>
</tr>
<tr>
<td>Optimal</td>
<td>Yes(^3)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes(^3)</td>
<td>Yes(^3,4)</td>
</tr>
</tbody>
</table>

Table 2.19: Complexities of uninformed search procedures. Cells regarding time and space denote the nodes generated.

1 if \(b\) is finite;
2 if step costs \(\geq \varepsilon\) for positive \(\varepsilon\);
3 if step costs are all identical;
4 if both directions use Breadth-first search

\(b\) branching factor, \(b \geq 2\)
\(d\) depth of the shallowest solution, \(d < m\)
\(m\) maximum depth of the search tree
\(l\) depth limit, \(l \leq m\)
\(C^*\) cost of the optimal solution
How to avoid repeated states?

- Can we avoid infinite trees by checking for loops?
- Compare number of states with number of paths in the search tree.
State space vs. Search tree

Rectangular grid: How many different states are reachable within a path of length $d$?

Table 2.20: State space versus Search tree: exponential blow-up.
**Graph-Search** = **Tree-Search** + Loop-checking

**function** `GRAPH-SEARCH(problem, fringe)` **returns** a solution, or failure

- `closed` ← an empty set
- `fringe` ← `INSERT(MAKE-NODE(INITIAL-STATE[problem]), fringe)`

**loop** do
  - **if** `EMPTY?(fringe)` **then** `return` failure
  - `node` ← `REMOVE-FIRST(fringe)`
  - **if** `GOAL-TEST(problem)(STATE[node])` **then** `return` `SOLUTION(node)`
  - **if** `STATE[node]` is not in `closed` **then**
    - add `STATE[node]` to `closed`
    - `fringe` ← `INSERT-ALL(EXPAND(node, problem), fringe)`

---

Table 2.21: Graph-Search.
2.3 Best-First Search
Idea:

Use **problem-specific knowledge** to improve the search.

**Tree-search** is precisely defined. Only freedom: Make-Queue.

Let’s assume we have an evaluation-function $f$ which assigns a value $f(n)$ to each node $n$.

We change Make-Queue as follows

- the nodes with smallest $f$ are located at the beginning of the queue
- thus the queue is sorted wrt. $f$. 
function BEST-FIRST-SEARCH( problem, EVAL-FN) returns a solution sequence
inputs: problem, a problem
Eval-Fn, an evaluation function
Queueing-Fn ← a function that orders nodes by EVAL-FN
return GENERAL-SEARCH( problem, Queueing-Fn)

Table 2.22: Best-First-Search.

What about time and space complexity?
**Best-first search:** here the evaluation-function is
\[ f(n) := \text{expected costs of an optimal path from the state in } n \text{ to a goal state.} \]

The word **optimal** is used with respect to the given cost-function \( g \).

This evaluation-function is also called **heuristic function**, written \( h(n) \).

**Heuristic Function**

We require from all heuristic functions that they assign the value 0 to goal states.
Example 2.6 (path-finding)

$h(n)$ might be defined as the direct-line distance between Bucharest and the city denoted by $n$. 

Straight-line distance to Bucharest

<table>
<thead>
<tr>
<th>City</th>
<th>Distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arad</td>
<td>366</td>
</tr>
<tr>
<td>Bucharest</td>
<td>0</td>
</tr>
<tr>
<td>Craiova</td>
<td>160</td>
</tr>
<tr>
<td>Dobrota</td>
<td>242</td>
</tr>
<tr>
<td>Eforie</td>
<td>161</td>
</tr>
<tr>
<td>Fagaras</td>
<td>178</td>
</tr>
<tr>
<td>Giurgiu</td>
<td>77</td>
</tr>
<tr>
<td>Hirsova</td>
<td>151</td>
</tr>
<tr>
<td>Iasi</td>
<td>226</td>
</tr>
<tr>
<td>Lugoj</td>
<td>244</td>
</tr>
<tr>
<td>Mehadia</td>
<td>241</td>
</tr>
<tr>
<td>Neamet</td>
<td>234</td>
</tr>
<tr>
<td>Oradea</td>
<td>380</td>
</tr>
<tr>
<td>Pitesti</td>
<td>98</td>
</tr>
<tr>
<td>Rimnicu Vilcea</td>
<td>193</td>
</tr>
<tr>
<td>Sibiu</td>
<td>253</td>
</tr>
<tr>
<td>Timisoara</td>
<td>329</td>
</tr>
<tr>
<td>Urziceni</td>
<td>80</td>
</tr>
<tr>
<td>Vaslui</td>
<td>199</td>
</tr>
<tr>
<td>Zerind</td>
<td>374</td>
</tr>
</tbody>
</table>
(a) The initial state

(b) After expanding Arad

(c) After expanding Sibiu

(d) After expanding Fagaras

Table 2.23: Illustration of Best-first search.
Questions:

1. Is Best-first search optimal?
2. Is Best-first search complete?
2.4 A* Search
**Definition of A***

**A* search:** Here the evaluation-function is the *sum* of an heuristic function \( h(n) \) and the real path-costs \( g(n) \):

\[
  f(n) := h(n) + g(n).
\]

So A* search is “best-first + uniform-cost”, because \( h(n_z) = 0 \) holds for final states \( n_z \), as \( f(n_z) = g(n_z) \).
The notion of admissibility

On Slide 140 we required from any heuristic function that its value is 0 for goal nodes. An important generalization of this is that it never overestimates the cost to reach the goal.

**Definition 2.7 (Admissible heuristic function)**

The heuristic function $h$ is called **admissible** if $h(n)$ is always smaller or equal than the optimal costs $h^*$ from $n$ to a goal-node:

$$h(n) \leq h^*(n)$$
Table 2.24: Illustration of A* (1).
(e) After expanding Fagaras

(f) After expanding Pitesti

Table 2.25: Illustration of A* (2).
To show completeness of A* we have to ensure:

1. Never it is the case that an infinite number of nodes is generated in one step (locally finite).
2. There is a $\delta > 0$ such that in each step the path costs increase by at least $\delta$.

These conditions must also hold in the following optimality results.

**Theorem 2.8 (Completeness of A*)**

A* is complete (wrt. Tree Search or Graph Search), if the above two properties are satisfied.
- $f$ is monotone in our example.

- **This does not hold in general.**

- Monotony of $f$ is **not needed** to ensure optimality.

- But if the heuristic function is **admissible**, then we can easily modify $f$ to be monotone (how?) and make the search more efficient.
Theorem 2.9 (Optimality of $A^*$ wrt Tree Search)

$A^*$ is optimal using Tree Search, if the heuristic function $h$ is admissible.
Figure 2.9: Goal-directed contours of A*.
What if we use Graph Search?
The proof breaks down!
The notion of consistency

Definition 2.10 (Consistent heuristic function)

The heuristic function $h$ is called consistent if the following holds for every node $n$ and successor $n'$ of $n$:

$$h(n) \leq \text{cost}(n, a, n') + h(n').$$

Consistency of $h$ implies monotony of $f$. Is the converse also true?
Theorem 2.11 (Optimality of A* wrt Graph Search)

**A* is optimal using Graph Search, if the heuristic function** $h$ **is consistent.**

Is the last theorem also true if we require monotony of $f$ (instead of consistency of $h$)?
Question:
How many nodes does A* store in memory?

Answer:
Virtually always exponentially many with respect to the length of the solution.

It can be shown: As long as the heuristic function is not extremely exact

\[ |h(n) - h^*(n)| < O(\log h^*(n)) \]

the amount of nodes is always exponential with respect to the solution.
For almost every usable heuristic a bad error-estimation holds:

\[ |h(n) - h^*(n)| \approx O(h^*(n)) \]

**Important:**

A*’s problem is space not time.
Important:

A* is even **optimally efficient**: No other optimal algorithm (which expands search-paths beginning with an initial node) **expands less nodes** than A*. 
2.5 Heuristics
Table 2.26: An instance of the 8-puzzle.
Question:
Which branching-factor?

Answer:
Approx. 3 (more exactly $\frac{8}{3}$).

Question:
How many nodes have to be considered?

Answer:
$3^g \approx 10^{10}$
in which $g$ is the amount of moves necessary to get a solution. $g$ is approx. 22.
Checking for loops

But: There are only $9! \approx 10^5$ states!

In other words: Looking at cycles can be very helpful.
Question:
Which heuristic functions come in handy?

**Hamming-distance:** $h_1$ is the amount of numbers which are in the wrong position. I.e. $h_1(start) = 8$.

**Manhattan-distance:** Calculate for every piece the distance to the right position and sum up:

$$h_2 := \sum_{i=1}^{8} \text{(distance of } i \text{ to the right position)}$$

$$h_2(start) = 2 + 3 + 2 + 1 + 2 + 2 + 1 + 2 = 15.$$
Question:
How to determine the quality of a heuristic function? (In a single value)

Definition 2.12 (Effective Branching Factor)
Suppose A* detects an optimal solution for an instance of a problem at depth \(d\) with \(N\) nodes generated. Then we define \(b^*\) via
\[
N + 1 = 1 + b^* + (b^*)^2 + \ldots + (b^*)^d:
\]
the effective branching-factor of A*. 
Attention:

\(b^*\) depends on \(h\) and on the special problem instance.

But for many classes of problem instances \(b^*\) is quite constant.
Table 2.27: Comparing $A^\ast$ (Hamming and Manhattan) with IDS.

<table>
<thead>
<tr>
<th>$d$</th>
<th>IDS</th>
<th>$A^\ast(h_1)$</th>
<th>$A^\ast(h_2)$</th>
<th>$A^\ast(h_1)$</th>
<th>$A^\ast(h_2)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>10</td>
<td>6</td>
<td>6</td>
<td>2.45</td>
<td>1.79</td>
</tr>
<tr>
<td>4</td>
<td>112</td>
<td>13</td>
<td>12</td>
<td>2.87</td>
<td>1.48</td>
</tr>
<tr>
<td>6</td>
<td>680</td>
<td>20</td>
<td>18</td>
<td>2.73</td>
<td>1.34</td>
</tr>
<tr>
<td>8</td>
<td>6384</td>
<td>39</td>
<td>25</td>
<td>2.80</td>
<td>1.33</td>
</tr>
<tr>
<td>10</td>
<td>47127</td>
<td>93</td>
<td>39</td>
<td>2.79</td>
<td>1.38</td>
</tr>
<tr>
<td>12</td>
<td>364404</td>
<td>227</td>
<td>73</td>
<td>2.78</td>
<td>1.42</td>
</tr>
<tr>
<td>14</td>
<td>3473941</td>
<td>539</td>
<td>113</td>
<td>2.83</td>
<td>1.44</td>
</tr>
<tr>
<td>16</td>
<td>–</td>
<td>1301</td>
<td>211</td>
<td>–</td>
<td>1.45</td>
</tr>
<tr>
<td>18</td>
<td>–</td>
<td>3056</td>
<td>363</td>
<td>–</td>
<td>1.46</td>
</tr>
<tr>
<td>20</td>
<td>–</td>
<td>7276</td>
<td>676</td>
<td>–</td>
<td>1.47</td>
</tr>
<tr>
<td>22</td>
<td>–</td>
<td>18094</td>
<td>1219</td>
<td>–</td>
<td>1.48</td>
</tr>
<tr>
<td>24</td>
<td>–</td>
<td>39135</td>
<td>1641</td>
<td>–</td>
<td>1.48</td>
</tr>
</tbody>
</table>

Question:

Is Manhattan better than Hamming?
How to determine a (good) heuristics for a given problem?

- There is no general solution, it always depends on the problem.
- Often one can consider a relaxed problem, and take the precise solution of the relaxed problem as a heuristics for the original one.
Start State

Goal State

Table 2.28: A relaxed version of the 8-puzzle.
Relaxed problem:

Try to bring 1–4 in the right positions, but do not care about all the others.

This heuristics is better than Manhattan distance (for the 8-puzzle).
2.6 Limited Memory
Question:

A* is memory-intensive. What if the memory is limited? What to do if the queue is restricted in its length?

This leads to:

- **IDA**: A* + iterative deepening,
- **RBFS**: Recursive Best-First Search,
- **SMA**: Simplified memory bounded A*. 
IDA*: A* combined with iterative-deepening search. We perform Depth-first search (small memory), but use values of $f$ instead of depth.

So we consider contures: only nodes within the $f$-limit. Concerning those beyond the limit we only store the smallest $f$-value above the actual limit.
function IDA*(problem) returns a solution sequence
inputs: problem, a problem
static: f-limit, the current f-COST limit
root, a node

root ← MAKE-NODE(INITIAL-STATE[problem])
f-limit ← f-COST(root)

loop do
    solution, f-limit ← DFS-CONTOUR(root, f-limit)
    if solution is non-null then return solution
    if f-limit = ∞ then return failure; end

function DFS-CONTOUR(node, f-limit) returns a solution sequence and a new f-COST limit
inputs: node, a node
f-limit, the current f-COST limit
static: next-f, the f-COST limit for the next contour, initially ∞

if f-COST[node] > f-limit then return null, f-COST[node]
if GOAL-TEST[problem](STATE[node]) then return node, f-limit
for each node s in SUCCESSORS(node) do
    solution, new-f ← DFS-CONTOUR(s, f-limit)
    if solution is non-null then return solution, f-limit
    next-f ← MIN(next-f, new-f); end
return null, next-f

Figure 2.10: IDA*. 
Theorem 2.13 (Properties of IDA*)

IDA* is optimal if enough memory is available to store the longest solution-path with costs $\leq f^*$. 
Question:
What about complexity?

**space:** depends on the smallest operator-costs $\delta$, the branching-factor $b$ and the optimal costs $f^*$. $bf^*/\delta$-many nodes are generated (worst case)

**time:** depends on the amount of values of the heuristic function.

If we consider a small amount of values, the last iteration of IDA* will often be like A*.

- Consider a large amount of values. Then only one node per contour will be added: **How many nodes will IDA* expand if A* expands $n$-many?**
- What does this say about the time-complexity of A* and IDA*?
**RBFS:** recursive Depth-first version of Best-First search using only linear memory. Memorizes $f$-value of best alternative path.

$f$-value of each node is replaced by the best (smallest) $f$-value of its children.
function RECURSIVE-BEST-FIRST-SEARCH(problem) returns a solution, or failure

RBFS(problem, MAKE-NODE(INITIAL-STATE[problem]), ∞)

function RBFS(problem, node, f_limit) returns a solution, or failure and a new f-cost limit

if GOAL-TEST[problem](state) then return node

successors ← EXPAND(node, problem)

if successors is empty then return failure, ∞

for each s in successors do

  f[s] ← max(g(s) + h(s), f[node])

repeat

  best ← the lowest f-value node in successors

  if f[best] > f_limit then return failure, f[best]

  alternative ← the second-lowest f-value among successors

  result, f[best] ← RBFS(problem, best, min(f_limit, alternative))

if result ≠ failure then return result

Table 2.29: RBFS
(a) After expanding Arad, Sibiu, and Rimnicu Vilcea

(b) After unwinding back to Sibiu and expanding Fagaras

(c) After switching back to Rimnicu Vilcea and expanding Pitesti

Table 2.30: Illustration of RBFS.
RBFS is optimal and complete under the same assumptions as $A^*$. 

IDA* and RBFS suffer from using too little memory.

We would like to use as much memory as possible. This leads to SMA*
**SMA**: is an extension of A*, which only needs a limited amount of memory.

If there is no space left but nodes have to be expanded, nodes will be removed from the queue:

- those with possibly great $f$-value (forgotten nodes). But their $f$-costs will be stored. Later those nodes will be considered if all other paths lead to higher costs.
Figure 2.11: Illustration of SMA*. 
function SMA*(problem) returns a solution sequence
  inputs: problem, a problem
  static: Queue, a queue of nodes ordered by f-cost

  Queue ← MAKE-QUEUE({MAKE-NODE(INITIAL-STATE[problem])})
  loop do
    if Queue is empty then return failure
    n ← deepest least-f-cost node in Queue
    if GOAL-TEST(n) then return success
    s ← NEXT-SUCCESSOR(n)
    if s is not a goal and is at maximum depth then
      f(s) ← ∞
    else
      f(s) ← MAX(f(n), g(s)+h(s))
    if all of n’s successors have been generated then
      update n’s f-cost and those of its ancestors if necessary
    if SUCCESSORS(n) all in memory then remove n from Queue
    if memory is full then
      delete shallowest, highest-f-cost node in Queue
      remove it from its parent’s successor list
      insert its parent on Queue if necessary
    insert s on Queue
  end

Table 2.31: SMA*.

Prof. Dr. Jürgen Dix
Clausthal, SS 2011
**Theorem 2.14 (Properties of SMA*)**

SMA* is **complete** if enough memory is available to store the shortest solution-path.

SMA* is **optimal** if there is enough memory to store the optimal solution-path.
2.7 Iterative Improvements
Idea:

Considering certain problems only the actual state is important, but not the path leading to it: Local Search problems

Of course this problem is as difficult as you like!
Hill-climbing: gradient-descent (-ascent). Move in a direction at random. Compare the new evaluation with the old one. Move to the new point if the evaluation is better.

Problems:

- **local optima**: (getting lost).
- **plateaux**: (wandering around).
- **ridge**: (detour).
function HILL-CLIMBING(problem) returns a solution state

inputs: problem, a problem
static: current, a node
next, a node

current $\leftarrow$ MAKE-NODE(INITIAL-STATE[problem])

loop do
    next $\leftarrow$ a highest-valued successor of current
    if VALUE[next] < VALUE[current] then return current
    current $\leftarrow$ next
end

Table 2.32: Hill Climbing.
Random Restart Hill Climbing: Start again and again from randomly generated initial states.

- $N$-queens: heuristic function $h$ is the number of pairs of queens that attack each other. Random restart hill climbing works well even for $N = 10^6$. 
Simulated annealing: modified hill-climbing: also bad moves (small evaluation-value) are allowed with the small probability of \( e^{-\frac{\Delta f}{T}} \). It depends on the “temperature” \( T \) ranging from \( \infty \) to 0.
Table 2.33: Simulated Annealing.
function GENETIC-ALGORITHM( population, FITNESS-FN) returns an individual

inputs: population, a set of individuals
        FITNESS-FN, a function that measures the fitness of an individual

repeat
    new_population ← empty set
    loop for i from 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
    population ← new_population
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to FITNESS-FN

function REPRODUCE(x, y) returns an individual

inputs: x, y, parent individuals

n ← LENGTH(x)
 c ← random number from 1 to n
return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))

Table 2.34: Genetic Algorithm.
Table 2.35: Illustration of Genetic Algorithm.

<table>
<thead>
<tr>
<th>(a) Initial Population</th>
<th>(b) Fitness Function</th>
<th>(c) Selection</th>
<th>(d) Crossover</th>
<th>(e) Mutation</th>
</tr>
</thead>
<tbody>
<tr>
<td>24748552</td>
<td>24 31%</td>
<td>32752411</td>
<td>32748552</td>
<td>32748152</td>
</tr>
<tr>
<td>32752411</td>
<td>23 29%</td>
<td>24748552</td>
<td>24752411</td>
<td>24752411</td>
</tr>
<tr>
<td>24415124</td>
<td>20 26%</td>
<td>32752411</td>
<td>32752124</td>
<td>3252124</td>
</tr>
<tr>
<td>32543213</td>
<td>11 14%</td>
<td>24415124</td>
<td>24415411</td>
<td>24415417</td>
</tr>
</tbody>
</table>
Table 2.36: Crossover in the 8-queens problem.

Fitness function: number of non-attacking pairs of queens.
2.8 Online Search
Up to now: **offline search**. What about interleaving actions and computation? Remember the **exploration problem**. One does not know the effect of actions, nor the state space. Therefore one has **to try out all actions** and remember to which states they lead.
Table 2.37: Online vs Offline search.
How to measure the quality of an online algorithm? Number of steps alone does not make sense. One has to **explore the space and find the optimal path**.

**Definition 2.15 (Competitive Ratio)**

The competitive ratio of an online search problem is the costs of the path taken by the agent divided by the costs of the optimal path.
Infinite competitive ratio
Unbounded competitive ratio

$S \quad G \quad \ldots$
We assume the following

1. actions are reversible (why?),
2. once a state has been visited, it will be recognized when it is visited again.

- Compare backtracking in offline versus online search. In online search, we have to find an action to backtrack to the previous state. We cannot take it from the queue!!
- Similarly in online search we can only expand a node that we physically occupy.
- We have to keep a table $\text{result}(a, s)$ listing the effects of actions $a$ executed in state $s$.
- We also have to keep the following two tables:
  1. unexplored: for each state the actions not yet tried,
  2. unbacktracked: for each state the backtracks not yet tried.
function ONLINE-DFS-AGENT(s') returns an action

inputs: s', a percept that identifies the current state

static: result, a table, indexed by action and state, initially empty

unexplored, a table that lists, for each visited state, the actions not yet tried

unbacktracked, a table that lists, for each visited state, the backtracks not yet tried

s, a, the previous state and action, initially null

if GOAL-TEST(s') then return stop

if s' is a new state then unexplored[s'] ← ACTIONS(s')

if s is not null then do

result[a, s] ← s'

add s to the front of unbacktracked[s']

if unexplored[s'] is empty then

if unbacktracked[s'] is empty then return stop

else a ← an action b such that result[b, s'] = POP(unbacktracked[s'])

else a ← POP(unexplored[s'])

s ← s'

return a

Table 2.38: Online DFS Agent.
Hill Climbing is an Online Search algorithm!

What about using random restarts?

What about using random walks?

This certainly works for finite spaces, but not for infinite ones.
Why not simply taking random walks?

Because they can lead to exponentially many steps.

Will a random walk eventually find the goal (completeness)?

\[
\begin{cases}
  \text{Yes,} & \text{if state space is finite;} \\
  \text{Yes,} & \text{for 2-dimensional grids;} \\
  \text{with probability } 0.34, & \text{for 3-dimensional grids.}
\end{cases}
\]

Table 2.39: Random Walk.
Instead of randomness, let’s try to use memory!

- Given a heuristics $h$, it should be used.
- But it can be misleading (local maxima).
- Therefore we build a better, more realistic estimate $H$ that takes $h$ into account and the path that the hill climbing algorithm takes to explore the states.
- This can help to get out of the local maximum.
Hill-Climbing + Memory=LRTA*

Table 2.40: Illustration of LRTA*

(a) 1 ← 8 → 9 → 2 ← 1 → 2 → 4 ← 1 → 3

(b) 1 ← 8 → 9 → 3 ← 1 → 2 → 4 ← 1 → 3

(c) 1 ← 8 → 9 → 3 ← 1 → 4 → 4 ← 1 → 3

(d) 1 ← 8 → 9 → 5 ← 1 → 4 → 4 ← 1 → 3

(e) 1 ← 8 → 9 → 5 ← 1 → 5 ← 1 → 4 ← 1 → 3

Prof. Dr. Jürgen Dix

Clausthal, SS 2011
LRTA*

a) The algorithm got stuck in a local minimum. The best neighbour is right from the current (yellow) state.

b) Therefore the expected costs of the previous state have to be updated (3 instead of 2 because the best neighbour has expected costs 2 and the costs to go there is 1). Therefore the algorithm walks back to this node (it is the best one).

c) In the same way the expected costs of the previous node have to be updated (4 instead of 3).

d) Similarly, the expected costs of the current node have to be updated (5 instead of 4).

e) Finally, the best next node is to the right (4 is better than 5).
function LRTA*-AGENT($s'$) returns an action

inputs: $s'$, a percept that identifies the current state

static: result, a table, indexed by action and state, initially empty

$H$, a table of cost estimates indexed by state, initially empty

$s$, $a$, the previous state and action, initially null

if GOAL-TEST($s'$) then return stop

if $s'$ is a new state (not in $H$) then $H[s'] ← h(s')$

unless $s$ is null

result[$a$, $s$] ← $s'$

$H[s] ← \min_{b ∈ ACTIONS(s)} LRTA*-COST(s, b, result[b, s], H)$

$a ←$ an action $b$ in ACTIONS($s'$) that minimizes $LRTA*-COST(s', b, result[b, s'], H)$

$s ← s'$

return $a$

function LRTA*-COST($s$, $a$, $s'$, $H$) returns a cost estimate

if $s'$ is undefined then return $h(s)$
else return $c(s, a, s') + H[s']$

Table 2.41: LRTA*
3. Supervised Learning

- Basics
- Decision Trees
- Ensemble Learning
- PL1 Formalisations
- PAC Learning
- Noise and overfitting
Content of this chapter (1):

**Learning:** We describe the general structure of a Learning Agent. An agent should be capable of learning new concepts through observing its environment. We distinguish between supervised-, reinforcement- and unsupervised learning.

**Decision Trees:** We describe a simple algorithm built on some general assumption of Shannon’s information theory, to construct decision trees given a table of observations. We apply Ockham’s razor and generate a tree that can be used to make predictions about unseen cases.
Content of this chapter (2):

**Ensembles:** Here we use not a single hypothesis, but an ensemble of hypotheses to make predictions. We describe the **Ada Boost Algorithm**, which often improves the hypothesis enormously.

**Logic:** We discuss another formulation of learning, based on first-order logical formulae: The **Version Space Algorithm**.
Learning in general

**Hitherto:** An agent’s intelligence is in his program, it is *hard-wired*.

**Now:** We want a more autonomous agent, which should *learn through percepts* (experiences) to know its environment.

**Important:** If the domain in which it acts can’t be described completely.
3.1 Basics
3 Supervised Learning

3.1 Basics

Performance standard

Agent

Environment

Critic

Sensors

Learning element

feedback

changes

knowledge

Problem generator

Performance element

Effectors

feedback

learning goals

Prof. Dr. Jürgen Dix
Clausthal, SS 2011
- **Performance element:** Observes and chooses actions. This was the whole agent until now.

- **Critic:** Observes the result of an action and assesses it with respect to an *external* standard.

**Why external?**

Otherwise it would set its own standard so low that it always holds!
Learning element: Modifies the performance element by considering the critic (and architecture of the performance element). It also creates new goals (to improve understanding effects of actions).

Problem generator: It proposes the execution of actions to satisfy the goals of the learning element. These do not have to be the “best” actions (wrt. performance element): but they should be informative and deliver new knowledge about the world.

Example: Driving a taxi.
Question:
What does learning (the design of the learning element) really depend on?

1. Which components of the performance element should be improved?
2. How are these components represented?

(⇝ Slide 224 (Decision trees), ⇝ Slide 248 (Ensemble Learning), ⇝ Slide 257 (Domains formalised in PL1))
3. Which **kind of feedback**?

- **Supervised learning**: The execution of an incorrect action leads to the “right” solution as feedback (e.g. How intensively should the brakes be used?).

- **Driving instructor**

- **Reinforcement learning**: Only the result is perceived. Critic tells, if good or bad, but not what would have been right.

- **Unsupervised learning**: No hints about the right actions.

4. Which **a-priori-information** is there? (Often there is useful background knowledge)
Components of the performance element:

1. Mappings from the conditions of the actual state into the set of actions
2. Deriving relevant properties of the world from the percepts
3. Information about the development of the world
4. Information about the sequence of actions
5. Assessing-function of the world-states
6. Assessing-function concerning the quality of single actions in one state
7. Description of state-classes, which maximise the utility-function of an agent
Important:

All these components are – from a mathematical point of view – *mappings*.

Learning *means to represent these mappings*. 
Inductive learning: Given a set of pairs $(x, y)$. Find $f$ with $f(x) = y$.

Example 3.1 (Continue a series of numbers)

Which number is next?

- 3, 5, 7, ?
- 3, 5, 17, 257, ?

(a)  
(b)  
(c)  
(d)  

Prof. Dr. Jürgen Dix
Simple reflex agent:

```
global examples ← {}  

function REFLEX-PERFORMANCE-ELEMENT( percept) returns an action

    if (percept, a) in examples then return a
    else
        h ← INDUCE(examples)
        return h(percept)

procedure REFLEX-LEARNING-ELEMENT (percept, action)
    inputs: percept, feedback percept
    action, feedback action

    examples ← examples ∪ {(percept, action)}
```
Example 3.2 (Wason’s Test; Verify and Falsify)

Consider a set of cards. Each card has a letter printed on one side and a number on the other. Having taken a look at some of these cards you formulate the following hypothesis:

If there is a vowel on one side then there is an even number on the other.

Now there are the following cards on the table:

4  T  A  7.

You are allowed to turn around only two cards to check the hypothesis. Which card(s) do you flip?
3.2 Decision Trees
Decision trees represent boolean functions

Small example:

You plan to go out for dinner and arrive at a restaurant. Should you wait for a free table or should you move on?
3.2 Decision Trees

- Patrons?
  - None
  - Some
  - Full
    - No
    - Yes

- WaitEstimate?
  - >60
  - 30–60
  - 10–30
  - 0–10
    - No
    - Yes

- Alternate?
  - No
  - Yes

- Hungry?
  - No
  - Yes

- Reservation?
  - No
  - Yes

- Fri/Sat?
  - No
  - Yes

- Bar?
  - No
  - Yes

- Raining?
  - No
  - Yes

**Learned Tree**

**Decision Tree**
decision tree = conjunction of implications

(implication = path leading to a leaf)

For all restaurants $r$:

$\forall r:\ (\text{Patrons}(r, \text{Full}) \land \text{Wait\_estimate}(r, 10 - 30) \land \neg \text{Hungry}(r))$

$\rightarrow \text{Will\_Wait}(r)$

Attention:

This is written in first order logic but a decision tree talks only about a single object ($r$ above). So this is really propositional logic:

$$\forall r: (\text{Patrons}_{r}^{\text{Full}} \land \text{Wait\_estimate}_{r}^{10-30} \land \neg \text{Hungry}_{r})$$

$\rightarrow \text{Will\_Wait}_{r}$
Question:

boolean functions = decision trees?

Answer:

Yes! Each row of the table describing the function belongs to one path in the tree.

Attention

Decision trees can be much smaller! But there are boolean function which can only be represented by trees with an exponential size:

Parity function: \( par(x_1, \ldots, x_n) := \begin{cases} 1, & \text{if } \sum_{i=1}^{n} x_i \text{ is even} \\ 0, & \text{else} \end{cases} \)
Variant of decision-trees

Example 3.3 (Decision List)

All attributes are boolean. A decision list is a tree of the following form

\[
\begin{array}{cccc}
\text{TEST 1} & \text{no} & \text{TEST 2} & \text{no} \\
\text{yes} & \text{yes} & \ldots & \text{yes} \\
\text{Answer 1} & \text{Answer 2} & \ldots & \text{Answer e} \\
\text{Answer e+1} & \\
\end{array}
\]

with \( \text{Answer}_i \in \{\text{Yes, No}\} \) and Test\(_i\) a conjunction of (possibly negated) attributes (\( \leadsto \) Exercise: Compare decision trees and decision lists).

\( k-\text{DL}(n) \) is the set of boolean functions with \( n \) attributes, which can be represented by decision lists with at most \( k \) checks in each test.
Obviously:

- $n-DL(n)$ = set of all boolean functions
- $\text{card}(n-DL(n)) = 2^{2^n}$. 
### Question: Table of examples

**How should decision trees be learned?**

<table>
<thead>
<tr>
<th>Example</th>
<th>Attributes</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alt</td>
<td>Bar</td>
</tr>
<tr>
<td>X₁</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X₂</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X₃</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X₄</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X₅</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X₆</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X₇</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X₈</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>X₉</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X₁₀</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>X₁₁</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>X₁₂</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
The set of **examples-to-be-learned** is called **training set**. Examples can be evaluated **positively** (attribute holds) or **negatively** (attribute does not hold).

**Trivial solution of learning**

The paths in the tree are exactly the examples.

**Disadvantage:**

New cases can not be considered.
Idea:

Choose the **simplest** tree (or rather the **most general**) which is **compatible** with all examples.

**Ockham’s razor**: Entia non sunt multiplicanda praeter necessitatem.
Example 3.4 (Guess!)

A computer program encodes triples of numbers with respect to a certain rule. **Find out that rule.**

You enter triples \((x_1, x_2, x_3)\) of your choice \((x_i \in \mathbb{N})\) and get as answers “yes” or “no”.

Simplification: At the beginning the program tells you that these triples are in the set:

\[(4, 6, 8), (6, 8, 12), (20, 22, 40)\]

Your task:

Make more enquiries (approx. 10) and try to find out the rule.
The idea behind the learning algorithm

**Goal:** A tree which is as small as possible. First test the *most important* attributes (in order to get a quick classification).

This will be formalised later, using *information theory*.

Then *proceed recursively*, i.e. with decreasing amounts of examples and attributes.
We distinguish the following cases:

1. There are positive and negative examples. Choose the best attribute.
2. There are only positive or only negative examples. Done – a solution has been found.
3. There are no more examples. Then a default value has to be chosen, e.g. the majority of examples of the parent node.
4. There are positive and negative examples, but no more attributes. Then the basic set of attributes does not suffice, a decision can not be made. Not enough information is given.
function DECISION-TREE-LEARNING(examples, attributes, default) returns a decision tree
inputs: examples, set of examples
        attributes, set of attributes
        default, default value for the goal predicate

if examples is empty then return default
else if all examples have the same classification then return the classification
else if attributes is empty then return MAJORITY-VALUE(examples)
else
    best ← CHOOSE-ATTRIBUTE(attributes, examples)
    tree ← a new decision tree with root test best
    for each value $v_i$ of best do
        examples$_i$ ← \{elements of examples with best = $v_i$\}
        subtree ← DECISION-TREE-LEARNING(examples$_i$, attributes $-$ best,
                                           MAJORITY-VALUE(examples))
        add a branch to tree with label $v_i$ and subtree subtree
    end
    return tree
3 Supervised Learning

3.2 Decision Trees

Learned Tree

- **Patrons?**
  - None
  - Some
  - Full
    - No
    - Yes

- **Hungry?**
  - No
  - Yes

- **Type?**
  - No
  - Yes
    - French
      - Yes
    - Italian
      - No
    - Thai
      - Fri/Sat?
        - No
        - Yes
          - Burger
            - Yes
          - Fri/Sat?
            - No
            - Yes
The algorithm computes a tree which is as small as possible and consistent with the given examples.

Question:
How good is the generated tree? How different is it from the “actual” tree? Is there an a-priory-estimation? (⇝ PAC learning).
Empiric approach:

1. Chose a set of examples $M_{Ex}$.
2. Divide into two sets: $M_{Ex} = M_{Trai} \cup M_{Test}$.
3. Apply the learning algorithm on $M_{Trai}$ and get a hypothesis $H$.
4. Calculate the amount of correctly classified elements of $M_{Test}$.
5. Repeat 1.-4. for many $M_{Trai} \cup M_{Test}$ with randomly generated $M_{Trai}$.

Attention: Peeking!
3 Supervised Learning

3.2 Decision Trees

![Graph showing the proportion correct on the test set as a function of training set size.](image)

- **X-axis**: Training set size
- **Y-axis**: Proportion correct on test set

The graph illustrates the relationship between the size of the training set and the proportion of correct predictions on the test set.

---

Prof. Dr. Jürgen Dix

Clausthal, SS 2011

240
Question:
How to choose the best attribute? The best attribute is the one that delivers the highest amount of information.

Example: Flipping a coin
3 Supervised Learning

3.2 Decision Trees

Shannon’s theory

**Definition 3.5 (1 bit, information)**

1 bit is the information contained in the outcome of flipping a (fair) coin.

More generally: assume there is an experiment with \( n \) possible outcomes \( v_1, \ldots, v_n \). Each outcome \( v_i \) will result with a probability of \( P(v_i) \). The information encoded in this result (the outcome of the experiment) is defined as follows:

\[
I(P(v_1), \ldots, P(v_n)) := \sum_{i=1}^{n} -P(v_i) \log_2 P(v_i)
\]
Assume the coin is manipulated. With a probability of $90\%$ head will come out. Then

\[ l(0.1, 0.9) = \ldots \approx 0.47 \]
Question:

For each attribute $A$: If this attribute is evaluated with respect to the actual training-set, how much information will be gained this way?

The “best” attribute is the one with the highest gain of information!
Definition 3.6 (Gain of Information)

We gain the following information by testing the attribute $A$:

$$\text{Gain}(A) = \mathcal{I}(\frac{p}{p+n}, \frac{n}{p+n}) - \text{Missing}_\text{Inf}(A)$$

with

$$\text{Missing}_\text{Inf}(A) = \sum_{i=1}^{\nu} \frac{p_i + n_i}{p + n} \mathcal{I}(\frac{p_i}{p_i + n_i}, \frac{n_i}{p_i + n_i})$$

Choose\_Attribute chooses the $A$ with maximal $\text{Gain}(A)$.
The figure implies $\text{Gain}(\text{Patron}) \approx 0.54$. Calculate $\text{Gain}(\text{Type})$, $\text{Gain}(\text{Hungry})$ ($\text{Hungry}$ as the first attribute), $\text{Gain}(\text{Hungry})$ (with predecessor $\text{Patron}$).
3.3 Ensemble Learning
3 Supervised Learning

3.3 Ensemble Learning

So far:
A single hypothesis is used to make predictions.

Idea:
Let’s take a whole bunch of them (an ensemble).

Motivation:
Among several hypotheses, use majority voting. The misclassified ones get higher weights!
Consider three simple hypotheses. No one is perfect. But all taken together, a **new hypothesis** is created (which is not constructible by the original method).
3 Supervised Learning

3.3 Ensemble Learning

Prof. Dr. Jürgen Dix
Clausthal, SS 2011
**Weighted Training Set:** Each example gets a weight \( w_j \geq 0 \).

**Initialisation:** All weights are set to \( \frac{1}{n} \).

**Boosting:** Misclassified examples are getting higher weights.

**Iterate:** We get new hypotheses \( h_i \). After we got a certain number \( M \) of them we feed them into the

**Boosting-Algorithm:** It creates a weighted ensemble hypothesis.
function ADABoost(examples, L, M) returns a weighted-majority hypothesis
inputs: examples, set of N labelled examples \((x_1, y_1), \ldots, (x_N, y_N)\)
        L, a learning algorithm
        M, the number of hypotheses in the ensemble
local variables: w, a vector of N example weights, initially 1/N
                h, a vector of M hypotheses
                z, a vector of M hypothesis weights

for \(m = 1\) to \(M\) do
    \(h[m] \leftarrow L(examples, w)\)
    error \leftarrow 0
    for \(j = 1\) to \(N\) do
        if \(h[m](x_j) \neq y_j\) then error \leftarrow error + w[j]
    for \(j = 1\) to \(N\) do
        if \(h[m](x_j) = y_j\) then \(w[j] \leftarrow w[j] \cdot error / (1 - error)\)
    \(w \leftarrow \text{Normalize}(w)\)
    \(z[m] \leftarrow \log (1 - error) / error\)
return \text{Weighted-Majority}(h, z)\)
Theorem 3.7 (Effect of boosting)

Suppose the Learning algorithm has the following property: it always returns a hypothesis with weighted error that is slightly better than random guessing.

Then AdaBOOST will return a hypothesis classifying the training data perfectly for large enough $M$. 
3.3 Ensemble Learning

The graph illustrates the proportion of correct test set predictions for different training set sizes. It compares two methods: Boosted decision stumps (solid line) and Decision stump (dashed line).

- **Proportion correct on test set**
- **Training set size**

The x-axis represents the training set size, ranging from 0 to 100. The y-axis shows the proportion of correct predictions, ranging from 0.5 to 1.0.

- **Boosted decision stumps**
- **Decision stump**

As the training set size increases, the proportion of correct predictions increases for both methods, with Boosted decision stumps generally performing better than the Decision stump.
3 Supervised Learning

3.3 Ensemble Learning

Training error
Test error

Number of hypotheses $M$
3.4 PL1 Formalisations
Goal:
A more general framework.

Idea:
To learn means to search in the hypotheses space (∽ planning).

Goal-predicate:

\[ Q(x), \text{ one-dimensional (hitherto: Will\_Wait)} \]

We seek a \textit{definition of} \( Q(x) \), i.e. a formula \( C(x) \) with

\[ \forall x \ (Q(x) \leftrightarrow C(x)) \]
Each example $X_i$ represents a set of conditions under which $Q(X_i)$ holds or not. We look for an explanation: a formula $C(x)$ which uses all predicates of the examples.

\[ \forall r \quad \text{Will\_Wait}(r) \iff \]
\[ \text{Patrons}(r, \text{Some}) \]
\[ \lor (\text{Patrons}(r, \text{Full}) \land \neg \text{Hungry}(r) \land \text{Type}(r, \text{French})) \]
\[ \lor (\text{Patrons}(r, \text{Full}) \land \neg \text{Hungry}(r) \land \text{Type}(r, \text{Thai}) \land \text{Fri\_Sat}(r)) \]
\[ \lor (\text{Patrons}(r, \text{Full}) \land \neg \text{Hungry}(r) \land \text{Type}(r, \text{Burger})) \]

Note that we require the formulae $C(x)$ to be of a certain form: disjunctions of conjunctions of atomic or negated predicates. The negations of such formulae are also called clauses. They will be defined more precisely on Slide 401 in Chapter 5.
Definition 3.8 (Hypothesis, Candidate Function)

A formula $C_i(x)$ with $\forall x \ (Q(x) \leftrightarrow C_i(x))$ is called candidate function. The whole formula is called hypothesis:

$$H_i : \forall x \ (Q(x) \leftrightarrow C_i(x))$$
Definition 3.9 (Hypotheses-Space $\mathcal{H}$)

The **hypotheses-space** $\mathcal{H}$ of a learning-algorithm is the set of hypotheses the algorithm can create.

The **extension** of a hypothesis $H$ with respect to the goal-predicate $Q$ is the set of examples for which $H$ holds.

**Attention:**

The combination of hypotheses with **different extensions** leads to inconsistency.

Hypotheses with the **same extensions** are logically **equivalent**.
The situation in general:

- We have a set of examples \( \{X_1, \ldots, X_n\} \). We describe each example \( X \) through a clause and the declaration \( Q(X) \) or \( \neg Q(X) \).

<table>
<thead>
<tr>
<th>Example</th>
<th>Attributes</th>
<th>Goal</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alt</td>
<td>Bar</td>
</tr>
<tr>
<td>X_1</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X_2</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X_3</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X_4</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X_5</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>X_6</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X_7</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X_8</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>X_9</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>X_{10}</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>X_{11}</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>X_{12}</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
$X_1$ is defined through

$$t(X_1) \land \neg \text{Bar}(X_1) \land \neg \text{Fri\_Sat}(X_1) \land \text{Hungry}(X_1) \land \ldots \land \text{Will\_Wait}(X_1).$$

Note that $\mathcal{H}$ is the set of hypotheses as defined in Definition 3.8. While it corresponds to decision trees, it is not the same.

The training set is the set of all such conjunctions.
We search for a **hypothesis** which is **consistent with the training set**.

**Question:**

Under which conditions is a hypothesis $H$ **inconsistent** with an example $X$?

**false negative:** Hypothesis says *no* ($\neg Q(X)$) but $Q(X)$ does hold.

**false positive:** Hypothesis says *yes* ($Q(X)$) but $\neg Q(X)$ does hold.

**Attention:**

*Inductive learning* in logic-based domains means to restrict the set of possible hypotheses with every example.
For first order logic: $\mathcal{H}$ is in general infinite, thus automatic theorem proving is much too general.

**1st approach:** We keep one hypothesis and modify it if the examples are inconsistent with it.

**2nd approach:** We keep the whole subspace that is still consistent with the examples (version space). This is effectively represented by two sets (analogical to the representation of a range of real numbers by $[a, b]$).
1st approach: **Current-best-hypothesis Search.** Begin with a simple hypothesis $H$.

- If a new example is consistent with $H$: okay.
- If it is **false negative**: **Generalise** $H$.
- If it is **false positive**: **Specialise** $H$. 

- (a) (b) (c) (d) (e)
This leads to an algorithm:

```
function CURRENT-BEST-LEARNING(examples) returns a hypothesis

    $H \leftarrow$ any hypothesis consistent with the first example in examples

    for each remaining example in examples do
        if $e$ is false positive for $H$ then
            $H \leftarrow$ choose a specialization of $H$ consistent with examples
        else if $e$ is false negative for $H$ then
            $H \leftarrow$ choose a generalization of $H$ consistent with examples
        end

    if no consistent specialization/generalization can be found then fail

return $H$
```
How to generalize/specialize?

$$H_1 : \forall x (Q(x) \leftrightarrow C_1(x))$$
$$H_2 : \forall x (Q(x) \leftrightarrow C_2(x))$$

- **$H_1$ generalises $H_2$, if $\forall x (C_2(x) \rightarrow C_1(x))$,**
- **$H_1$ specialises $H_2$, if $\forall x (C_1(x) \rightarrow C_2(x))$.**

**Generalisation means:** leave out $\land$-elements in a conjunction, add $\lor$-elements to a disjunction.

**Specialisation means:** add $\land$-elements to a conjunction, leave out $\lor$-elements in a disjunction.
2nd approach: **Version-space.**

```
function VERSION-SPACE-LEARNING(examples) returns a version space
    local variables: V, the version space: the set of all hypotheses
    V ← the set of all hypotheses
    for each example e in examples do
        if V is not empty then V ← VERSION-SPACE-UPDATE(V, e)
    end
    return V

definition function VERSION-SPACE-UPDATE(V, e) returns an updated version space
    V ← \{ h ∈ V : h is consistent with e \}
```

```
Problem:

$\mathcal{H}$ is a big disjunction $H_1 \lor \ldots \lor H_n$. How to represent this?

Reminder:

How is the set of real numbers between 0 and 1 represented? Through the range $[0, 1]$.

To solve our problem:

There is a partial order on $\mathcal{H}$ (generalize/specialize). The borders are defined through

- $G$ set: is consistent with all previous examples and there is no more general hypothesis.
- $S$ set: is consistent with all previous examples and there is no more special hypothesis.
Initially, the $G$ set consists of $\top$ (true), and the $S$ set consists of $\bot$ (false). This is because the initial version space should represent all possible hypotheses.

We have to ensure the following:

1. Each consistent hypothesis (except those in $G$ or $S$) is more specific than some member of $G$ and more general than some member of $S$.

2. Each hypothesis that is more specific than some member of $G$ and more general than some member of $S$ is a consistent hypothesis.
When considering new examples the $G$- and $S$-sets must be appropriately modified in VERSION-SPACE-UPDATE (so that the two conditions on the last slide are satisfied).

**False positive for $S_i$:** $S_i$ too general, no consistent specializations: Remove $S_i$ from $S$.

**False negative for $S_i$:** $S_i$ too specific: Replace it by all generalizations that are more specific than some element of $G$.

**False positive for $G_i$:** $G_i$ too general: Replace it by all specializations that are more general than some element of $S$.

**False negative for $G_i$:** $G_i$ too specific, no consistent generalizations: Remove $G_i$ from $G$. 
Question: What can happen?

We iterate the algorithm on the previous slide until one of the following happens:

1. Only one hypothesis remains: That is our solution!
2. The space collapses: There is no consistent hypothesis, $G = \emptyset$ or $S = \emptyset$.
3. No examples are left, but there are still several hypotheses: Result is a big disjunction.
3.5 PAC Learning
Question:

What is the distance between the hypothesis $H$ calculated by the learning algorithm and the real function $f$?

〜〜 computational learning theory: PAC-learning – Probably Approximately Correct.

Idea:

If a hypothesis is consistent with a big training set then it cannot be completely wrong.
Question:

How are the training set and test set related?

We assume:

The elements of the training and test set are taken from the set of all examples with the same probability.
Definition 3.10 (error(h))

Let $h \in \mathcal{H}$ be a hypothesis and $f$ the target (i.e. to-be-learned) function. We are interested in the set

$$\text{Diff}(f, h) := \{ x : h(x) \neq f(x) \}.$$ 

We denote with $\text{error}(h)$ the probability of a randomly selected example being in $\text{Diff}(f, h)$. With $\epsilon > 0$ the hypothesis $h$ is called \( \epsilon \) approximatively correct, if $\text{error}(h) \leq \epsilon$ holds.
Question:

\( \epsilon > 0 \) is given. **How many examples** must the training set contain to make sure that the **hypothesis** created by a learning algorithm is \( \epsilon \) approximatively correct?

**Question is wrongly stated!**
Different sets of examples lead to different propositions and with it to different values of $\epsilon$. It depends not only on **how many** but also **which** examples are chosen. For this reason we reformulate our question more carefully.

**Question: More carefully and precisely stated.**

Let $\epsilon > 0$ and $\delta > 0$ be given. **How many examples must the training-set contain to make sure that the hypothesis computed by a learning-algorithm is $\epsilon$ approximatively correct with a probability of at least $1 - \delta$?**
We want to abstract from particular learning-algorithms and make a statement about all possible learning algorithms. So we assume only that a learning-algorithm calculates a hypothesis that is consistent with all previous examples. Our result holds for this class of learning algorithms.
Definition 3.11 (Example Complexity)

Let $\delta > 0$ and $\epsilon > 0$ be given. The example complexity is the number $m$ of examples an arbitrary learning algorithm needs so that the created hypothesis $h$ is $\epsilon$ approximatively correct with the probability $1 - \delta$. 
Theorem 3.12 (Example Complexity)

The example complexity $m$ depends on $\epsilon$, $\delta$ and the hypotheses-space $\mathcal{H}$ as follows:

$$m \geq \frac{1}{\epsilon} \left( \ln \frac{1}{\delta} + \ln |\mathcal{H}| \right)$$
Question:
What does the last result mean?
Question:
The complexity depends on $\log(|\mathcal{H}|)$. What does this mean for boolean functions with $n$ arguments?

Answer:
We have $\log(|\mathcal{H}|) = 2^n$. Thus one needs exponentially-many examples even if one is satisfied with $\epsilon$ approximative correctness under a certain probability!
Proof (of the theorem):

$h_b \in \mathcal{H}$ with $\text{error}(h_b) > \epsilon$.

- What is the probability $P_b$ of $h_b$ being consistent with the chosen examples? For one single example: it is $\leq (1 - \epsilon)$ because of the definition of $\text{error}$. Therefore:

$$P_b \leq (1 - \epsilon)^m$$

- What is the probability $P'$ that there is a hypothesis $h_b$ with $\text{error}(h_b) > \epsilon$ and consistent with $m$ examples at all?

$$P' \leq |\mathcal{H}_{bad}|(1 - \epsilon)^m$$

$$\leq |\{h : \text{error}(h) > \epsilon\}|(1 - \epsilon)^m$$

$$\leq |\mathcal{H}|(1 - \epsilon)^m$$
Proof (continuation):

- We want $P' \leq \delta$:

$$|\mathcal{H}|(1 - \epsilon)^m \leq \delta$$

After some transformations:

$$\begin{align*}
(1 - \epsilon)^m &\leq \frac{\delta}{|\mathcal{H}|} \\
m \ln(1 - \epsilon) &\leq \ln(\delta) - \ln(|\mathcal{H}|) \\
m &\geq -\frac{1}{\ln(1 - \epsilon)}(\ln(\frac{1}{\delta}) + \ln(|\mathcal{H}|)) \\
m &\geq \frac{1}{\epsilon}(\ln(\frac{1}{\delta}) + \ln(|\mathcal{H}|))
\end{align*}$$

The last line holds because of $\ln(1 - \epsilon) < -\epsilon$. 

\[ \square \]
Essential result:

Learning is never better than looking up in a table!

1st way out: We ask for a more specialised hypothesis instead of one that is just consistent (complexity gets worse).

2nd way out: We give up on learning arbitrary boolean functions and concentrate on appropriate subclasses.
We consider Decision lists.

**Theorem 3.13 (Decision Lists can be learned)**

**Learning functions in** $k$-$DL(n)$ (decision lists with a maximum of $k$ tests) **has a PAC-complexity of**

$$m = \frac{1}{\epsilon} (\ln(\frac{1}{\delta}) + O(n^k \log_2(n^k))).$$

Each algorithm which returns a consistent decision list for a set of examples can be turned into a PAC-learning-algorithm, which learns a $k$-$DL(n)$ function after a maximum of $m$ examples.
Important estimates (exercises):

- $|\text{Conj}(n, k)| = \sum_{i=0}^{k} \binom{2n}{i} = \mathcal{O}(n^k)$,
- $|k-DL(n)| \leq 3|\text{Conj}(n,k)| |\text{Conj}(n, k)!|$
- $|k-DL(n)| \leq 2^{\mathcal{O}(n^k \log_2(n^k))}$. 
function DECISION-LIST-LEARNING(examples) returns a decision list, No or failure

if examples is empty then return the value No

t ← a test that matches a nonempty subset examples\(_t\) of examples such that the members of examples\(_t\) are all positive or all negative

if there is no such \( t \) then return failure

if the examples in examples\(_t\) are positive then \( o \leftarrow Yes \)

else \( o \leftarrow No \)

return a decision list with initial test \( t \) and outcome \( o \)

and remaining elements given by DECISION-LIST-LEARNING(examples − examples\(_t\))
3.5 PAC Learning

The graph illustrates the proportion of correct predictions on the test set as a function of the training set size for two different models: Decision tree and Decision list. The solid line represents the Decision tree, while the dashed line represents the Decision list. As the training set size increases, the proportion of correct predictions on the test set also increases for both models, with the Decision tree generally showing a higher proportion of correct predictions compared to the Decision list.
3.6 Noise and overfitting
Noise:

- examples are inconsistent ($Q(x)$ together with $\neg Q(x)$),
- no attributes left to classify more examples,
- makes sense if the environment is **stochastic**.
Overfitting: Dual to noise.

remaining examples can be classified using attributes which establish a pattern, which is not existent (irrelevant attributes).

Example 3.14 (Tossing Dice)

Several coloured dice are tossed. Every toss is described via (day, month, time, colour). As long as there is no inconsistency every toss is described by exactly one (totally overfitted) hypothesis.
Other examples:

- the pyramids,
- astrology,
- “Mein magisches Fahrrad”.
4. Learning in networks

- The human brain
- Neural networks
- The Perceptron
- Multi-layer feed-forward
Content of this chapter:

**Neural Nets:** Neural nets are an abstraction from entities operating in our brain.

**Perceptrons:** The perceptron is a particularly simple model. We describe the perceptron learning algorithm, which converges for each representable function: The linear separable functions.

**Feed Forward Nets:** We illustrate back propagation by simulating methods and techniques from perceptrons.
Learning with networks is a method to

- build **complex functions** from **many very simple but connected units** and to learn this construction from examples,
- improve the **understanding** of the functionality of **the human brain**.
4.1 The human brain
4 Learning in networks

4.1 The human brain

- Axon
- Cell body or Soma
- Nucleus
- Dendrite
- Synapse
- Axonal arborization
- Axon from another cell
- Synapses
A neuron consists of

- **the soma**: the body of the cell,
- **the nucleus**: the core of the cell,
- **the dendrites**, 
- **the axon**: 1 cm - 1 m in length.

The axon branches and connects to the dendrites of other neurons: these locations are called **synapses**. Each neuron shares synapses with 10-100000 others.
Signals are **propagated** from neuron to neuron by a complicated electrochemical reaction.

**Chemical transmitter substances** are released from the synapses and enter the dendrite, raising or lowering the electrical potential of the cell body.

When the **potential reaches a threshold** an electrical pulse or action potential is sent along the axon.

The pulse spreads out along the branches of the axons and releases transmitters into the bodies of other cells.
Question:
How does the building process of the network of neurons look like?

Answer:
Long term changes in the strength of the connections are in response to the pattern of stimulation.
Biology versus electronics:

<table>
<thead>
<tr>
<th></th>
<th>Computer</th>
<th>Human Brain</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computational units</td>
<td>1 CPU, $10^5$ gates</td>
<td>$10^{11}$ neurons</td>
</tr>
<tr>
<td></td>
<td>$10^9$ bits RAM, $10^{10}$ bits disk</td>
<td>$10^{11}$ neurons, $10^{14}$ synapses</td>
</tr>
<tr>
<td>Storage units</td>
<td>$10^{-8}$ sec</td>
<td>$10^{-3}$ sec</td>
</tr>
<tr>
<td>Cycle time</td>
<td>$10^9$ bits/sec</td>
<td>$10^{14}$ bits/sec</td>
</tr>
<tr>
<td>Bandwidth</td>
<td>$10^5$</td>
<td></td>
</tr>
<tr>
<td>Neuron updates/sec</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Computer:** sequential processes, very fast, “rebooting quite often”

**Brain:** works profoundly **concurrently**, quite slow, **error-correcting**, fault-tolerant (neurons die constantly)
4.2 Neural networks
Definition 4.1 (Neural Network)

A neural network consists of:

1. units,
2. links between units.

The links are weighted. There are three kinds of units:

1. input units,
2. hidden units,
3. output units.
A unit $i$ receives an input via links to other units $j$. The input function

\[ \text{in}_i := \sum_j W_{j,i} a_j \]

calculates the weighted sum.
<table>
<thead>
<tr>
<th>Notation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_i$</td>
<td>Activation value of unit $i$ (also the output of the unit)</td>
</tr>
<tr>
<td>$\mathbf{a}_i$</td>
<td>Vector of activation values for the inputs to unit $i$</td>
</tr>
<tr>
<td>$g$</td>
<td>Activation function</td>
</tr>
<tr>
<td>$g'$</td>
<td>Derivative of the activation function</td>
</tr>
<tr>
<td>$\text{Err}_i$</td>
<td>Error (difference between output and target) for unit $i$</td>
</tr>
<tr>
<td>$\text{Err}^e$</td>
<td>Error for example $e$</td>
</tr>
<tr>
<td>$I_i$</td>
<td>Activation of a unit $i$ in the input layer</td>
</tr>
<tr>
<td>$\mathbf{I}$</td>
<td>Vector of activations of all input units</td>
</tr>
<tr>
<td>$\mathbf{I}^e$</td>
<td>Vector of inputs for example $e$</td>
</tr>
<tr>
<td>$\text{in}_i$</td>
<td>Weighted sum of inputs to unit $i$</td>
</tr>
<tr>
<td>$N$</td>
<td>Total number of units in the network</td>
</tr>
<tr>
<td>$O$</td>
<td>Activation of the single output unit of a perceptron</td>
</tr>
<tr>
<td>$O_i$</td>
<td>Activation of a unit $i$ in the output layer</td>
</tr>
<tr>
<td>$\mathbf{O}$</td>
<td>Vector of activations of all units in the output layer</td>
</tr>
<tr>
<td>$t$</td>
<td>Threshold for a step function</td>
</tr>
<tr>
<td>$T$</td>
<td>Target (desired) output for a perceptron</td>
</tr>
<tr>
<td>$\mathbf{T}$</td>
<td>Target vector when there are several output units</td>
</tr>
<tr>
<td>$\mathbf{T}^e$</td>
<td>Target vector for example $e$</td>
</tr>
<tr>
<td>$W_{j,i}$</td>
<td>Weight on the link from unit $j$ to unit $i$</td>
</tr>
<tr>
<td>$W_i$</td>
<td>Weight from unit $i$ to the output in a perceptron</td>
</tr>
<tr>
<td>$\mathbf{W}_i$</td>
<td>Vector of weights leading into unit $i$</td>
</tr>
<tr>
<td>$\mathbf{W}$</td>
<td>Vector of all weights in the network</td>
</tr>
</tbody>
</table>
\[ a_i = g(in_i) \]
The **activation function** $g$ calculates the output $a_i$ (from the inputs) which will be transferred to other units via output-links:

$$a_i := g(in_i)$$

**Examples:**

(a) Step function

(b) Sign function

(c) Sigmoid function
Standardisation:

We consider $\text{step}_0$ instead of $\text{step}_t$. If an unit $i$ uses the activation function $\text{step}_t(x)$ then we bring in an additional input link “0” which adds a constant value of $a_0 := -1$. This value is weighted as $W_{0,i} := t$. Now we can use $\text{step}_0$ for the activation function:

\[
\text{step}_t\left(\sum_{j=1}^{n} W_{j,i}a_j\right) = \text{step}_0\left(-t + \sum_{j=1}^{n} W_{j,i}a_j\right) = \text{step}_0\left(\sum_{j=0}^{n} W_{j,i}a_j\right)
\]
Networks can be
- **recurrent**, i.e. somehow connected,
- **feed-forward**, i.e. they form an acyclic graph.

Usually networks are partitioned into **layers**:
- units in one layer have only links to units of the next layer.

E.g. **multi-layer feed-forward networks**: without internal states (no short-term memory).
Important:
The output of the input-units is determined by the environment.

Question 1:
Which function does the figure describe?

Question 2:
Why can non-trivial functions be represented at all?
Question 3:

How many units do we need?

- **a few**: a small number of functions can be represented,
- **many**: the network learns by heart (⇝ overfitting)
Hopfield networks:

1. bidirectional links with symmetrical weights,
2. activation function: \( \text{sign} \),
3. units are input- and output-units,
4. can store up to \( 0.14N \) training examples.
Learning with neural networks means the adjustment of the parameters to ensure consistency with the training-data.

Question:
How to find the optimal network structure?

Answer:
Perform a search in the space of network structures (e.g. with genetic algorithms).
4.3 The Perceptron
Definition 4.2 (Perceptron)

A **perceptron** is a feed-forward network with one layer based on the activation function $\text{step}_0$. 

---

**Perceptron Network**

- Input Units
- Output Units

**Single Perceptron**

- Input Units
- Output Unit
Question:
Can all boolean functions be represented by a feed-forward network?

Can \textit{AND}, \textit{OR} and \textit{NOT} be represented? Is it possible to represent every boolean function by simply combining these?

What about

$$f(x_1, \ldots, x_n) := \begin{cases} 1, & \text{if } \sum_{i=1}^{n} x_i > \frac{n}{2} \\ 0, & \text{else.} \end{cases}$$
Solution:

- Every boolean function can be composed using \( \text{AND} \), \( \text{OR} \) and \( \text{NOT} \) (or even only \( \text{NAND} \)).

- The **combination** of the respective perceptrons is **not a perceptron**!
Perceptron with sigmoid activation

Perceptron output
Question:
What about $XOR$?

- **(a)** $I_1$ and $I_2$
- **(b)** $I_1$ or $I_2$
- **(c)** $I_1$ xor $I_2$
Output $= \text{step}_0\left(\sum_{j=1}^{n} W_j I_j \right)$

$\sum_{j=1}^{n} W_j I_j = 0$ defines a $n$-dimensional hyperplane.

**Definition 4.3 (Linear Separable)**

A boolean function with $n$ attributes is called **linear separable** if there is a hyperplane ($(n - 1)$-dimensional subspace) which separates the positive domain-values from the negative ones.
4 Learning in networks

4.3 The Perceptron

(a) Separating plane

(b) Weights and threshold

\[ W = -1 \]
\[ t = -1.5 \]
\[ W = -1 \]
Learning algorithm:

Similar to **current best hypothesis** (⇝ chapter on learning).

- **hypothesis**: network with the current weights (firstly randomly generated)
- **UPDATE**: make it consistent through small changes.

Important: for each example **UPDATE is called several times**. These calls are called **epochs**.
function NEURAL-NETWORK-LEARNING(examples) returns network

network ← a network with randomly assigned weights
repeat
  for each e in examples do
    O ← NEURAL-NETWORK-OUTPUT(network, e)
    T ← the observed output values from e
    update the weights in network based on e, O, and T
  end
until all examples correctly predicted or stopping criterion is reached
return network
Definition 4.4 (Perceptron Learning (step_0))

Perceptron learning modifies the weights $W_j$ with respect to this rule:

$$W_j := W_j + \alpha \times I_j \times \text{Error}$$

with $\text{Error} := T - O$ (i.e. the difference between the correct and the current output-value). $\alpha$ is the learning rate.
Definition 4.5 (Perceptron Learning (sigmoid))

Perceptron learning modifies the weights $W_j$ with respect to this rule:

$$W_j := W_j + \alpha \times g'(in) I_j \times \text{Error}$$

with $\text{Error} := T - O$ (i.e. the difference between the correct and the current output-value). $\alpha$ is the learning rate.
The following algorithm uses $x_j[e]$ for $I_j$.

```plaintext
function PERCEPTRON-LEARNING(examples, network) returns a perceptron hypothesis
inputs: examples, a set of examples, each with input $x_1, \ldots, x_n$ and output $y$
        network, a perceptron with weights $W_j$, $j = 0 \ldots n$, and activation function $g$
repeat
    for each $e$ in examples do
        $in \leftarrow \sum_{j=0}^{n} W_j \cdot x_j[e]$
        $Err \leftarrow y[e] - g(in)$
        $W_j \leftarrow W_j + \alpha \times Err \times g'(in) \times x_j[e]$
until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS(network)
```
Theorem 4.6 (Rosenblatt’s Theorem)

Every function which can be represented by a perceptron is learned through the perceptron learning algorithm (Definition 4.4).

More exactly: The series $W_j$ converges to a function which represents the examples correctly.
Proof:

Let $\hat{\omega}$ be a solution, wlog. we can assume (why? $\rightsquigarrow$ exercise)

$$\hat{\omega} \overrightarrow{I} > 0 \text{ for all } \overrightarrow{I} \in I_{\text{pos}} \cup -I_{\text{neg}}$$

with $I_{\text{pos}}$ consisting of the positive and $I_{\text{neg}}$ consisting of the negative examples (and $-I_{\text{neg}} = \{-\overrightarrow{I} : \overrightarrow{I} \in I_{\text{neg}}\}$).

Let $I' := I_{\text{pos}} \cup -I_{\text{neg}}$ and $m := \min \{\hat{\omega} \overrightarrow{I} : \overrightarrow{I} \in I'\}$.

Let $w_1, \ldots, w_j, \ldots$ be the sequence of weights resulting from the algorithm.

We want to show that this sequence eventually becomes constant.
Proof (continued):

Consider the possibility that not all $\vec{w}_i$ are different from their predecessor (or successor) (this is the case if the new example is not consistent with the current weights). Let $k_1, k_2, \ldots, k_j$ be the indices of the changed weights (where error is non-zero), i.e.

$$\vec{w}_{k_j} \vec{I}_{k_j} \leq 0, \quad \vec{w}_{k_j+1} = \vec{w}_{k_j} + \alpha \vec{I}_{k_j}.$$ 

With $\vec{I}_{k_j}$ being the $k_j$-th tested example in $I'$ (which is not consistent (wrt the definition of $k_j$)).

The we have

$$\vec{w}_{k_j+1} = \vec{w}_{k_1} + \alpha \vec{I}_{k_1} + \alpha \vec{I}_{k_2} + \ldots + \alpha \vec{I}_{k_j}$$
Proof (continued):

We use this to show that $j$ cannot become arbitrarily big. We compose

$$\cos \omega = \frac{\hat{w}w_{k_{j+1}}}{\|\hat{w}\| \|w_{k_{j+1}}\|}$$

and estimate as follows (by decreasing the numerator and increasing the denominator):

$$\cos \omega = \frac{\hat{w}w_{k_{j+1}}}{\|\hat{w}\| \|w_{k_{j+1}}\|} \geq \frac{\hat{w}w_{k_{1}} + \alpha mj}{\|\hat{w}\| \sqrt{\|w_{k_{1}}\|^2 + \alpha^2 Mj}}$$

The right side converges to infinity (when $j$ increases to infinity) and cosinus will never get greater than 1. This leads to the contradiction.
Proof (continued):

How do we get the estimation above?

- the scalar product $\hat{w}w_{k_j+1}$ leads to
  $$\hat{w}w_{k_1} + \alpha \hat{w} (I_1 + \ldots + I_j) \geq \hat{w}w_{k_1} + \alpha mj.$$ 

- $\|w_{k_j+1}\|^2 = \|w_{k_j} + \alpha I_{k_j}\|^2 = \|w_{k_j}\|^2 + 2\alpha I_{k_j}w_{k_j} + \alpha^2 \|I_{k_j}\|^2 \leq \|w_{k_j}\|^2 + \alpha^2 \|I_{k_j}\|^2$, da $I_{k_j}w_{k_j} \leq 0$, this is how we have chosen $k_j$.

Now be $M := \max \{\|I\|^2 : I \in I'\}$. Then
  $$\|w_{k_j+1}\|^2 \leq \|w_{k_1}\|^2 + \alpha^2 \|I_{k_1}\|^2 + \ldots + \alpha^2 \|I_{k_j}\|^2 \leq \|w_{k_1}\|^2 + \alpha^2 Mj$$ holds.
What is the underlying example?

Figure 4.12: Perceptron Better than Decision Tree
What is the underlying example?

Figure 4.13: Decision Tree Better than Perceptron
4.4 Multi-layer feed-forward
Problem:
How does the error-function of the hidden units look like?

Learning with multi-layer networks is called back propagation. Hidden units can be seen as perceptrons (Figure on page 320). The outcome can be a linear combination of such perceptrons (see next two slides).
$h_w(x_1, x_2)$
$h_w(x_1, x_2)$

$0.1 \ 0.2 \ 0.3 \ 0.4 \ 0.5 \ 0.6 \ 0.7 \ 0.8 \ 0.9 \ 1$

$x_1$ $x_2$

-4 -2 0 2 4

-4 -2 0 2 4
4 Learning in networks

4.4 Multi-layer feed-forward

Output units $O_i$

Hidden units $a_j$

Input units $I_k$

$W_{j,i}$

$W_{k,j}$
The perceptron was not powerful enough in our restaurant-example (Figure 20). So we try 2 layers. 10 attributes lead to 10 input-units.

**Question**

How many hidden units are necessary?

**Answer:**

Four!

Perceptron’s error is easily determined because there was only one \( W_j \) between input and output. Now we have several.

How should the error be distributed?
Minimising by using derivatives (1)

We minimise \( E = \frac{1}{2} \sum_i (T_i - O_i)^2 \) and get

\[
E = \frac{1}{2} \sum_i (T_i - g(\sum_j W_{j,i}a_j))^2 \\
= \frac{1}{2} \sum_i (T_i - g(\sum_j W_{j,i}g(\sum_k W_{k,j}I_k))))^2
\]

Do a gradient descent.

\[
\frac{\partial E}{\partial W_{j,i}} = \frac{1}{2} \sum_i (T_i - O_i)^2 \\
= \frac{1}{2} 2(T_i - g(\ldots))(-\frac{\partial g}{\partial x}(in_i)\big)a_j \\
= -a_j(T_i - O_i)\frac{\partial g}{\partial x}(in_i) \\
= -a_j \Delta_i
\]
Minimising by using derivatives (2)

Now the $W_{k,j}$:

$$
\frac{\partial E}{\partial W_{k,j}} = \frac{1}{2} \sum_i \left( 2(T_i - g(\ldots))(-\frac{\partial g}{\partial x}(in_i))(W_{j,i}\frac{\partial g}{\partial x}(in_j)I_k) \right) \\
= \sum_i \left( \Delta_i W_{j,i}\frac{\partial g}{\partial x}(in_j)I_k \right) \\
= \frac{\partial g}{\partial x}(in_j)I_k \sum_i W_{j,i} \Delta_i
$$
Idea:

We perform **two** different updates. One for the weights to the input units and one for the weights to the output units.

**output units:** similar to the perceptron

\[ W_{j,i} := W_{j,i} + \alpha \times a_j \times Error_i \times g'(in_i) \]

Instead of \( Error_i \times g'(in_i) \) write \( \Delta_i \).

**hidden units:** each hidden unit \( j \) is partly responsible for the error \( \Delta_i \) (if \( j \) is connected with the output unit \( i \)).

\[ W_{k,j} := W_{k,j} + \alpha \times I_k \times \Delta_j \]

with \( \Delta_j := g'(in_j) \sum_i W_{j,i} \Delta_i \).
function BACK-PROP-LEARNING (examples, network) returns a neural network

inputs: examples, a set of examples, each with input vector $x$ and output vector $y$

network, a multilayer network with $L$ layers, weights $W_{j,i}$, activation function $g$

repeat
  for each $e$ in examples do
    for each node $j$ in the input layer do $a_j \leftarrow x_j[e]$
    for $\ell = 2$ to $M$ do
      $in_i \leftarrow \sum_j W_{j,i} a_j$
      $a_i \leftarrow g(in_i)$
    for each node $i$ in the output layer do
      $\Delta_i \leftarrow g'(in_i) \times (y_i[e] - a_i)$
    for $\ell = M - 1$ to 1 do
      for each node $j$ in layer $\ell$ do
        $\Delta_j \leftarrow g'(in_j) \sum_i W_{j,i} \Delta_i$
        for each node $i$ in layer $\ell + 1$ do
          $W_{j,i} \leftarrow W_{j,i} + \alpha \times a_j \times \Delta_i$
  until some stopping criterion is satisfied
return NEURAL-NET-HYPOTHESIS (network)
4 Learning in networks

4.4 Multi-layer feed-forward

![Graph showing the relationship between total error on training set and number of epochs. The error decreases significantly as the number of epochs increases.](image-url)
4 Learning in networks

4.4 Multi-layer feed-forward

The graph shows the proportion correct on the test set as a function of the training set size for decision trees and multilayer networks.

- **Decision tree** (solid line)
- **Multilayer network** (dashed line)

The x-axis represents the training set size, while the y-axis shows the proportion correct on the test set.
Back propagation algorithm:

1. calculate $\Delta_i$ for the output units based on the observed error $error_i$.
2. for each layer proceed recursively (output layer first):
   - back propagate the $\Delta_i$ (predecessor layer)
   - modify the weight between the current layers

Important:

Back propagation is gradient search!
error is the function of the network’s weights. This function delivers the error surface.
General remarks:

**expressibility:** neural networks are suitable for continuous input and outputs (noise). To represent all boolean functions with $n$ attributes, $\frac{2^n}{n}$ hidden units suffice.

Often much less suffice: the art of determining the topology of the network.
efficiency: \( m \) examples, \( |W| \) weights: each epoch needs \( \mathcal{O}(m \times |W|) \)-time. We know: 
**Number of epochs is exponential.**
In practice the time of convergence is very variable.

**Problem:** local minima on the error surface.

transparency: black box. Trees and lists explain their results!
5. Knowledge Engineering (1)

- Sentential Logic
- Sudoku
- Calculi for SL
- Wumpus in SL
- A Puzzle
Content of this chapter (1):

**Logic:** We introduce *sentential* logic (also called *propositional* logic). This logic dates back to Boole and is the basis for many logical frameworks. The essential features of most logics can be illustrated in a puristic way.

We are using logics to *describe* the world and how the world behaves.

**Sudoku:** We illustrate how to use SL with the game of *Sudoku*. The conditions of being a solution can be easily stated in SL.
Content of this chapter (2):

**Calculi for SL:** While it is nice to describe the world, or the solution of a Sudoku puzzle, we also want to draw conclusions about it or even to solve the puzzle. Therefore we have to derive new information and deduce statements, that are not explicitly given.

**Examples:** We illustrate the use of SL with two more examples: The *Wumpus world* and one of the weekly puzzles in the newspaper *Die Zeit.*
5.1 Sentential Logic
Definition 5.1 (Sentential Logic $\mathcal{L}_{SL}$, Lang. $\mathcal{L} \subseteq \mathcal{L}_{SL}$)

The language $\mathcal{L}_{SL}$ of propositional (or sentential) logic consists of

- $\Box$ and $\top$: the constants *falsum* and *verum*,
- $p, q, r, x_1, x_2, \ldots x_n, \ldots$: a countable set $\mathcal{A}_T$ of SL-constants,
- $\neg, \land, \lor, \rightarrow$: the sentential connectives ($\neg$ is unary, all others are binary operators),
- $(, )$: the parentheses to help readability.

In most cases we consider only a finite set of SL-constants. They define a language $\mathcal{L} \subseteq \mathcal{L}_{SL}$. The set of $\mathcal{L}$-formulae $Fml_\mathcal{L}$ is defined inductively.
Definition 5.2 (Semantics, Valuation, Model)

A valuation $v$ for a language $\mathcal{L} \subseteq \mathcal{L}_{SL}$ is a mapping from the set of SL-constants defined by $\mathcal{L}$ into the set $\{\text{true}, \square\}$ with $v(\square) = \text{false}$, $v(\top) = \text{true}$. Each valuation $v$ can be uniquely extended to a function $\bar{v} : Fml_\mathcal{L} \rightarrow \{\text{true}, \square\}$ so that:

- $\bar{v}(\neg p) = \begin{cases} \text{true, if } \bar{v}(p) = \text{false,} \\ \text{false, if } \bar{v}(p) = \text{true.} \end{cases}$

- $\bar{v}(\varphi \land \gamma) = \begin{cases} \text{true, if } \bar{v}(\varphi) = \top \text{ and } \bar{v}(\gamma) = \text{true,} \\ \text{false, else} \end{cases}$

- $\bar{v}(\varphi \lor \gamma) = \begin{cases} \text{true, if } \bar{v}(\varphi) = \top \text{ or } \bar{v}(\gamma) = \text{true,} \\ \text{false, else} \end{cases}$
Definition (continued)

\[ \bar{v}(\varphi \rightarrow \gamma) = \begin{cases} 
\text{true, if } \bar{v}(\varphi) = \text{false or } (\bar{v}(\varphi) = \text{true and } \bar{v}(\gamma) = \text{true}), \\
\text{false, else} 
\end{cases} \]

Thus each valuation \( v \) uniquely defines a \( \bar{v} \). We call \( \bar{v} \) an \( \mathcal{L} \)-structure.

A structure determines for each formula if it is true or false. If a formula \( \phi \) is true in structure \( \bar{v} \) we also say \( A_v \) is a model of \( \phi \). From now on we will speak of models, structures and valuations synonymously.

Semantics

The process of mapping a set of \( \mathcal{L} \)-formulae into \{true, false\} is called semantics.
Definition 5.3 (Model, Theory, Tautology (Valid))

1. A formula \( \varphi \in \text{Fml}_L \) holds under the valuation \( \nu \) if \( \bar{\nu}(\varphi) = \text{true} \). We also write \( \bar{\nu} \models \varphi \) or simply \( \nu \models \varphi \). \( \bar{\nu} \) is a model of \( \varphi \).

2. A theory is a set of formulae: \( T \subseteq \text{Fml}_L \). \( \nu \) satisfies \( T \) if \( \bar{\nu}(\varphi) = \text{true} \) for all \( \varphi \in T \). We write \( \nu \models T \).

3. A \( L \)-formula \( \varphi \) is called \( L \)-tautology (or simply called valid) if for all possible valuations \( \nu \) in \( L \) \( \nu \models \varphi \) holds.

From now on we suppress the language \( L \) when obvious from context.
Definition 5.4 (Consequence Set $Cn(T)$)

A formula $\varphi$ follows from $T$ if for all models $v$ of $T$ (i.e. $v \models T$) also $v \models \varphi$ holds. We write: $T \models \varphi$. We call

$$Cn_L(T) = \text{def} \{ \varphi \in Fml_L : T \models \varphi \},$$

or simply $Cn(T)$, the semantic consequence operator.
Lemma 5.5 (Properties of $C_n(T)$)

The semantic consequence operator has the following properties:

1. **$T$-expansion:** $T \subseteq C_n(T)$,
2. **Monotony:** $T \subseteq T' \Rightarrow C_n(T) \subseteq C_n(T')$,
3. **Closure:** $C_n(C_n(T)) = C_n(T)$.

Lemma 5.6 ($\varphi \notin C_n(T)$)

$\varphi \notin C_n(T)$ if and only if there is a model $v$ with $v \models T$ and $\bar{v}(\varphi) = \text{false}$.
5 Knowledge Engineering (1)
5.1 Sentential Logic

**Definition 5.7 (MOD(T), Cn(U))**

If $T \subseteq Fml_\mathcal{L}$ then we denote with $\text{MOD}(T)$ the set of all $\mathcal{L}$-structures $\mathcal{A}$ which are models of $T$:

$$\text{MOD}(T) = \text{def} \ \{ \mathcal{A} : \mathcal{A} \models T \}.$$  

If $\mathcal{U}$ is a set of models, we consider all those sentences, which are valid in all models of $\mathcal{U}$. We call this set $Cn(\mathcal{U})$:

$$Cn(\mathcal{U}) = \text{def} \ \{ \varphi \in Fml_\mathcal{L} : \forall v \in \mathcal{U} : \bar{v}(\varphi) = \text{true} \}.$$  

$\text{MOD}$ is obviously dual to $Cn$:

$$Cn(\text{MOD}(T)) = Cn(T), \quad \text{MOD}(Cn(T)) = \text{MOD}(T).$$
Definition 5.8 (Completeness of a Theory $T$)

$T$ is called **complete** if for each formula $\varphi \in Fml$: $T \models \varphi$ or $T \models \neg \varphi$ holds.

**Attention:**

Do not mix up this last condition with the property of a valuation (model) $v$: each model is complete in the above sense.
Definition 5.9 (Consistency of a Theory)

$T$ is called **consistent** if there is a valuation (model) $\nu$ with $\nu(\varphi) = \text{true}$ for all $\varphi \in T$.

Lemma 5.10 (Ex Falso Quodlibet)

$T$ is consistent if and only if $\text{Cn}(T) \neq Fml_\mathcal{L}$. 
5.2 Sudoku
Since some time, **Sudoku** puzzles are becoming quite famous.

Table 5.42: A simple Sudoku ($S_1$)
Can they be solved with sentential logic?

**Idea:** Given a Sudoku-Puzzle $S$, construct a language $\mathcal{L}_{\text{Sudoku}}$ and a theory $T_S \subseteq \text{Fml}_{\mathcal{L}_{\text{Sudoku}}}$ such that

$$\text{MOD}(T_S) = \text{Solutions of the puzzle } S$$

**Solution**

In fact, we construct a theory $T_{\text{Sudoku}}$ and for each (partial) instance of a $9 \times 9$ puzzle $S$ a particular theory $T_S$ such that

$$\text{MOD}(T_{\text{Sudoku}} \cup T_S) = \{S : S \text{ is a solution of } S\}$$
We introduce the following language $\mathcal{L}_{\text{Sudoku}}$:

1. $\text{eins}_{i,j}$, $1 \leq i, j \leq 9$,
2. $\text{zwei}_{i,j}$, $1 \leq i, j \leq 9$,
3. $\text{drei}_{i,j}$, $1 \leq i, j \leq 9$,
4. $\text{vier}_{i,j}$, $1 \leq i, j \leq 9$,
5. $\text{fuenf}_{i,j}$, $1 \leq i, j \leq 9$,
6. $\text{sechs}_{i,j}$, $1 \leq i, j \leq 9$,
7. $\text{sieben}_{i,j}$, $1 \leq i, j \leq 9$,
8. $\text{acht}_{i,j}$, $1 \leq i, j \leq 9$,
9. $\text{neun}_{i,j}$, $1 \leq i, j \leq 9$.

This completes the language, the syntax.

How many symbols are these?
We distinguished between the puzzle $S$ and a solution $S$ of it.

What is a model (or valuation) in the sense of Definition 5.2?

Table 5.43: How to construct a model $S$?
We have to give our symbols a meaning: the semantics!

\( \text{eins}_{i,j} \) means \( i, j \) contains a 1
\( \text{zwei}_{i,j} \) means \( i, j \) contains a 2
\( \vdots \)
\( \text{neun}_{i,j} \) means \( i, j \) contains a 9

To be precise: given a \( 9 \times 9 \) square that is completely filled out, we define our valuation \( v \) as follows (for all \( 1 \leq i, j \leq 9 \)).

\[
v(\text{eins}_{i,j}) = \begin{cases} 
  \text{true}, & \text{if 1 is at position } (i, j), \\
  \text{false}, & \text{else}.
\end{cases}
\]
\[ v(\text{zwei}_{i,j}) = \begin{cases} 
\text{true, if } 2 \text{ is at position } (i, j), \\
\text{false, else}.
\end{cases} \]

\[ v(\text{drei}_{i,j}) = \begin{cases} 
\text{true, if } 3 \text{ is at position } (i, j), \\
\text{false, else}.
\end{cases} \]

\[ v(\text{vier}_{i,j}) = \begin{cases} 
\text{true, if } 4 \text{ is at position } (i, j), \\
\text{false, else}.
\end{cases} \]

etc.

\[ v(\text{neun}_{i,j}) = \begin{cases} 
\text{true, if } 9 \text{ is at position } (i, j), \\
\text{false, else}.
\end{cases} \]

Therefore any \(9 \times 9\) square can be seen as a model or valuation with respect to the language \(L_{\text{Sudoku}}\).
How does $T_S$ look like?

$$T_S = \{ \text{eins}_{1,4}, \text{eins}_{5,8}, \text{eins}_{6,6}, \text{zwei}_{2,2}, \text{zwei}_{4,8}, \text{drei}_{6,8}, \text{drei}_{8,3}, \text{drei}_{9,4}, \text{vier}_{1,7}, \text{vier}_{2,5}, \text{vier}_{3,1}, \text{vier}_{4,3}, \text{vier}_{8,2}, \text{vier}_{9,8}, \vdots, \text{neun}_{3,4}, \text{neun}_{5,2}, \text{neun}_{6,9}, \}$$
How should the theory $T_{\text{Sudoku}}$ look like (s.t. models of $T_{\text{Sudoku}} \cup T_S$ correspond to solutions of the puzzle)?

**First square: $T_1$**

1. $\text{eins}_{1,1} \lor \ldots \lor \text{eins}_{3,3}$
2. $\text{zwei}_{1,1} \lor \ldots \lor \text{zwei}_{3,3}$
3. $\text{drei}_{1,1} \lor \ldots \lor \text{drei}_{3,3}$
4. $\text{vier}_{1,1} \lor \ldots \lor \text{vier}_{3,3}$
5. $\text{fuenf}_{1,1} \lor \ldots \lor \text{fuenf}_{3,3}$
6. $\text{sechs}_{1,1} \lor \ldots \lor \text{sechs}_{3,3}$
7. $\text{sieben}_{1,1} \lor \ldots \lor \text{sieben}_{3,3}$
8. $\text{acht}_{1,1} \lor \ldots \lor \text{acht}_{3,3}$
9. $\text{neun}_{1,1} \lor \ldots \lor \text{neun}_{3,3}$
The formulae on the last slide are saying, that

1. The number 1 must appear somewhere in the first square.
2. The number 2 must appear somewhere in the first square.
3. The number 3 must appear somewhere in the first square.
4. etc

Does that mean, that each number $1, \ldots, 9$ occurs exactly once in the first square?
No! We have to say, that each number occurs only once:

$T_1'$:

1. $\neg(\text{eins}_{i,j} \land \text{zwei}_{i,j}), 1 \leq i, j \leq 3,$
2. $\neg(\text{eins}_{i,j} \land \text{drei}_{i,j}), 1 \leq i, j \leq 3,$
3. $\neg(\text{eins}_{i,j} \land \text{vier}_{i,j}), 1 \leq i, j \leq 3,$
4. etc
5. $\neg(\text{zwei}_{i,j} \land \text{drei}_{i,j}), 1 \leq i, j \leq 3,$
6. $\neg(\text{zwei}_{i,j} \land \text{vier}_{i,j}), 1 \leq i, j \leq 3,$
7. $\neg(\text{zwei}_{i,j} \land \text{fuenf}_{i,j}), 1 \leq i, j \leq 3,$
8. etc

How many formulae are these?
Second square: \( T_2 \)

|   | eins\(_{1,4} \lor \ldots \lor eins\(_{3,6}  
|---|--------------------------------------------------|
| 2 | zwei\(_{1,4} \lor \ldots \lor zwei\(_{3,6}  
| 3 | drei\(_{1,4} \lor \ldots \lor drei\(_{3,6}  
| 4 | vier\(_{1,4} \lor \ldots \lor vier\(_{3,6}  
| 5 | fuenf\(_{1,4} \lor \ldots \lor fuenf\(_{3,6}  
| 6 | sechs\(_{1,4} \lor \ldots \lor sechs\(_{3,6}  
| 7 | sieben\(_{1,4} \lor \ldots \lor sieben\(_{3,6}  
| 8 | acht\(_{1,4} \lor \ldots \lor acht\(_{3,6}  
| 9 | neun\(_{1,4} \lor \ldots \lor neun\(_{3,6}  

And all the other formulae from the previous slides (adapted to this case): \( T'_2 \)
The same has to be done for all 9 squares.

What is still missing:

**Rows:** Each row should contain exactly the numbers from 1 to 9 (no number twice).

**Columns:** Each column should contain exactly the numbers from 1 to 9 (no number twice).
First Row: $T_{\text{Row 1}}$

1. $\text{eins}_{1,1} \lor \text{eins}_{1,2} \lor \ldots \lor \text{eins}_{1,9}$
2. $\text{zwei}_{1,1} \lor \text{zwei}_{1,2} \lor \ldots \lor \text{zwei}_{1,9}$
3. $\text{drei}_{1,1} \lor \text{drei}_{1,2} \lor \ldots \lor \text{drei}_{1,9}$
4. $\text{vier}_{1,1} \lor \text{vier}_{1,2} \lor \ldots \lor \text{vier}_{1,9}$
5. $\text{fuenf}_{1,1} \lor \text{fuenf}_{1,2} \lor \ldots \lor \text{fuenf}_{1,9}$
6. $\text{sechs}_{1,1} \lor \text{sechs}_{1,2} \lor \ldots \lor \text{sechs}_{1,9}$
7. $\text{sieben}_{1,1} \lor \text{sieben}_{1,2} \lor \ldots \lor \text{sieben}_{1,9}$
8. $\text{acht}_{1,1} \lor \text{acht}_{1,2} \lor \ldots \lor \text{acht}_{1,9}$
9. $\text{neun}_{1,1} \lor \text{neun}_{1,2} \lor \ldots \lor \text{neun}_{1,9}$
Analogously for all other rows, eg.

**Ninth Row:** $T_{\text{Row 9}}$

1. $\text{eins}_{9,1} \lor \text{eins}_{9,2} \lor \ldots \lor \text{eins}_{9,9}$
2. $\text{zwei}_{9,1} \lor \text{zwei}_{9,2} \lor \ldots \lor \text{zwei}_{9,9}$
3. $\text{drei}_{9,1} \lor \text{drei}_{9,2} \lor \ldots \lor \text{drei}_{9,9}$
4. $\text{vier}_{9,1} \lor \text{vier}_{9,2} \lor \ldots \lor \text{vier}_{9,9}$
5. $\text{fuenf}_{9,1} \lor \text{fuenf}_{9,2} \lor \ldots \lor \text{fuenf}_{9,9}$
6. $\text{sechs}_{9,1} \lor \text{sechs}_{9,2} \lor \ldots \lor \text{sechs}_{9,9}$
7. $\text{sieben}_{9,1} \lor \text{sieben}_{9,2} \lor \ldots \lor \text{sieben}_{9,9}$
8. $\text{acht}_{9,1} \lor \text{acht}_{9,2} \lor \ldots \lor \text{acht}_{9,9}$
9. $\text{neun}_{9,1} \lor \text{neun}_{9,2} \lor \ldots \lor \text{neun}_{9,9}$

*Is that sufficient? What if a row contains several 1’s?*
First Column: $T_{\text{Column 1}}$

1. $\text{eins}_{1,1} \lor \text{eins}_{2,1} \lor \ldots \lor \text{eins}_{9,1}$
2. $\text{zwei}_{1,1} \lor \text{zwei}_{2,1} \lor \ldots \lor \text{zwei}_{9,1}$
3. $\text{drei}_{1,1} \lor \text{drei}_{2,1} \lor \ldots \lor \text{drei}_{9,1}$
4. $\text{vier}_{1,1} \lor \text{vier}_{2,1} \lor \ldots \lor \text{vier}_{9,1}$
5. $\text{fuenf}_{1,1} \lor \text{fuenf}_{2,1} \lor \ldots \lor \text{fuenf}_{9,1}$
6. $\text{sechs}_{1,1} \lor \text{sechs}_{2,1} \lor \ldots \lor \text{sechs}_{9,1}$
7. $\text{sieben}_{1,1} \lor \text{sieben}_{2,1} \lor \ldots \lor \text{sieben}_{9,1}$
8. $\text{acht}_{1,1} \lor \text{acht}_{2,1} \lor \ldots \lor \text{acht}_{9,1}$
9. $\text{neun}_{1,1} \lor \text{neun}_{2,1} \lor \ldots \lor \text{neun}_{9,1}$

Analogously for all other columns.

Is that sufficient? What if a column contains several 1’s?
All put together:

\[
T_{\text{Sudoku}} = T_1 \cup T'_1 \cup \ldots \cup T_9 \cup T'_9 \\
T_{\text{Row } 1} \cup \ldots \cup T_{\text{Row } 9} \\
T_{\text{Column } 1} \cup \ldots \cup T_{\text{Column } 9}
\]
Here is a more difficult one.

Table 5.44: A difficult Sudoku $S_{\text{difficult}}$
The above formulation is strictly formulated in propositional logic.

Theorem provers, even if they consider only propositional theories, often use predicates, variables etc.

`smodels` uses a predicate logic formulation, including variables. But as there are no function symbols, such an input can be seen as a compact representation.

It allows to use a few rules as a shorthand for thousands of rules using propositional constants.
For example *smodels* uses the following constructs:

1. row(0..8) is a shorthand for row(0), row(1), ..., row(8).
2. val(1..9) is a shorthand for val(1), val(2), ..., val(9).
3. The constants 1, ..., 9 will be treated as numbers (so there are operations available to add, subtract or divide them).
Statements in *smodels* are written as

\[
\text{head} :- \text{body}
\]

This corresponds to an implication

\[
\text{body} \rightarrow \text{head}.
\]

An important feature in smodels is that *all* atoms that do not occur in any head, are automatically false.

For example, the theory

\[
p(X, Y, 5) :- \text{row}(X), \text{col}(Y)
\]

means that the whole grid is filled with 5’s *and only with 5’s*: eg. \(\neg p(X, Y, 1)\) is true for all \(X, Y\), as well as \(\neg p(X, Y, 2)\) etc.
More constructs in \textit{smodels}

1 \begin{verbatim}
1 \{ p(X,Y,A) : \text{val}(A) \} 1
   :- row(X), col(Y)
\end{verbatim}

this makes sure that in all entries of the grid, exactly one number (\text{val}()) is contained.

2 \begin{verbatim}
1 \{ p(X,Y,A) : row(X) : col(Y)
   : eq(div(X,3), div(R,3))
   : eq(div(Y,3), div(C,3) \} 1
   :- val(A), row(R), col(C)
\end{verbatim}

this rule ensures that in each of the 9 squares each number from 1 to 9 occurs only once.

3 More detailed info on the web-page.
5.3 Calculi for SL
A general notion of a certain sort of calculi.

**Definition 5.11 (Hilbert-Type Calculi)**

A *Hilbert-Type calculus over a language* $\mathcal{L}$ *is a pair* $\langle \text{Ax}, \text{Inf} \rangle$ *where*

- **Ax:** is a subset of $Fml_{\mathcal{L}}$, the set of well-formed formulae in $\mathcal{L}$: they are called **axioms**,
- **Inf:** is a set of pairs written in the form

$$\phi_1, \phi_2, \ldots, \phi_n \quad \psi$$

where $\phi_1, \phi_2, \ldots, \phi_n, \psi$ are $\mathcal{L}$-formulae: they are called **inference rules**.

Intuitively, one can assume all axioms as “true formulae” (*tautologies*) and then use the inference rules to derive even more new formulae.
Definition 5.12 (Calculus for Sentential Logic SL)

We define $\text{Hilbert}^{SL}_L = \langle \text{Ax}^{SL}_L, \{MP\} \rangle$, the Hilbert-Type calculus: $L \subseteq L_{SL}$ with the wellformed formulae $Fml_L$ as defined in Definition 5.1.

Axioms in SL ($\text{Ax}^{SL}_L$) are the following formulae:

1. $\phi \rightarrow \top, \Box \rightarrow \phi, \neg \top \rightarrow \Box, \Box \rightarrow \neg \top$,
2. $(\phi \rightarrow \psi) \rightarrow ((\phi \rightarrow (\psi \rightarrow \chi)) \rightarrow (\phi \rightarrow \chi))$,
3. $(\phi \land \psi) \rightarrow \phi, (\phi \land \psi) \rightarrow \psi$,
4. $\phi \rightarrow (\phi \lor \psi), \psi \rightarrow (\phi \lor \psi)$,
5. $\neg \neg \phi \rightarrow \phi, (\phi \rightarrow \psi) \rightarrow ((\phi \rightarrow \neg \psi) \rightarrow \neg \phi)$,
6. $\phi \rightarrow (\psi \rightarrow \phi), \phi \rightarrow (\psi \rightarrow (\phi \land \psi))$.
7. $(\phi \rightarrow \chi) \rightarrow ((\psi \rightarrow \chi) \rightarrow (\phi \lor \psi \rightarrow \chi))$.

$\phi, \psi, \chi$ stand for arbitrarily complex formulae (not just constants).

They represent schemata, rather than formulae in the language.
Definition (continued)

The only inference rule in SL is **modus ponens**:

\[ MP : \text{Fml} \times \text{Fml} \rightarrow \text{Fml} : (\varphi, \varphi \rightarrow \psi) \rightarrow \psi. \]

or short

\[ (\text{MP}) \quad \frac{\varphi, \varphi \rightarrow \psi}{\psi}. \]

(\varphi, \psi \text{ are arbitrarily complex formulae}).
Definition 5.13 (Proof)

A proof of a formula \( \varphi \) from a theory \( T \subseteq Fml_L \) is a sequence \( \varphi_1, \ldots, \varphi_n \) of formulae such that \( \varphi_n = \varphi \) and for all \( i \) with \( 1 \leq i \leq n \) one of the following conditions holds:

- \( \varphi_i \) is substitution instance of an axiom,
- \( \varphi_i \in T \),
- there is \( \varphi_l, \varphi_k = (\varphi_l \rightarrow \varphi_i) \) with \( l, k < i \). Then \( \varphi_i \) is the result of the application of modus ponens on the predecessor-formulae of \( \varphi_i \).

We write: \( T \vdash \varphi \) (\( \varphi \) can be derived from \( T \)).
We show that:

1. \( \because \top \) and \( \because \neg \Box \).
2. \( A \because A \lor B \) and \( \because A \lor \neg A \).
3. The rule
   \[
   \frac{A \rightarrow \varphi, \neg A \rightarrow \psi}{\varphi \lor \psi}
   \]
   can be derived.
4. Our version of sentential logic does not contain a connective “\( \leftrightarrow \)”. We define “\( \phi \leftrightarrow \psi \)” as a macro for “\( \phi \rightarrow \psi \land \psi \rightarrow \phi \)”. Show the following:

   If \( \because \phi \leftrightarrow \psi \), then \( \because \phi \) if and only if \( \because \psi \).
We have now introduced two important notions:

**Syntactic derivability** \( \vdash \): the notion that certain formulae can be derived from other formulae using a certain calculus,

**Semantic validity** \( \models \): the notion that certain formulae follow from other formulae based on the semantic notion of a model.
Definition 5.14 (Correct-, Completeness for a calculus)

Given an arbitrary calculus (which defines a notion $\vdash$) and a semantics based on certain models (which defines a relation $|=\), we say that

**Correctness:** The calculus is **correct** with respect to the semantics, if the following holds:

$$\Phi \vdash \phi \text{ implies } \Phi \models \phi.$$  

**Completeness:** The calculus is **complete** with respect to the semantics, if the following holds:

$$\Phi \models \phi \text{ implies } \Phi \vdash \phi.$$
Theorem 5.15 (Correct-, Completeness for \( \text{Hilbert}_{SL}^L \))

A formula follows semantically from a theory \( T \) if and only if it can be derived:

\[
T \models \varphi \text{ if and only if } T \vdash \varphi
\]
Theorem 5.16 (Compactness for Hilbert$_{SL}^L$)

A formula follows from a theory $T$ if and only if it follows from a finite subset of $T$:

$$Cn(T) = \bigcup \{Cn(T') : T' \subseteq T, T' \text{finite}\}.$$ 

Although the axioms from above and modus ponens suffice it is reasonable to consider more general systems. Therefore we introduce the notion of a rule system.
Definition 5.17 (Rule System $MP + D$)

Let $D$ be a set of general inference rules, e.g. mappings, which assign a formula $\psi$ to a finite set of formulae $\varphi_1, \varphi_2, \ldots, \varphi_n$. We write

$$\varphi_1, \varphi_2, \ldots, \varphi_n \psi.$$ 

$MP + D$ is the **rule system** which emerges from adding the rules in $D$ to modus ponens. For $W \subseteq Fml$ let

$$Cn^D(W)$$

be the set of all formulae $\varphi$, which can be derived from $W$ and the inference rules from $MP+D$. 
We bear in mind that $C_n(W)$ is defined semantically but $C_n^D(W)$ is defined syntactically (using the notion of proof). Both sets are equal according to the completeness-theorem in the special case $D = \emptyset$.

**Lemma 5.18 (Properties of $C_n^D$)**

Let $D$ be a set of general inference rules and $W \subseteq \text{Fml}$. Then:

1. $C_n(W) \subseteq C_n^D(W)$.
2. $C_n^D(C_n^D(W)) = C_n^D(W)$.
3. $C_n^D(W)$ is the smallest set which is closed in respect to $D$ and contains $W$. 
Question:
What is the difference between an inference rule \( \frac{\varphi}{\psi} \) and the implication \( \varphi \rightarrow \psi \)?

Assume we have a set \( T \) of formulae and we choose two constants \( p, q \in \mathcal{L} \). We can either consider

(1) \( T \) together with MP and \( \{ \frac{p}{q} \} \)

or

(2) \( T \cup \{ p \rightarrow q \} \) together with MP
1. Case:

\[ \text{Cn} \{ \frac{p}{q} \} (T), \]

2. Case:

\[ \text{Cn} (T \cup \{ p \rightarrow q \}). \]

If \( T = \{ \neg q \} \), then we have in (2):

\[ \neg p \in \text{Cn} (T \cup \{ p \rightarrow q \}), \text{ but not in (1)}. \]
It is well-known, that any formula $\phi$ can be written as a \textbf{conjunction of disjunctions}

\[
\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} \phi_{i,j}
\]

The $\phi_{i,j}$ are just constants or negated constants. The $n$ disjunctions $\bigvee_{j=1}^{m_i} \phi_{i,j}$ are called \textbf{clauses} of $\phi$.

\textbf{Normalform}

Instead of working on \textit{arbitrary formulae}, it is sometimes easier to work on \textit{finite sets of clauses}. 
A resolution calculus for SL

The resolution calculus is defined over the language $\mathcal{L}^{\text{res}} \subseteq \mathcal{L}_{\text{SL}}$ where the set of well-formed formulae $Fml_{\mathcal{L}_{\text{res}}}^{\text{Res}}$ consists of all disjunctions of the following form

$$A \lor \neg B \lor C \lor \ldots \lor \neg E,$$

i.e. the disjuncts are only constants or their negations. No implications or conjunctions are allowed. These formulae are also called clauses.

- $\Box$ is also a clause: the empty disjunction.
Set-notation of clauses

A disjunction $A \lor \neg B \lor C \lor \ldots \lor \neg E$ is often written as a set

$$\{A, \neg B, C, \ldots, \neg E\}.$$  

Thus the set-theoretic union of such sets corresponds again to a clause: $\{A, \neg B\} \cup \{A, \neg C\}$ represents $A \lor \neg B \lor \neg C$. Note that the empty set $\emptyset$ is identified with $\Box$. 
We define the following inference rule on $Fml_{Res}$:

**Definition 5.19 (SL resolution)**

Let $C_1, C_2$ be clauses (disjunctions). Deduce the clause $C_1 \lor C_2$ from $C_1 \lor A$ and $C_2 \lor \neg A$:

$$(Res) \quad \frac{C_1 \lor A, \ C_2 \lor \neg A}{C_1 \lor C_2}$$

If $C_1 = C_2 = \emptyset$, then $C_1 \lor C_2 = \Box$. 

Prof. Dr. Jürgen Dix

Clausthal, SS 2011
If we use the set-notation for clauses, we can formulate the inference rule as follows:

**Definition 5.20 (SL resolution (Set notation))**

Deduce the clause $C_1 \cup C_2$ from $C_1 \cup \{A\}$ and $C_2 \cup \{-A\}$:

\[
(\text{Res}) \quad \frac{C_1 \cup \{A\}, \ C_2 \cup \{-A\}}{C_1 \cup C_2}
\]

Again, we identify the empty set $\emptyset$ with $\square$. 
Definition 5.21 (Resolution Calculus for SL)

We define the resolution calculus \( \text{Robinson}_{SL}^{L_{res}} = \langle \emptyset, \{\text{Res}\} \rangle \) as follows. The underlying language is \( L_{res}^{res} \subseteq L_{SL} \) defined on Slide 402 together with the well-formed formulae \( Fml_{L_{res}}^{Res} \).

Thus there are no axioms and only one inference rule. The well-formed formulae are just clauses.

Question:

Is this calculus correct and complete?
Answer:

It is correct, but not complete!

But every problem of the kind “$T \models \phi$” is equivalent to

“$T \cup \{\neg \phi\}$ is unsatisfiable”

or rather to

$$T \cup \{\neg \phi\} \vdash \Box$$

($\vdash$ stands for the calculus introduced above).

Theorem 5.22 (Completeness of Resolution Refutation)

*If* $M$ *is an unsatisfiable set of clauses then the empty clause* $\Box$ *can be derived in Robinson* $^{SL}_{L_{res}}$.

We also say that resolution is refutation complete.
5.4 Wumpus in SL
5 Knowledge Engineering (1)
5.4 Wumpus in SL

Prof. Dr. Jürgen Dix
Clausthal, SS 2011
5 Knowledge Engineering (1)

5.4 Wumpus in SL

<table>
<thead>
<tr>
<th>1,1</th>
<th>2,1</th>
<th>3,1</th>
<th>4,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1,2</th>
<th>2,2</th>
<th>3,2</th>
<th>4,2</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1,3</th>
<th>2,3</th>
<th>3,3</th>
<th>4,3</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
<td>OK</td>
<td>OK</td>
<td>OK</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>1,4</th>
<th>2,4</th>
<th>3,4</th>
<th>4,4</th>
</tr>
</thead>
</table>

\[ A = \text{Agent} \]
\[ B = \text{Breeze} \]
\[ G = \text{Glitter, Gold} \]
\[ OK = \text{Safe square} \]
\[ P = \text{Pit} \]
\[ S = \text{Stench} \]
\[ V = \text{Visited} \]
\[ W = \text{Wumpus} \]
<table>
<thead>
<tr>
<th></th>
<th>1,4</th>
<th>2,4</th>
<th>3,4</th>
<th>4,4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,3</td>
<td>W!</td>
<td>2,3</td>
<td>3,3</td>
<td>4,3</td>
</tr>
<tr>
<td>1,2</td>
<td>A</td>
<td>2,2</td>
<td>3,2</td>
<td>4,2</td>
</tr>
<tr>
<td>1,1</td>
<td>V</td>
<td>2,1</td>
<td>3,1</td>
<td>P!</td>
</tr>
</tbody>
</table>

**Legend:**

- **A** = Agent
- **B** = Breeze
- **G** = Glitter, Gold
- **OK** = Safe square
- **P** = Pit
- **S** = Stench
- **V** = Visited
- **W** = Wumpus

(a)
Language definition:

\[
\begin{align*}
S_{i,j} & \quad \text{stench} \\
B_{i,j} & \quad \text{breeze} \\
Pit_{i,j} & \quad \text{is a pit} \\
Gl_{i,j} & \quad \text{glitters} \\
W_{i,j} & \quad \text{contains Wumpus}
\end{align*}
\]

General knowledge:

\[
\begin{align*}
\neg S_{1,1} & \rightarrow (\neg W_{1,1} \land \neg W_{1,2} \land \neg W_{2,1}) \\
\neg S_{2,1} & \rightarrow (\neg W_{1,1} \land \neg W_{2,1} \land \neg W_{2,2} \land \neg W_{3,1}) \\
\neg S_{1,2} & \rightarrow (\neg W_{1,1} \land \neg W_{1,2} \land \neg W_{2,2} \land \neg W_{1,3}) \\
S_{1,2} & \rightarrow (W_{1,3} \lor W_{1,2} \lor W_{2,2} \lor W_{1,1})
\end{align*}
\]
Knowledge after the 3rd move:

\[ \neg S_{1,1} \land \neg S_{2,1} \land S_{1,2} \land \neg B_{1,1} \land B_{2,1} \land \neg B_{1,2} \]

**Question:** Can we deduce that the wumpus is located at (1,3)?

**Answer:** Yes. Either via resolution or using our Hilbert-calculus.
Problem:

We want more: given a certain situation we would like to determine the **best** action, i.e. to ask a query which gives us back such an action. **This is impossible in SL**: we can only check for each action whether it is good or not and then, by comparison, try to find the best action.

But we can check for each action if it should be done or not. Therefore we need additional axioms:

\[
\begin{align*}
A_{1,1} \land East \land W_{2,1} & \rightarrow \neg \text{Forward} \\
A_{1,1} \land East \land Pit_{2,1} & \rightarrow \neg \text{Forward} \\
A_{i,j} \land Gl_{i,j} & \rightarrow \text{TakeGold}
\end{align*}
\]
5 Knowledge Engineering (1)
5.4 Wumpus in SL

Prof. Dr. Jürgen Dix Clausthal, SS 2011
5 Knowledge Engineering (1)
5.4 Wumpus in SL

Disadvantages

- actions can only be guessed
- database must be changed continuously
- the set of rules becomes very big because there are no variables

Using an appropriate formalisation (additional axioms) we can check if

\[ \text{KB} \vdash \neg \text{action} \quad \text{or} \quad \text{KB} \vdash \text{action} \]

But it can happen that neither one nor the other is deducible.
5.5 A Puzzle
We now want to formalize a "Logelei" and solve it with a *theorem prover*.

"Logelei" from "Die Zeit“ (1)

Alfred ist als neuer Korrespondent in Wongowongo. Er soll über die Präsidentschaftswahlen berichten, weiß aber noch nichts über die beiden Kandidaten, weswegen er sich unter die Leute begibt, um Infos zu sammeln. Er befragt eine Gruppe von Passanten, von denen drei Anhänger der Entweder-oder-Partei sind und drei Anhänger der Konsequenten.
Auf seinem Notizzettel notiert er stichwortartig die Antworten.

A: »Nachname Songo: Stadt Rongo«,
B: »Entweder-oder-Partei: älter«,
C: »Vorname Dongo: bei Umfrage hinten«,
A: »Konsequenzen: Vorname Mongo«,
B: »Stamm Bongo: Nachname Gongo«,
C: »Vorname Dongo: jünger«,
D: »Stamm Bongo: bei Umfrage vorn«,
E: »Vorname Mongo: bei Umfrage hinten«,
F: »Konsequenzen: Stamm Nongo«,
D: »Stadt Longo: jünger«,
E: »Stamm Nongo: jünger«,
F: »Konsequenzen: Nachname Gongo«.
Jetzt grübelt Alfred. Er weiß, dass die Anhänger der Entweder-oder-Partei (A, B und C) immer eine richtige und eine falsche Aussage machen, während die Anhänger der Konsequenten (D, E und F) entweder nur wahre Aussagen oder nur falsche Aussagen machen. Welche Informationen hat Alfred über die beiden Kandidaten?

(By Zweistein)
**Towards a solution**

- Selection of the language (Propositional Logic, Predicate Logic,...).
- Analysis and formalization of the problem.
- Transformation to the input format of a prover.
- Output of a solution, i.e. a **model**.
### Definition of the constants

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{Sur}_x,\text{Songo}$</td>
<td>$x$’s surname is Songo</td>
</tr>
<tr>
<td>$\text{Sur}_x,\text{Gongo}$</td>
<td>$x$’s surname is Gongo</td>
</tr>
<tr>
<td>$\text{First}_x,\text{Dongo}$</td>
<td>$x$’s first name is Dongo</td>
</tr>
<tr>
<td>$\text{First}_x,\text{Mongo}$</td>
<td>$x$’s first name is Mongo</td>
</tr>
<tr>
<td>$\text{Tribe}_x,\text{Bongo}$</td>
<td>$x$ belongs to the Bongos</td>
</tr>
<tr>
<td>$\text{Tribe}_x,\text{Nongo}$</td>
<td>$x$ belongs to the Nongos</td>
</tr>
<tr>
<td>$\text{City}_x,\text{Rongo}$</td>
<td>$x$ comes from Rongo</td>
</tr>
<tr>
<td>$\text{City}_x,\text{Longo}$</td>
<td>$x$ comes from Longo</td>
</tr>
<tr>
<td>$A_x$</td>
<td>$x$ is the senior candidate</td>
</tr>
<tr>
<td>$J_x$</td>
<td>$x$ is the junior candidate</td>
</tr>
<tr>
<td>$H_x$</td>
<td>$x$’s poll is worse</td>
</tr>
<tr>
<td>$V_x$</td>
<td>$x$’s poll is better</td>
</tr>
</tbody>
</table>

Here $x$ is a candidate, i.e. $x \in \{a, b\}$. So we have 24 constants in total.
The correspondent Alfred noted 12 statements about the candidates (each interviewee gave 2 statements, $\phi, \phi'$) which we enumerate as follows

$$\phi_A, \phi'_A, \phi_B, \phi'_B, \ldots, \phi_F, \phi'_F,$$

All necessary symbols are now defined, and we can formalize the given statements.
Formalization of the statements

\[ \phi_A \iff (\text{Sur}_a \land \text{Songo} \land \text{City}_a \land \text{Rongo}) \lor (\text{Sur}_b \land \text{Songo} \land \text{City}_b \land \text{Rongo}) \]

\[ \phi'_A \iff \text{First}_b \land \text{Mongo} \]

\[ \phi_B \iff \text{A}_a \]

\[ \phi'_B \iff (\text{Tribe}_a \land \text{Bongo} \land \text{Sur}_a \land \text{Gongo}) \lor (\text{Tribe}_b \land \text{Bongo} \land \text{Sur}_b \land \text{Gongo}) \]

\[ \vdots \]
Furthermore, **explicit** conditions between the statements are given, e.g.

\[(\phi_A \land \neg \phi'_A) \lor (\neg \phi_A \land \phi'_A)\]

and

\[(\phi_D \land \phi'_D) \lor (\neg \phi_D \land \neg \phi'_D).\]

Analogously, for the other statements.

**Is this enough information to solve the puzzle?**

E.g., can the following formula be satisfied?

\[\text{Sur}_a,\text{Songo} \land \text{Sur}_a,\text{Gongo}\]
We also need **implicit** conditions (**axioms**) which are required to solve this problem. It is necessary to state that each candidate has only one name, comes from one city, etc.

We need the following background knowledge...

\[
\begin{align*}
\text{Sur}_x, \text{Songo} & \iff \neg \text{Sur}_x, \text{Gongo} \\
\text{First}_x, \text{Dongo} & \iff \neg \text{First}_x, \text{Mongo} \\
\vdots \\
H_x & \iff \neg V_x
\end{align*}
\]

Can we abstain from these axioms by changing our representation of the puzzle?
What is still missing?
Can we prove that when $a$’s poll is worse, then $s$’s poll is better?
We need to state the relationships between these attributes:

\[ H_x \leftrightarrow V_y \]
\[ A_x \leftrightarrow J_y \]

Finally, we have modeled all “sensible” information. Does this yield a unique model?

No! There are 6 models in total, but this is all right. It just means there is no unique solution.
What if a unique model is desirable?

Often, there are additional assumptions hidden “between the lines”. Think, for example, of deductions by Sherlock Holmes (or Miss Marple, Spock etc).
For example, it might be sensible to assume that both candidates come from different cities:

\[ \text{City}_x, \text{Rongo} \leftrightarrow \text{City}_y, \text{Longo} \]

Indeed, with this additional axiom there is an unique model.

But, be careful...

... this additional information may not be justified by the nature of the task!
Tractatus Logico-Philosophicus

1. Die Welt ist alles was der Fall ist.
   1.1 Die Welt ist die Gesamtheit der Tatsachen, nicht der Dinge.

2. Was der Fall ist, die Tatsache, ist das Bestehen von Sachverhalten.

3. Das logische Bild der Tatsachen ist der Gedanke.

4. Der Satz ist eine Wahrheitsfunktion der Elementarsätze.

5. Die allgemeine Form der Wahrheitsfunktion ist: \([\bar{p}, \bar{\xi}, N(\bar{\xi})]\).

6. Wovon man nicht sprechen kann, darüber muß man schweigen.
6. Hoare Calculus

- Verification
- Core Programming Language
- Hoare Logic
- Proof Calculi: Partial Correctness
- Proof Calculi: Total Correctness
- Sound and Completeness
Content of this chapter (1):

We introduce a **calculus** to prove **correctness** of computer programs.

**Verification:** We argue why formal methods are useful/necessary for program verification.

**Core Programming Language:** An **abstract** but powerful programming language is introduced.

**Hoare Logic:** We introduce the **Hoare Calculus** which operates on triples \( \{ \phi_{pre} \} P \{ \psi_{post} \} \).
Content of this chapter (2):

**Proof Calculi (Partial Correctness):** Here we present three calculi (proof rules, annotation calculus, and the weakest precondition calculus) for partial correctness.

**Proof Calculi (Total Correctness):** Partial correctness is trivially satisfied when a program does not terminate. Therefore, we consider total correctness as well and an extension of the calculi to deal with it. We consider the partial and total correctness of programs.

**Sound & Completeness:** We briefly discuss the sound and completeness of the Hoare calculus.
6.1 Verification
Why to formally specify/verify code?

- Formal specifications are often important for unambiguous documentations.
- Formal methods cut down software development and maintenance costs.
- Formal specification makes software easier to reuse due to a clear specification.
- Formal verification can ensure error-free software required by safety-critical computer systems.
Verification of Programs

How can we define the state of a program?

The state of a program is given by the contents of all variables.

What about the size of the state-space of a program?

The state space is usually infinite! Hence, the technique of model checking is inappropriate.
Software Verification Framework

Main reasons for **formal specifications**:

- **Informal** descriptions often lead to **ambiguities** which can result in serious (and potentially expensive) design flaws.
- Without formal specifications a **rigorous verification** is not possible!
Methodology

We assume the following methodology:

1. Build an informal description $D$ of the program and the domain.
2. Convert $D$ into an equivalent formula $\phi$ in a suitable logic.
3. (Try to) build a program $P$ realizing $\phi$.
4. Prove that $P$ satisfies $\phi$. 
Methodology (2)

Some points to think about:

(i) Point 2 is a non-trivial and non-formal problem and thus “cannot be proven”: $D$ is an informal specification!

(ii) Often there are alternations between 3 and 4.

(iii) Sometimes one might realize that $\phi$ is not equivalent to $D$ and thus one has to revise $\phi$.

(iv) Often, $P$ must have a specific structure to prove it against $\varphi$.

In this lecture, we will focus on points 3, 4, and (iv).
Some Properties of the Procedure

Proof-based: Not every state of the system is considered (there are infinitely many anyway), rather a proof for correctness is constructed: this works then for all states.

Semi-automatic: Fully automatic systems are desireable but they are not always possible: undecidability, time constraints, efficiency, and “lack of intelligence”.
Some Properties of the Procedure (2)

Property-oriented: Only certain properties of a program are proven and not the “complete” behavior.

Application domain: Only sequential programs are considered.

Pre/post-development: The proof techniques are designed to be used during the programming process (development phase).
6.2 Core Programming Language
Core Programming Language

To deal with an up-to-date programming language like Java or C++ is out of scope of this introductory lecture. Instead we identify some core programming constructs and abstract away from other syntactic and language-specific variations.

Our programming language is built over

1. integer expressions,
2. boolean expressions, and
3. commands.
Definition 6.1 (Program Variables)

We use $\mathcal{V}ars$ to denote the set of (program) variables. Typically, variables will be denoted by $u, \ldots, x, y, z$ or $x_1, x_2, \ldots$.

Definition 6.2 (Integer Expression)

Let $x \in \mathcal{V}ars$. The set of integer expressions $I$ is given by all terms generated according to the following grammar:

$$I ::= 0 \mid 1 \mid x \mid (I + I) \mid (I - I) \mid (I \cdot I)$$

We also use $-I$ to stand for $0 - I$. 
Although the term “1 – 1” is different from “0”, its meaning (semantics) is the same. We identify “1 + 1 + … + 1” with “n” (if the term consist of n 1’s). The precise notion of meaning will be given in Definition 6.10.

Integer expressions are terms over a set of function symbols \( Func \), here

\[
Func = Vars \cup \{0, 1, +, −, \cdot\}
\]

Note also, that we consider elements from \( \mathbb{Z} \) as constants with their canonical denotation! Thus we write “3” instead of “1 + 1 + 1”.

**Example 6.3 (Some integer expressions)**

5, \( x \), 6 + (3 \cdot x), \( x \cdot y + z - 3 \), −\( x \cdot x \), …

Note that \( x^x \) or \( y! \) are not integer expressions. **Do we need them?**
Definition 6.4 (Boolean Expression)

The set of **Boolean expressions** $B$ is given by all formulae generated according to the following grammar:

$$B ::= \top \mid (\neg B) \mid (B \land B) \mid (I < I)$$

We also use $(B \lor B)$ as an abbreviation of $\neg(\neg B \land \neg B)$ and $\Box$ as an abbreviation for $\neg \top$. Boolean expressions are **formulae** over the set of relation symbols $\mathcal{P}_{red} = \{\top, <\}$. So, boolean expressions are similar to **sentential logic** formulae.

Note that we use $I = I'$ to stand for $\neg (I < I') \land \neg (I' < I)$. We also write $I \neq I'$ for $\neg (I = I')$. 
Formulae over $\mathbb{Z}$

When describing the Hoare calculus, we also need formulae to express (1) what should hold before a program is executed, and (2) what should be the case after the program has been executed. These formulae are built using the boolean expressions just introduced.

**Definition 6.5 (Formulae over $\mathbb{Z}$)**

The set of formulae $\mathcal{Fml}_{\mathcal{V}ars}$ is given by all formulae generated according to the following grammar:

$$
\phi ::= \exists x \phi \mid B \mid (\neg \phi) \mid (\phi \land \phi)
$$

where $B$ is a boolean expression according to Definition 6.4 and $x \in \mathcal{V}ars$.

We note that “$\forall x \phi$” is just an abbreviation for “$\neg \exists x \neg \phi$”
Formulas over $\mathbb{Z}$ (2)

We have to give a meaning (semantics) to the syntactic constructs! Our model is given by $\mathbb{Z}$ where $\mathbb{Z}$ is the set of integer numbers. Note that in $\mathbb{Z}$ we have a natural meaning of the constructs $0, 1, -, +, \cdot, <$.

Which of the following formulae are true in $\mathbb{Z}$?

1. $\forall x \forall y \forall z \left( x \cdot x + y \cdot y + z \cdot z > 0 \right)$,
2. $\forall x \exists y \left( y < x \right)$,
3. $x \cdot x < x \cdot x \cdot x$,
4. $x + 4 < x + 5$. 

Prof. Dr. Jürgen Dix

Clausthal, SS 2011
Formulae over $\mathbb{Z}$ (3)

Note that when evaluating a formula with free variables, there are two possibilities:

1. Either we need to assign values to these variables, or
2. we add “∀” quantifiers to bind all variables.

The truth of a formula can only be established if the formula is a sentence, i.e. does not contain free variables. We will be more formal in Chapter 7.

Let $\phi(x, y)$ be the formula $\exists z \ x + z < y$ and let $n, n' \in \mathbb{Z}$. Then we denote

- by $\forall \phi(x, y)$ the formula $\forall x \forall y (\exists z \ x + z < y)$,
- by $\phi(x, y)[n/x, n'/y]$ the formula where $x$ is replaced by $n$ and $y$ is replaced by $n'$: $\exists z \ n + z < n'$. 

Formulae over $\mathbb{Z}$ (4)

How large is this class of formulae over $\mathbb{Z}$? Is it expressive enough?

1. How can we express the function $x!$ or $x^x$?
2. Which functions are not expressible? Are there any?
3. Is there an algorithm to decide whether a given sentence $\phi$ (formulae without free variables) holds in $\mathbb{Z}$, i.e. whether $\mathbb{Z} \models \phi$?
4. Attention: everything radically changes, when we do not allow multiplication! Then the resulting theory is decidable.
Formulae over $\mathbb{Z}$ (5)

Let us express some functions $f : \mathbb{Z} \rightarrow \mathbb{Z}$ that are not available as integer expressions.

**Definition 6.6 (Functions Expressible over $\mathbb{Z}$)**

A function $f : \mathbb{Z} \rightarrow \mathbb{Z}$ is expressible over $\mathbb{Z}$ if there is a formula $\Phi(x, y)$ with $x, y$ as the only free variables such that the following holds for all $z, z' \in \mathbb{Z}$:

$$\mathbb{Z} \models \Phi(x, y)[z/x, z'/y] \iff f(z) = z'$$
6 Hoare Calculus

6.2 Core Programming Language

Formulae over $\mathbb{Z}$ (6)

$x!$: Not obvious.

$x^x$: Not obvious.

Using Gödelization, it can be shown that all naturally occurring functions can be expressed. In fact, all recursive functions are expressible. We shall therefore use functions like $x!$ as macros (knowing that they can be expressed as formulae in the language).
Definition 6.7 (Program)

The set $\mathcal{P}rog$ of (while) programs over $\mathcal{V}ars$, $\mathcal{I}$ and $\mathcal{B}$ is given by all well-formed sequences which can be formed according to the following grammar:

$$C ::= \text{skip} \mid x := I \mid C; C \mid \text{if } B \{C\} \text{ else } \{C\} \mid \text{while } B \{C\}$$

where $B \in \mathcal{B}$, $I \in \mathcal{I}$, and $x \in \mathcal{V}ars$.

We usually write programs in lines (for better readability).
skip: Do nothing.

\[ x := I: \text{Assign } I \text{ to } x. \]

\[ C_1; C_2: \text{Sequential execution: } C_2 \text{ is executed after } C_1 \text{ provided that } C_1 \text{ terminates.} \]

\[ \text{if } B \{ C_1 \} \text{ else } \{ C_2 \}: \text{If } B \text{ is true then } C_1 \text{ is executed otherwise } C_2. \]

\[ \text{while } B \{ C \}: \text{C is executed as long as } B \text{ is true.} \]

The statement “\( B \) is true” is defined in Definition 6.11.
Example 6.8

What does the following program $\text{Fac}(x)$ calculate?

\[
\begin{align*}
y & := 1; \\
z & := 0; \\
\text{while } z \neq x \{ \\
& \quad z := z + 1; \\
& \quad y := y \cdot z \\
\}\end{align*}
\]
Is this class of programs large or small?

Can we express everything we want?

1. No, because we need better software engineering constructs.
2. Yes, because we have "while" and therefore we can do anything.
3. No, because we can not simulate (emulate) Java or C++.
4. It is already too large, because we assume that we can store arbitrary numbers. This is clearly not true on real computers.
5. Yes, because this class of programs corresponds to the class of deterministic Turing machines. And we cannot aim for more.
While Programs = Turing Machines

Turing Machines

The class of programs we have introduced corresponds exactly to programs of Turing machines. Thus it is an \textit{idealization} (arbitrary numbers can be stored in one cell) and therefore it is much more expressive than any real programming language. But playing quake requires a lot of coding …

- In particular, there is no algorithm to decide whether a given program terminates or not.
- The set of all terminating programs is recursive enumerable, but not recursive.
- Therefore the set of non-terminating programs is not even recursively enumerable.
Meaning of a Program

Definition 6.9 (State)

A state $s$ is a mapping

$$s : \text{Vars} \rightarrow \mathbb{Z}$$

A state assigns to each variable an integer. The set of all states is denoted by $S$.

The semantics of a program $P$ is a partial function

$$[P] : S \rightarrow S$$

that describes how a state $s$ changes after executing the program.
Definition 6.10 (Semantics: Integer Expression)

The semantics of integer expressions is defined:

\[
\begin{align*}
[0]_s &= 0 \\
[1]_s &= 1 \\
[x]_s &= s(x) \text{ for } x \in \text{Vars} \\
[E \ast E']_s &= [E]_s \ast [E']_s \text{ for } \ast \in \{+,-,\cdot\}
\end{align*}
\]

Definition 6.11 (Meaning: Boolean Expression)

The semantics of a Boolean expression is given as for sentential logic; that is, we write \( \mathbb{Z}, s \models B \) if \( B \) is true in \( \mathbb{Z} \) wrt to state (valuation) \( s \).
**Definition 6.12 (Meaning: Satisfaction \( \mathbb{Z}, s \models \phi \))**

The **semantics of a formula** \( \phi \) is defined inductively. We have already defined it in Definition 6.11 for atomic expressions. Arbitrary formulae can also contain the quantifier \( \exists \).

\[
\mathbb{Z}, s \models \exists x \phi(x) \text{ iff there is a } n \in \mathbb{Z} \text{ such that } \mathbb{Z}, s \models \phi(x)[n/x]
\]

1. \( \exists x \phi : \exists x \ 3x < 4 \),
2. \( \exists x \phi : \exists x \ 3x < 4y \).
Definition 6.13 (Meaning: Program)

1. \([\text{skip}](s) = s\)

2. \([x := t](s) = t \text{ where } t(y) = \begin{cases} s(y) & \text{if } y \neq x \\ [I]_s & \text{else.} \end{cases}\)

3. \([C_1; C_2](s) = [C_2]([C_1](s))\)

4. \([\text{if } B \{C_1\} \text{ else } \{C_2\}](s) = \begin{cases} [C_1](s) & \text{if } Z, s \models B, \\ [C_2](s) & \text{else.} \end{cases}\)

5. \([\text{while } B \{C\}](s) = \begin{cases} \text{while } B \{C\}([C](s)) & \text{if } Z, s \models B, \\ s & \text{else.} \end{cases}\)

Note that the recursive definition of the while cases is the reason that \([\cdot]\) might be a partial function: it is (perhaps) not everywhere defined.
6.3 Hoare Logic
Hoare Triples

How can we prove that a program/method $P$ really does what it is intended to do?
We describe the desired state of the (overall) program before and after the execution of $P$.
For example, let $P(x)$ be a program that should return a number whose square is strictly less than $x$.

Is the correctness of the program ensured when we require that $P(x) \cdot P(x) < x$?
Not completely! What if $x < 0$? So, the precondition is also very important!
Definition 6.14 (Hoare Triple)

Let $\phi$ and $\psi$ be formulae of $Fml_{\nu \text{ars}}$ and $P$ be a program. A Hoare triple is given by

$$\{ \phi \} P \{ \psi \}$$

where $\phi$ is said to be the precondition and $\psi$ the postcondition.
A Hoare triple \( \{ \phi \}\text{P}\{ \psi \} \) is read as follows:

- If the overall program is in a state that satisfies \( \phi \), then,
- after the execution (and termination) of \( \text{P} \) the resulting state of the program satisfies \( \psi \).

Let \( \text{P} \) be the program given above and \( y \) be the variable returned. Then the following Hoare triple would be a valid specification:

\[ \{ x > 0 \}\text{P}\{ y \cdot y < x \}. \]
Partial and Total Correctness

We already introduced the informal reading of Hoare triples. Now we will be more formal. There are two cases one has to distinguish: Programs that terminate and ones which do not. Accordingly, we have two definitions of correctness:

**Partially correct**: $\psi$ holds after execution of $P$, provided that $P$ terminates,

**Totally correct**: we require in addition that $P$ terminates.
Definition 6.15 (Partial Correctness)

A triple \( \{ \phi \} P \{ \psi \} \) is satisfied under **partial correctness**, written as

\[ \models^p \{ \phi \} P \{ \psi \}, \]

if for each state \( s \)

\[ Z, [P](s) \models \psi \]

provided that \( Z, s \models \phi \) and \( [P](s) \) are defined.

The following program is **always partially correct**: while \( \top \) \( \{ x := 0 \} \), for arbitrary pre and postconditions.
**Definition 6.16 (Total Correctness)**

A triple \( \{ \phi \} P \{ \psi \} \) is satisfied under **total correctness**, written as

\[
\models^t \{ \phi \} P \{ \psi \},
\]

if for each state \( s \), \( [P](s) \) is defined and

\[
\mathbb{Z}, [P](s) \models \psi
\]

provided that \( \mathbb{Z}, s \models \phi \).

The following program is usually **not** totally correct: while \( \top \) \( \{ x := 0 \} \). Why “usually”? 
Example 6.17

Consider the following program $\text{Succ}(x)$:

\begin{verbatim}
    a := x + 1;
    if (a − 1 = 0) {
        y := 1
    } else{
        y := a
    }
\end{verbatim}

Under which semantics does the program satisfy $\{\top\} \text{Succ}\{y = x + 1\}$?
Note, that the last program is only one program (and a very silly one) that ensures the condition \( \{ \top \} \mathsf{Succ}\{ y = x + 1 \} \). There are many more.

**Example 6.18**

Recall the program \( \mathsf{Fac}(x) \) stated in Example 6.8. Which of the following statements are correct?

- \( \models^t \{ x \geq 0 \} \mathsf{Fac}\{ y = x! \} \),
- \( \models^t \{ \top \} \mathsf{Fac}\{ y = x! \} \),
- \( \models^p \{ x \geq 0 \} \mathsf{Fac}\{ y = x! \} \) and
- \( \models^p \{ \top \} \mathsf{Fac}\{ y = x! \} \).
Program and Logical Variables

The pre- and postconditions in a Hoare triple may contain two kinds of variables:

- **program variables** and
- **logical variables**.

Given a program $P$ the former kind occurs in the program whereas the latter refers to fresh variables.

The following example makes clear why we need logical variables.
Example 6.19

The following program Fac2 works as well.

\[
\begin{align*}
y &:= 1; \\
\text{while } (x \neq 0) \{ \\
\quad &y := y \cdot x; \\
\quad &x := x - 1 \\
\} 
\end{align*}
\]

Why is it not a good idea to use the Hoare triple \{ \( x \geq 0 \) \} Fac2 \{ y = x! \} in this case?

What about \{ \( x = x_0 \land x \geq 0 \) \} Fac2 \{ y = x_0! \}? The variable \( x_0 \) is a logical variable!
6.4 Proof Calculi: Partial Correctness
Proof Rules

We have introduced a *semantic* notion of (partial and total) correctness. As for resolution we are after *syntactic versions* ($\vdash^t$ and $\vdash^p$) which can be used on computers.

**Sound Calculus:** Can we define a calculus, that allows us to derive *only valid Hoare triples*?

**Complete Calculus:** Can we define a calculus, that generates *all valid Hoare triples*?
Answer to question 1: Yes, for both versions.
Answer to question 2: No, not even for $\neg^p$:

$\neg^p \{\top\} P \{\Box\}$ iff $P$ is not terminating.

And this set is not recursive enumerable.

The following rules were proposed by R. Floyd and C.A.R. Hoare. A rule for each basic program construct is presented.

If a program is correct we may be able to show it by only applying the following rules (compare with Modus Ponens).
Composition

Definition 6.20 (Composition Rule (comp))

\[
\begin{array}{c}
\{ \phi \} C_1 \{ \eta \} \\
\{ \eta \} C_2 \{ \psi \}
\end{array}
\quad \Rightarrow \quad
\{ \phi \} C_1 ; C_2 \{ \psi \}
\]

In order to prove that \( \{ \phi \} C_1 ; C_2 \{ \psi \} \) we have to prove the Hoare triples \( \{ \phi \} C_1 \{ \eta \} \) and \( \{ \eta \} C_2 \{ \psi \} \) for some appropriate \( \eta \).
An axiom

**Definition 6.21 (Assignment Rule (assign))**

\[
\{\psi[I/x]\}x := I\{\psi\}
\]

The rule is self-explanatory. Recall that \(\psi[I/x]\) denotes the formula that is equal to \(\psi\) but each free occurrence of \(x\) is replaced by \(I\).
Another axiom

Definition 6.22 (Skip Rule (skip))

\[
\{ \phi \} \text{skip} \{ \phi \}
\]
If Rule

Definition 6.23 (If Rule (if))

\[
\begin{align*}
\{ \phi \land B \} & \xrightarrow{C_1} \{ \psi \} \\
\{ \phi \land \neg B \} & \xrightarrow{C_2} \{ \psi \}
\end{align*}
\]

\{ \phi \} \text{ if } B \{ C_1 \} \text{ else } \{ C_2 \} \{ \psi \}

In order to prove the conclusion we prove that \( \psi \) holds for both possible program executions of the if-rule: when \( B \) holds and when it does not.
Partial-While Rule

Definition 6.24 (Partial-While Rule (while))

\[
\{ \psi \land B \} C \{ \psi \} \\
\{ \psi \} \text{ while } B \{ C \} \{ \psi \land \neg B \}
\]

The while-rule is the most sophisticated piece of code; it may allow infinite looping. The formula \( \psi \) in the premise plays a decisive rule: \( \psi \) is true \textbf{before} and \textbf{after} the execution of \( C \), i.e. \( C \) does not change the truth value of \( \psi \).
### Definition 6.25 (Invariant)

An **invariant** of the while-statement while $B \{ C \}$ is any formula $\psi$ such that

$\models p \{ \psi \land B \} C \{ \psi \}$.

The conclusion says that $\psi$ does not change, even when $C$ is executed several times and if $C$ terminates then $B$ is false.
Example 6.26

What is an **invariant** of the following program?

\[
y := 1; \\
z := 0; \\
\text{while } z \neq x \{ \\
\quad z := z + 1; \\
\quad y := y \cdot z \\
\}
\]

An invariant is \( y = z \).
Implication Rule

Definition 6.27 (Implication Rule (impl))

\[
\begin{align*}
\{\phi\} \mathcal{C} \{\psi\} \\
\{\phi'\} \mathcal{C} \{\psi'\}
\end{align*}
\]

whenever \(Z \models \forall (\phi' \rightarrow \phi)\) and \(Z \models \forall (\psi \rightarrow \psi')\).
Implication Rule (2)

The implication rule allows us to **strengthen** the precondition $\phi$ to $\phi'$ and to **weaken** the postcondition $\psi$ to $\psi'$, provided that the two implications hold.

Note, that this rule links program logic with the truths of formulae in $\mathcal{Z}$, which have to be established. We call them **proof obligations**.
We will try to prove the correctness of the program \texttt{Fac} given in Example 6.8. That is, we would like to derive

$$\{\top\}\texttt{Fac}\{y = x!\}$$

For any input state after the execution of \texttt{Fac} the return value $y$ should have the value $x!$. 
We have to start with axioms (by the assignment rule):

\[
\begin{align*}
\{1=1\}y:=1\{y=1\} & \quad \text{(impl)} \\
\{y=1\}y:=1\{y=1\} & \quad \text{(impl)} \\
\{\top\}y:=1\{y=1\} & \quad \text{(comp)} \\
\{\top\}y := 1; z := 0\{y = 1 \land z = 0\} & \\
\end{align*}
\]

\(\psi_1\)

Again, on top are axioms:

\[
\begin{align*}
\{y \cdot (z+1)=(z+1)!\}z:=z+1\{y \cdot z=z!\} & \quad \text{(impl)} \\
\{y=0\land z\neq x\}z:=z+1\{y \cdot z=z!\} & \quad \text{(comp)} \\
\{y=0\}z:=z+1; y:=y \cdot z\{y=z!\} & \quad \text{(while)} \\
\{y = z!\} \text{ while } z \neq x \{z := z + 1; y := y \cdot z\}\{y = z! \land z = x\} & \\
\end{align*}
\]

\(\psi_2\)

Putting both parts together:

\[
\begin{align*}
\psi_1 & \quad \psi_2 \quad \{y=1\land z=0\} \text{ while } z\neq x \{z:=z+1; y:=y \cdot z\}\{y=x!\} & \quad \text{(impl)} \\
\{\top\} & \quad \{\top\} \text{ Fac}\{y = x!\} & \quad \text{(comp)} \\
\end{align*}
\]
We have to show that the following formulae hold in $\mathbb{Z}$:

1. $\top \to 1 = 1$,
2. $\forall y(y = 1 \to y = 1 \land 0 = 0)$,
3. $\forall x \forall y \forall z(y = z! \land z \neq x \to (y(z + 1) = (z + 1)!))$,
4. $\forall y \forall z(y = 1 \land z = 0 \to y = z!)$,
5. $\forall x \forall y \forall z(y = z! \land z = x \to y = x!)$.
Annotation Calculus

- The proof rules just presented are very similar to Hilbert style calculus rules.
- They inherit some undesirable properties: The proof calculus is not “easy to use”.
- The proof given for the small Fac-program looks already quite complicated.

In this section we present an annotation calculus which is much more convenient for practical purposes.
We consider a program $P$ as a sequence of basic commands:

$$C_1; C_2; \ldots ; C_n$$

That is, none of the commands $C_i$ is directly composed of smaller programs by means of composition. Assume we intend to show that

$$\vdash^p \{\phi\} P \{\psi\}$$

In the annotation calculus we try to find appropriate $\phi_i$, $i = 0, \ldots, n$ such that

if $\vdash^p \{\phi_i\} C_i \{\phi_{i+1}\}$ for all $i = 0, \ldots, n - 1$

then also $\vdash^p \{\phi\} P \{\psi\}$. 
In other words, we **interleave** the program $P$ with intermediate conditions $\phi_i$

$$\{\phi_0\}C_1\{\phi_1\} \; C_2\{\phi_2\} \ldots \{\phi_{n-1}\}C_n\{\phi_n\}$$

where each step $\{\phi_i\}C_i\{\phi_{i+1}\}$ is justified by one of the proof rules given above.

That is, an **annotation calculus** is a way of summarizing the application of the proof rules to a program.

How to find the appropriate $\phi_i$?
We determine “something like” the weakest preconditions successively such that

\[ |-p \{ \phi_0 \} C_1 \{ \phi_1 \} C_2 \{ \phi_2 \} \ldots \{ \phi_{n-1} \} C_n \{ \phi_n \} \]

Under which condition would this guarantee \[ |-p \{ \phi_0 \} P \{ \phi_n \} \]?

It must be the case that \( \mathbb{Z} \models \forall \phi_0 \rightarrow \phi_0 \) and \( \mathbb{Z} \models \forall \phi_n \rightarrow \phi_n \)

Why do we say “something like” the weakest precondition? We come back to this point later.
In the following we label the program $P$ with preconditions by means of *rewrite rules*

\[
\frac{X}{Y}
\]

Such a rule denotes that $X$ can be *rewritten* (or *replaced*) by $Y$.

Each rewrite rule results from a proof rule.
Definition 6.28 (Assignment Rule)

\[
x := E \{ \psi \}
\]

\[
\{ \psi \{ E/x \} \} x := E \{ \psi \}
\]

Definition 6.29 (If Rule (1))

\[
\text{if } B \{ C_1 \} \text{ else } \{ C_2 \} \{ \psi \}
\]

\[
\text{if } B \{ C_1 \{ \psi \} \} \text{ else } \{ C_2 \{ \psi \} \} \{ \psi \}
\]

Definition 6.30 (If Rule (2))

\[
\text{if } B \{ \{ \phi_1 \} \cdots \} \text{ else } \{ \{ \phi_2 \} \cdots \}
\]

\[
\{ (B \rightarrow \phi_1) \land (\neg B \rightarrow \phi_2) \} \text{ if } B \{ \{ \phi_1 \} \cdots \} \text{ else } \{ \{ \phi_2 \} \cdots \}
\]
**Definition 6.31 (Partial-While Rule)**

\[
\text{while } B \{ P \} \quad \{ \phi \} \quad \text{while } B \{ \{ \phi \land B \} P \{ \phi \} \} \{ \phi \land \neg B \}
\]

where \( \phi \) is an **invariant** of the while-loop.

**Definition 6.32 (Skip Rule)**

\[
\text{skip}\{ \phi \} \quad \{ \phi \} \quad \text{skip}\{ \phi \}
\]
Applying the rules just introduced, we end up with a finite sequence

\[ s_1 s_2 \ldots s_m \]

where each \( s_i \) is of the form \( \{\phi\} \) or it is a command of a program (which in the case of if or while commands can also contain such a sequence).

It can also happen that two subsequent elements have the same form: \( \ldots \{\phi\}\{\psi\} \ldots \). Whenever this occurs, we have to show that \( \phi \) implies \( \psi \): a proof obligation (see Slide 11).

**Definition 6.33 (Implied Rule)**

Whenever applying the rules lead to a situation where two formulae stand next to each other \( \ldots \{\phi\}\{\psi\} \ldots \), then we add a proof obligation of the form

\[ \mathbb{Z} \models \forall (\phi \rightarrow \psi). \]
Consider \( \{ \phi \} \) while \( B \) \( \{ P \} \{ \psi \} \). Then the while-rule yields the following construct:

\[
\{ \phi \} \{ \eta \} \text{ while } B \{ \{ \eta \land B \} P \{ \eta \} \} \{ \eta \land \neg B \} \{ \psi \}
\]

That is, we have to show that the invariant \( \eta \) satisfies the following properties:

1. \( \models (\phi \rightarrow \eta) \),
2. \( \models (\eta \land \neg B) \rightarrow \psi \),
3. \( \vdash \{ \eta \} \text{ while } B \{ P \} \{ \eta \land \neg B \} \), and
4. \( \vdash \{ \eta \land B \} P \{ \eta \} \) (this is the fact that it is an invariant).
Example 6.34

Prove: $\vdash^p \{y = 5\}x := y + 1\{x = 6\}$

1. $x := y + 1\{x = 6\}$
2. $\{y + 1 = 6\}x := y + 1\{x = 6\}$ (Assignment)
3. $\{y = 5\}\{y + 1 = 6\}x := y + 1\{x = 6\}$ (Implied)

Example 6.35

Prove: $\vdash^p \{y < 3\}y := y + 1\{y < 4\}$

1. $y := y + 1\{y < 4\}$
2. $\{y + 1 < 4\}y := y + 1\{y < 4\}$ (Assignment)
3. $\{y < 3\}\{y + 1 < 4\}y := y + 1\{y < 4\}$ (Implied)
Example 6.36

\{ y = y_0 \}
\z := 0;
while y \neq 0 do {
    \z := \z + x;
    y := y - 1
}
\{ z = xy_0 \}

Firstly, we have to use the rules of the annotation calculus.
\{y = y_0\} \\
\{I[0/z]\}\} \\
z := 0; \\
\{I\} \\
while y \neq 0 do \\
\{I \land y \neq 0\} \\
\{I[y - 1/y][z + x/z]\}\} \\
z := z + x; \\
\{I[y - 1/y]\}\} \\
y := y - 1 \\
\{I\}\} \\
\{I \land y = 0\} \\
\{z = xy_0\}\} \\

What is a suitable invariant?
We have to choose an invariant such that the following hold in \( \mathbb{Z} \):

1. \( y = y_0 \rightarrow I[0/z] \),
2. \( I \land y \neq 0 \rightarrow I[y - 1/y][z + x/z] \),
3. \( I \land y = 0 \rightarrow z = xy_0 \).

What about \( I : \top \)? What about \( I : (z + xy = xy_0) \)?

It is easy to see that the latter invariant satisfies all three conditions. This proves the partial correctness of \( \{ y = y_0 \} P \{ z = xy_0 \} \).
How to ensure that $\models^p \{ \phi \} P \{ \psi \}$?

**Definition 6.37 (Valid Annotation)**

A **valid annotation** of $\{ \phi \} P \{ \psi \}$ is given if

1. only the rules of the **annotation calculus** are used;
2. each command in $P$ is **embraced by a post and a precondition**;
3. the assignment rule is applied **to each assignment**;
4. each **proof obligation** introduced by the implied rule has to be verified.
Proposition 6.38 ($\vdash p$ iff Valid Annotation)

Constructing valid annotations is equivalent to deriving the Hoare triple $\{\phi\}P\{\psi\}$ in the Hoare calculus:

$$\vdash p \quad \{\phi\}P\{\psi\} \quad \text{iff there is a valid annotation of } \{\phi\}P\{\psi\}.$$

Note, this does not mean that the calculus is complete!
Exercise

Let the program $P$ be:

\[
\begin{align*}
z & := x; \\
z & := z + y; \\
u & := z
\end{align*}
\]

Use the annotation calculus to prove that

$\vdash P \{ \top \} P\{ u = x + y \}$. 
Exercise

Given the program $P$:

\[
\begin{align*}
  a &:= x + 1; \\
  &\text{if } a - 1 = 0 \{ y := 1 \} \text{ else } \{ y := a \}
\end{align*}
\]

Use the annotation calculus to prove that

\[
\vdash_P \{ \top \} P \{ y = x + 1 \}.
\]
Exercise

Given the program \textit{Sum}:

\[
\begin{align*}
&z := 0; \\
&\text{while } x > 0 \{ \\
&\hspace{1em} z := z + x; \\
&\hspace{1em} x := x - 1 \\
&\}\n\end{align*}
\]

Use the annotation calculus to prove that
\[
\begin{align*}
\vdash p \{ x = x_0 \land x \geq 0 \} \text{Sum} \{ z = \frac{x_0(x_0+1)}{2} \}.
\end{align*}
\]
Weakest Liberal Precondition

In principle, the annotation calculus determines the weakest precondition of a program and checks whether the given precondition implies the calculated precondition. I.e. when we start with something of the form

\[ P \{ \psi \} \]

then the annotation calculus leads to a Hoare triple

\[ \{ \phi \} P' \{ \psi \} \]

where \( \phi \) is "something like the weakest precondition":

\[ \vdash P \{ \phi \} P \{ \psi \}. \]

Without the while-command the annotation calculus does calculate the weakest precondition! But in the rule for the while-command some invariant is selected. This does not ensure that the weakest precondition is determined!
A formula $\psi$ is said to be weaker than $\chi$ if it holds that $\mathbb{Z} \models \forall (\chi \rightarrow \psi)$.

**Theorem 6.39 (Weakest Liberal Precondition)**

The weakest liberal precondition of a program $P$ and postcondition $\phi$, denoted by $wp(P, \phi)$, is the weakest formula $\psi$ such that $\models^p \{ \psi \} P \{ \phi \}$. Such a formula exists and can be constructed as a formula in our language.

The reason for the last theorem is that the model $\mathbb{Z}$ with $+, \cdot, <$ is powerful enough to express all notions we need.
6.5 Proof Calculi: Total Correctness
Proof Rules

We extend the proof calculus for partial correctness presented to one that covers total correctness.

Partial correctness does not say anything about the termination of programs. It is easy to see that only the while construct can cause the nontermination of a program. Hence, in addition to the partial correctness calculus we have to prove that a while loop terminates.
The proof of termination of while statements follows the following schema:

- Given a program $P$, identify an integer expression $I$ whose value decreases after performing $P$ but which is always non-negative.

Such an expression $E$ is called a variant. Now, it is easy to see that a while loop has to terminate if such a variant exists. The corresponding rule is given below.
Definition 6.40 (Total-While Rule)

\[
\{ \phi \land B \land 0 \leq E = E_0 \} \mathcal{C} \{ \phi \land 0 \leq E < E_0 \}
\]

\[ \{ \phi \land 0 \leq E \} \text{ while } B \{ \mathcal{C} \{ \phi \land \neg B \} \]

where

- \( \phi \) is an invariant of the while-loop,
- \( E \) is a variant of the while-loop,
- \( E_0 \) represents the initial value of \( E \) before the loop.
Example 6.41

Let us consider the program Fac once more:

\[
\begin{align*}
y &:= 1; \\
z &:= 0; \\
\textbf{while } (z \neq x) \{z := z + 1; \; y := y \cdot z\}
\end{align*}
\]

How does a variant for the proof of \{x \geq 0\}Fac\{y = x!\} look?

A possible variant is for instance \(x - z\). The first time the loop is entered we have that \(x - z = x\) and then the expression decreases step by step until \(x - z = 0\).
Annotation Calculus

Definition 6.42 (Total-While Rule)

\[
\text{while } B \{P\} \quad \frac{\{\phi \land 0 \leq E\} \quad \text{while } B \{\{\phi \land 0 \leq E = E_0\} \quad \text{P}\{\phi \land 0 \leq E < E_0\}\} \quad \{\phi \land \neg B\}}
\]

where

- \(\phi\) is an \textbf{invariant} of the while-loop,
- \(E\) is a \textbf{variant} of the while-looping,
- \(E_0\) represents the \textbf{initial value} of \(E\) before the loop.
Example 6.43

Prove that $\models^t \{ x \geq 0 \} \text{Fac}\{ y = x! \}$. What do we have to do at the beginning?

We have to determine a suitable **variant** and an **invariant**. As variant we may choose $x - z$ and as invariant $y = z!$. Now we can apply the annotation calculus.
\{ x \geq 0 \}
\{ 1 = 0! \land 0 \leq x - 0 \}
y := 1;
\{ y = 0! \land 0 \leq x - 0 \}
z := 0;
\{ y = z! \land 0 \leq x - z \}
while x \neq z {
    \{ y = z! \land x \neq z \land 0 \leq x - z = E_0 \}
    \{ y(z + 1) = (z + 1)! \land 0 \leq x - (z + 1) < E_0 \}
    z := z + 1;
    \{ yz = z! \land 0 \leq x - z < E_0 \}
    y := y \cdot z
    \{ y = z! \land 0 \leq x - z < E_0 \}
}\{ y = z! \land x = z \}
\{ y = x! \}
Weakest Precondition

Proposition 6.44 (**wp**- Total Correctness)

The **weakest precondition** for total correctness exists and can be expressed as a formula in our language.

Example 6.45

What is the weakest precondition for

```plaintext
while i < n {i := i + 1}\{i = n + 5\}?
```

\(i = n + 5\).
6.6 Sound and Completeness
Theoretical Aspects

Finally, we consider some theoretical aspects with respect to the sound and completeness of the introduced calculi.

Recall, that a calculus is **sound** if everything that can be derived is also semantically true:

\[
\text{If } \vdash^x \{ \phi \} P \{ \psi \} \text{ then } \models^x \{ \phi \} P \{ \psi \}
\]

where \( x \in \{ p, t \} \) stands for partial and total correctness.
The reverse direction is referred to as completeness.
A calculus is complete if everything that can semantically be derived is also derivable by the calculus:

$$\text{If } \models^x \{\phi\} P \{\psi\} \text{ then } \vdash^x \{\phi\} P \{\psi\}$$

where $x \in \{p, t\}$ stands for partial and total correctness.

Which direction is more difficult to prove?
For the soundness we just have to make sure that all proof rules introduced make sense. That is, given a rule \( \frac{X}{Y} \) it has to be shown that \( Y \) holds whenever \( X \) holds.

**Theorem 6.46**

*The Hoare calculus is sound.*

**Proof.**

Exercise!
Theorem 6.47

The Hoare calculus is complete.

- The Hoare calculus contains axioms and rules that require to determine whether certain formulae (proof obligations) are true in \( \mathbb{Z} \).
- The theory of \( \mathbb{Z} \) is undecidable.
- Thus any re axiomatization of \( \mathbb{Z} \) is incomplete.
- However, most proof obligations occurring in practice can be checked by theorem provers.
7. Knowledge Engineering: FOL

- First Order Logic
- Sit-Calculus
- The Blocksworld
- Higher order logic
Content of this chapter:

**FOL:** While sentential logic SL has some nice features, it is quite restricted in expressivity. *First order logic (FOL)* is an extension of SL which allows us to express much more in a succinct way.

**SIT-calculus:** The *Situation Calculus*, introduced by John McCarthy, is a special method for using FOL to express the *dynamics* of an agent. Applying actions to the world leads to a series of successor worlds that can be represented with special FOL *terms.*
Content of this chapter (2):

**Blocksworld:** We consider the blocksworld scenario and discuss how to formalize that with FOL.

**HOL:** Finally, we give an outlook to higher order logics (HOL), in particular to second order logic. While we can express much more than in 1st order logic, the price we have to pay is that there is no correct and complete calculus.

**Declarative:** In this chapter we only consider the question How to use FOL to model the world? We are not concerned with deriving new information or with implementing FOL. This will be done in the next chapter.
7.1 First Order Logic
Definition 7.1 (First order logic $\mathcal{L}_{FOL}$, $\mathcal{L} \subseteq \mathcal{L}_{FOL}$)

The language $\mathcal{L}_{FOL}$ of first order logic (Prädikatenlogik erster Stufe) is:

- $x, y, z, x_1, x_2, \ldots, x_n, \ldots$: a countable set $\text{Var}$ of variables
- for each $k \in \mathbb{N}_0$: $P^k_1, P^k_2, \ldots, P^k_n, \ldots$ a countable set $\text{Pred}^k$ of $k$-dimensional predicate symbols (the $0$-dimensional predicate symbols are the propositional logic constants from $\mathcal{A}t$ of $\mathcal{L}_{SL}$, including $\square, \top$).
- for each $k \in \mathbb{N}_0$: $f^k_1, f^k_2, \ldots, f^k_n, \ldots$ a countable set $\text{Funct}^k$ of $k$-dimensional function symbols.
- $\neg, \land, \lor, \rightarrow$: the sentential connectives.
- $\langle, \rangle$: the parentheses.
- $\forall, \exists$: the quantifiers.
Definition (continued)

The 0-dimensional function symbols are called **individuum constants** — we leave out the parentheses. In general we will need — as in propositional logic — only a certain subset of the predicate or function symbols.

These define a language \( \mathcal{L} \subseteq \mathcal{L}_{FOL} \) (analogously to definition 5.1 on page 2). The used set of predicate and function symbols is also called **signature** \( \Sigma \).
The concept of an \( \mathcal{L} \)-term \( t \) and an \( \mathcal{L} \)-formula \( \varphi \) are defined inductively:

**Definition**

**Term:** \( \mathcal{L} \)-terms \( t \) are defined as follows:

1. each variable is a \( \mathcal{L} \)-term.
2. if \( f^k \) is a \( k \)-dimensional function symbol from \( \mathcal{L} \) and \( t_1, \ldots, t_k \) are \( \mathcal{L} \)-terms, then \( f^k(t_1, \ldots, t_k) \) is a \( \mathcal{L} \)-term.

The set of all \( \mathcal{L} \)-terms that one can create from the set \( X \subseteq \text{Var} \) is called \( \text{Term}_\mathcal{L}(X) \) or \( \text{Term}_\Sigma(X) \). Using \( X = \emptyset \) we get the set of basic terms \( \text{Term}_\mathcal{L}(\emptyset) \), short: \( \text{Term}_\mathcal{L} \).
Definition (continued)

**Formula**: $\mathcal{L}$-formulae $\varphi$ are also defined inductively:

1. If $P^k$ is a $k$-dimensional predicate symbol from $\mathcal{L}$ and $t_1, \ldots, t_k$ are $\mathcal{L}$-terms, then $P^k(t_1, \ldots, t_k)$ is a $\mathcal{L}$-formula.

2. For all $\mathcal{L}$-formulae $\varphi$, $(\neg \varphi)$ is a $\mathcal{L}$-formula.

3. For all $\mathcal{L}$-formulae $\varphi$ and $\psi$, $(\varphi \land \psi)$ and $(\varphi \lor \psi)$ are $\mathcal{L}$-formulae.

4. If $x$ is a variable and $\varphi$ is a $\mathcal{L}$-formula, then $(\exists x \varphi)$ and $(\forall x \varphi)$ are $\mathcal{L}$-formulae.
**Definition (continued)**

**Atomic $\mathcal{L}$-formulae** are those which are composed according to 1., we call them $At_{\mathcal{L}}(X)$ ($X \subseteq \text{Var}$). The set of all $\mathcal{L}$-formulae in respect to $X$ is called $Fml_{\mathcal{L}}(X)$.

**Positive formulae** ($Fml^+_\mathcal{L}(X)$) are those which are composed using only 1, 3. and 4.

If $\varphi$ is a $\mathcal{L}$-formula and is part of an other $\mathcal{L}$-formula $\psi$ then $\varphi$ is called **sub-formula** of $\psi$. 
Example 7.2 (From semigroups to rings)

We consider \( \mathcal{L} = \{0, 1, +, \cdot, \leq, =\} \), where 0, 1 are constants, +, · binary operations and \( \leq, = \) binary relations. What can be expressed in this language?

Ax 1: \( \forall x \forall y \forall z \quad x + (y + z) = (x + y) + z \)
Ax 2: \( \forall x \quad (x + 0 = 0 + x) \land (0 + x = x) \)
Ax 3: \( \forall x \exists y \quad (x + y = 0) \land (y + x = 0) \)
Ax 4: \( \forall x \forall y \quad x + y = y + x \)
Ax 5: \( \forall x \forall y \forall z \quad x \cdot (y \cdot z) = (x \cdot y) \cdot z \)
Ax 6: \( \forall x \forall y \forall z \quad x \cdot (y + z) = x \cdot y + x \cdot z \)
Ax 7: \( \forall x \forall y \forall z \quad (y + z) \cdot x = y \cdot x + z \cdot x \)

Axiom 1 describes an semigroup, the axioms 1-2 describe a monoid, the axioms 1-3 a group, and the axioms 1-7 a ring.
Definition 7.3 ($\mathcal{L}$-structure $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$)

A $\mathcal{L}$-structure or a $\mathcal{L}$-interpretation is a pair $\mathcal{A} = \text{def} (U_\mathcal{A}, I_\mathcal{A})$ with $U_\mathcal{A}$ being an arbitrary non-empty set, which is called the basic set (the universe or the individuum range) of $\mathcal{A}$. Further $I_\mathcal{A}$ is a mapping which

- assigns to each $k$-dimensional predicate symbol $P^k$ in $\mathcal{L}$ a $k$-dimensional predicate over $U_\mathcal{A}$
- assigns to each $k$-dimensional function symbol $f^k$ in $\mathcal{L}$ a $k$-dimensional function on $U_\mathcal{A}$

In other words: the domain of $I_\mathcal{A}$ is exactly the set of predicate and function symbols of $\mathcal{L}$. 

Prof. Dr. Jürgen Dix
Clausthal, SS 2011
**Definition (continued)**

The range of $I_A$ consists of the predicates and functions on $U_A$. We write:

$$I_A(P) = P^A, \ I_A(f) = f^A.$$  

Let $\varphi$ be a $\mathcal{L}_1$-formula and $A = \text{def } (U_A, I_A)$ a $\mathcal{L}$-structure. $A$ is called **matching with** $\varphi$ if $I_A$ is defined for all predicate and function symbols which appear in $\varphi$, i.e. if $\mathcal{L}_1 \subseteq \mathcal{L}$.

**FOL with Equality**

Often, one assumes that the predicate symbol $\equiv$ is **built-in** and interpreted by identity in all structures $A$:

$$I_A(\equiv) = \{(x, x) : x \in A\}.$$ In that case, $\equiv$ is not listed among the predicates in a model $A$. 

Definition 7.4 (Variable assignment $\varrho$)

A variable assignment $\varrho$ over a $\mathcal{L}$-structure $\mathcal{A} = (U_\mathcal{A}, I_\mathcal{A})$ is a function

$$\varrho : \text{Var} \rightarrow U_\mathcal{A}; \quad x \mapsto \varrho(x).$$

Note that this is exactly what we called state in the chapter about the Hoare calculus.
Some structures

The following will be explained in detail on the blackboard.

We consider $\mathcal{L} = \{\leq, \div\}$ and the following structures:

1. $(\mathbb{Z}, \leq^\mathbb{Z})$
2. $(\mathbb{R}, \leq^\mathbb{R})$
3. $(\mathbb{Q}, \leq^\mathbb{Q})$
4. $(\mathbb{Q}_0^+, \leq^\mathbb{Q}_0^+)$

State formulae in $\mathcal{L}$ that are true in all structures, just in some, .... Is there a formula $\phi$ which can distinguish between $(\mathbb{R}, \leq^\mathbb{R})$ and $(\mathbb{Q}, \leq^\mathbb{Q})$?
Some structures (2)

We have seen some formulae that are true in these structures. Can we come up with a finite axiomatization?

Dense linear order without endpoints

It turns out, that the set of all formulae true in \((\mathbb{R}, \leq_{\mathbb{R}})\) coincides with the set all formulae true in \((\mathbb{Q}, \leq_{\mathbb{Q}})\). An axiomatization is given by the finite set of formulae stating that \(<\) is a dense, linear order without endpoints. This theory is also complete.
Definition 7.5 (Semantics of first order logic, Model $\mathcal{A}$)

Let $\varphi$ be a formula, $\mathcal{A}$ a structure matching with $\varphi$ and $\varrho$ a variable assignment over $\mathcal{A}$. For each term $t$, which can be built from components of $\varphi$, we define the value of $t$ in the structure $\mathcal{A}$, called $\mathcal{A}(t)$.

1. For a variable $x$ is $\mathcal{A}(x) = \text{def} \varrho(x)$.

2. If $t$ has the form $t = f^k(t_1, \ldots, t_k)$, with $t_1, \ldots, t_k$ being terms and $f^k$ a $k$-dimensional function symbol, then $\mathcal{A}(t) = \text{def} f^\mathcal{A}(\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k))$.

Compare with Definition 6.10.
We define inductively the logical value of a formula $\varphi$ in $\mathcal{A}$:

1. if $\varphi = \text{def} \ P^k(t_1, \ldots, t_k)$ with the terms $t_1, \ldots, t_k$ and the $k$-dimensional predicate symbol $P^k$, then
   
   $$\mathcal{A}(\varphi) = \text{def} \begin{cases} \top, & \text{if } (\mathcal{A}(t_1), \ldots, \mathcal{A}(t_k)) \in \mathcal{P}, \\ \square, & \text{else.} \end{cases}$$

2. if $\varphi = \text{def} \ \neg \psi$, then
   
   $$\mathcal{A}(\varphi) = \text{def} \begin{cases} \top, & \text{if } \mathcal{A}(\psi) = \square, \\ \square, & \text{else.} \end{cases}$$

3. if $\varphi = \text{def} \ (\psi \land \eta)$, then
   
   $$\mathcal{A}(\varphi) = \text{def} \begin{cases} \top, & \text{if } \mathcal{A}(\psi) = \top \text{ and } \mathcal{A}(\eta) = \top, \\ \square, & \text{else.} \end{cases}$$
Definition (continued)

4. if $\varphi = \text{def} (\psi \lor \eta)$, then

$$A(\varphi) = \text{def} \begin{cases} \top, & \text{if } A(\psi) = \top \text{ or } A(\eta) = \top, \\ \square, & \text{else}. \end{cases}$$

5. if $\varphi = \text{def} \forall x \psi$, then

$$A(\varphi) = \text{def} \begin{cases} \top, & \text{if } \forall d \in U_A : A[x/d](\psi) = \top, \\ \square, & \text{else}. \end{cases}$$

6. if $\varphi = \text{def} \exists x \psi$, then

$$A(\varphi) = \text{def} \begin{cases} \top, & \text{if } \exists d \in U_A : A[x/d](\psi) = \top, \\ \square, & \text{else}. \end{cases}$$

For $d \in U_A$ let $A[x/d]$ be the structure $A'$, identical to $A$ except for the definition of $x'A' : x'A' = \text{def} d$ (whether $I_A$ is defined for $x$ or not).
Definition (continued)

We write:

- \( A \models \varphi[\varrho] \) for \( A(\varphi) = \top \): \( A \) is a **model** for \( \varphi \) with respect to \( \varrho \).
- If \( \varphi \) does not contain free variables, then \( A \models \varphi[\varrho] \) is independent from \( \varrho \). We simply leave out \( \varrho \).
- If there is at least one model for \( \varphi \), then \( \varphi \) is called **satisfiable** or **consistent**.

A **free variable** is a variable which is not in the scope of a quantifier. For instance, \( z \) is a free variable of \( \forall x P(x, z) \) but not free (or bounded) in \( \forall z \exists x P(x, z) \).

A variable can occur free and bound in the same formula. So we have to talk about a particular **occurrence** of a variable: the very position of it in the formula.
Definition 7.6 (Tautology)

1. A theory is a set of formulae without free variables: $T \subseteq \text{Fml}_\mathcal{L}$. The structure $\mathcal{A}$ satisfies $T$ if $\mathcal{A} \models \varphi$ holds for all $\varphi \in T$. We write $\mathcal{A} \models T$ and call $\mathcal{A}$ a model of $T$.

2. A $\mathcal{L}$-formula $\varphi$ is called $\mathcal{L}$-tautology, if for all matching $\mathcal{L}$-structures $\mathcal{A}$ the following holds: $\mathcal{A} \models \varphi$.

From now on we suppress the language $\mathcal{L}$, because it is obvious from context.
Definition 7.7 (Consequence set $Cn(T)$)

A formula $\varphi$ follows semantically from $T$, if for all structures $A$ with $A \models T$ also $A \models \varphi$ holds.

We write: $T \models \varphi$.

In other words: all models of $T$ do also satisfy $\varphi$.

We denote by $Cn_L(T) = \text{def} \{ \varphi \in Fml_L : T \models \varphi \}$, or simply $Cn(T)$, the semantic consequence operator.
Lemma 7.8 (Properties of $C_n(T)$)

The semantic consequence operator $C_n$ has the following properties:

1. **$T$-extension**: $T \subseteq C_n(T)$,
2. **Monotony**: $T \subseteq T' \Rightarrow C_n(T) \subseteq C_n(T')$,
3. **Closure**: $C_n(C_n(T)) = C_n(T)$.

Lemma 7.9 ($\varphi \notin C_n(T)$)

$\varphi \notin C_n(T)$ if and only if there is a structure $\mathcal{A}$ with $\mathcal{A} \models T$ and $\mathcal{A} \models \neg \varphi$.

Or: $\varphi \notin C_n(T)$ iff there is a counterexample: a model of $T$ in which $\varphi$ is not true.
Definition 7.10 \((\text{MOD}(T), \text{Cn}(U))\)

If \(T \subseteq Fml_{\mathcal{L}}\), then we denote by \(\text{MOD}(T)\) the set of all \(\mathcal{L}\)-structures \(A\) which are models of \(T\):

\[
\text{MOD}(T) = \text{def} \ \{A : A \models T\}.
\]

If \(U\) is a set of structures then we can consider all sentences, which are true in all structures. We call this set also \(\text{Cn}(U)\):

\[
\text{Cn}(U) = \text{def} \ \{\varphi \in Fml_{\mathcal{L}} : \forall A \in U : A \models \varphi\}.
\]

\(\text{MOD}\) is obviously dual to \(\text{Cn}\):

\[
\text{Cn}(\text{MOD}(T)) = \text{Cn}(T), \quad \text{MOD}(\text{Cn}(T)) = \text{MOD}(T).
\]
Definition 7.11 (Completeness of a theory $T$)

$T$ is called **complete**, if for each formula $\varphi \in Fml_L$: $T \models \varphi$ or $T \models \neg \varphi$ holds.

Attention:

Do not mix up this last condition with the property of a structure $\nu$ (or a model): **each structure is complete in the above sense**.

Lemma 7.12 (Ex Falso Quodlibet)

$T$ is consistent if and only if $\text{Cn}(T) \neq Fml_L$. 

Prof. Dr. Jürgen Dix
An illustrating example

Example 7.13 (Natural numbers in different languages)

- $\mathcal{N}_{Pr} = (\mathbb{N}_0, 0^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, =^{\mathcal{N}})$ ("Presburger Arithmetik"),
- $\mathcal{N}_{PA} = (\mathbb{N}_0, 0^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, =^{\mathcal{N}})$ ("Peano Arithmetik"),
- $\mathcal{N}_{PA'} = (\mathbb{N}_0, 0^{\mathcal{N}}, 1^{\mathcal{N}}, +^{\mathcal{N}}, \cdot^{\mathcal{N}}, =^{\mathcal{N}})$ (variant of $\mathcal{N}_{PA}$).

These sets each define the same natural numbers, but in different languages.

Question:

If the language is bigger then we might express more. Is $\mathcal{L}_{PA'}$ strictly more expressive than $\mathcal{L}_{PA}$?
Answer:

No, because one can replace the $1^N$ by a $\mathcal{L}_{PA}$-formula: there is a $\mathcal{L}_{PA}$-formula $\phi(x)$ so that for each variable assignment $\rho$ the following holds:

$$\mathcal{N}_{PA'} \models_{\rho} \phi(x) \text{ if and only if } \rho(x) = 1^N$$

- Thus we can define a macro for $1$.
- Each formula of $\mathcal{L}_{PA'}$ can be transformed into an equivalent formula of $\mathcal{L}_{PA}$.
Question:
Is $\mathcal{L}_{PA}$ perhaps more expressive than $\mathcal{L}_{Pr}$, or can the multiplication be defined somehow?

Indeed, $\mathcal{L}_{PA}$ is more expressive:

- the set of sentences valid in $\mathcal{N}_{Pr}$ is \textit{decidable}, whereas
- the set of sentences valid in $\mathcal{N}_{PA}$ is \textit{not even recursively enumerable}. 
Question:

We have introduced $\mathbb{Z} = (\mathbb{Z}, 0^\mathbb{Z}, 1^\mathbb{Z}, +^\mathbb{Z}, -^\mathbb{Z}, \cdot^\mathbb{Z}, <^\mathbb{Z})$ in the chapter about the Hoare calculus. How does it compare with $\mathcal{N}_{PA}$?

- “$\doteq$” can be defined with $<$ and vice versa in $\mathbb{Z}$ and $\mathcal{N}_{PA}$ (resp. in $\mathcal{N}_{Pr}$).
- “$-$” can also be defined with the other constructs.
- $\mathcal{N}_{PA}$ can be defined in $\mathbb{Z}$ with an appropriate formula $\phi(x)$:

$$\mathbb{Z} \models_{\rho} \phi(x) \text{ if and only if } \rho(x) \in \mathbb{N}_0$$
Can $\mathcal{N}_{PA}$ be defined in $(\mathbb{Z}, +^\mathbb{Z}, \cdot^\mathbb{Z})$?

To be more precise, for each $\mathcal{L}_{PA}$ formula $\phi$, there is a $\mathcal{L}_{\mathbb{Z}}$ formula $\phi'$ such that: if $\mathcal{N}_{PA} \models \phi$ then $\mathbb{Z} \models \phi'$.

So $\mathbb{Z}$ is at least as difficult as $\mathcal{N}_{PA}$.

The converse is true as well. Therefore although the theories of $\mathbb{Z}$ and $\mathcal{N}_{PA}$ are not identical, the truth of a formula in one of them can be reduced to the truth of a translated formula in the other one.
Question:

What about \( \mathcal{R} = (\mathbb{R}, 0^\mathcal{R}, 1^\mathcal{R}, +^\mathcal{R}, -^\mathcal{R}, \cdot^\mathcal{R}, <^\mathcal{R}) \) and \( \mathcal{Q} = (\mathbb{Q}, 0^\mathcal{Q}, 1^\mathcal{Q}, +^\mathcal{Q}, -^\mathcal{Q}, \cdot^\mathcal{Q}, <^\mathcal{Q}) \)? How do they compare with \( \mathcal{N}_{PA} \)?

- State a formula that distinguishes both structures.
- Can one define \( \mathcal{Q} \) within \( \mathcal{R} \) (as we did define \( \mathcal{N}_{PA} \) in \( \mathcal{E} \))?  
- Is there an axiomatization of \( \mathcal{R} \)?

In general, theories of particular structures are undecidable. But it depends on the underlying language.
As for sentential logic, formulae can be derived from a given theory and they can also (semantically) follow from it.

**Syntactic derivability** $\vdash$: the notion that certain formulae can be derived from other formulae using a certain calculus,

**Semantic validity** $\models$: the notion that certain formulae follow from other formulae based on the semantic notion of a model.
Definition 7.14 (Correct-Completeness for a calculus)

Given an arbitrary calculus (which defines a notion $\vdash$) and a semantics based on certain models (which defines a relation $|=\$), we say that

**Correctness:** The calculus is *correct†* with respect to the semantics, if the following holds:

$$\Phi \vdash \phi \text{ implies } \Phi |= \phi.$$ 

**Completeness:** The calculus is *complete* with respect to the semantics, if the following holds:

$$\Phi |= \phi \text{ implies } \Phi \vdash \phi.$$
We have already defined a complete and correct calculus for sentential logic $L_{SL}$. Such calculi also exist for first order logic $L_{FOL}$.

**Theorem 7.15 (Correct-, Completeness of FOL)**

A formula follows semantically from a theory $T$ if and only if it can be derived:

$$T \vdash \varphi \text{ if and only if } T \models \varphi$$

**Theorem 7.16 (Compactness of FOL)**

A formula follows semantically from a theory $T$ if and only if it follows semantically from a finite subset of $T$:

$$Cn(T) = \bigcup \{Cn(T') : T' \subseteq T, T' \text{ finite}\}.$$
Predicate- or function- symbols?

How to formalize each animal has a brain?

1. Two unary predicate symbols: \( \text{animal}(x) \), \( \text{has\_brain}(x) \). The statement becomes

\[
\forall x \ ( \text{animal}(x) \rightarrow \text{has\_brain}(x) )
\]

2. Finally, what about a binary predicate \( \text{is\_brain\_of}(y, x) \) and the statement

\[
\forall x \ ( \text{animal}(x) \rightarrow \exists y \ \text{is\_brain\_of}(y, x) )
\]

3. But why not introducing a unary function symbol \( \text{brain\_of}(x) \) denoting \( x \)'s brain? Then

\[
\forall x \ \exists y \ ( \text{animal}(x) \rightarrow y = \text{brain\_of}(x) )
\]
Unary function or binary predicate?

Given a unary function symbol $f(x)$ and a constant $c$. Then the only terms that can be built are of the form $f^n(c)$. Assume we have $\equiv$ as the only predicate. Then the atomic formulae that we can built have the form $f^n(c) \equiv f^m(c)$. We call this language $\mathcal{L}_1$. 
Unary function or binary predicate? (2)

Assume now that instead of $f$ and $\dot{=} =$, we use a binary predicate $p_f(y, x)$ which formalizes $y \dot{=} f(x)$. We call this language $L_2$.

- Can we **express** all formulae of $L_1$ in $L_2$?
- And vice versa?
- **What is the difference between both approaches?**
Tuna the cat

We formalize again the example of the killed cat, this time in FOL.

1. Jack owns a dog.
   \[ \rightarrow \text{owns}(x, y), \text{dog}(x), \text{jack}. \]

2. Dog owners are animal lovers.
   \[ \rightarrow \text{animal	extunderscore lover}(x). \]

3. Animal lovers do not kill animals.
   \[ \rightarrow \text{killer	extunderscore of}(x, y), \text{animal}(x). \]

4. Either Jack or Bill killed Tuna, the cat.
   \[ \rightarrow \text{bill}, \text{tuna}, \text{cat}(x), \text{killer	extunderscore of}(x, y). \]

The formalization follows on the blackboard.
7.2 Sit-Calculus
function KB-AGENT(percept) returns an action
    static: KB, a knowledge base
            t, a counter, initially 0, indicating time

    TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
    action ← ASK(KB, MAKE-ACTION-QUERY(t))
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t ← t + 1

return action
Idea:

In order to describe actions or their effects consistently we consider the world as a sequence of situations (snapshots of the world). Therefore we have to extend each predicate by an additional argument.

We use the function symbol

\[ \text{result}(\text{action}, \text{situation}) \]

as the term for the situation which emerges when the action \text{action} is executed in the situation \text{situation}.

Actions: \text{Turn\_right}, \text{Turn\_left}, \text{Foreward}, \text{Shoot}, \text{Grab}, \text{Release}, \text{Climb}. 
7 Knowledge Engineering: FOL
7.2 Site-Logic

Forwards $S_0$

Turn (Right) $S_1$

Forward $S_3$

$S_0 \rightarrow$ Forward

$S_1 \rightarrow$ Turn (Right)

$S_2 \rightarrow$ PIT

$S_3 \rightarrow$ PIT
We also need a **memory**, a predicate

\[
\text{At}(\text{person}, \text{location}, \text{situation})
\]

with \textit{person} being either \textit{Wumpus} or \textit{Agent} and \textit{location} being the actual position (stored as pair \([i,j]\)).

Important axioms are the so called **successor-state axioms**, they describe how actions effect situations. The most general form of these axioms is

true afterwards \iff an action made it true or it is already true and no action made it false
Axioms about $At(p, l, s)$:

$$At(p, l, \text{result}(\text{Forward}, s)) \leftrightarrow ((l \doteq \text{location\_ahead}(p, s) \land \neg \text{Wall}(l))$$

$$At(p, l, s) \rightarrow \text{Location\_ahead}(p, s) \doteq \text{Location\_toward}(l, \text{Orient.}(p, s))$$

$$\text{Wall}([x, y]) \leftrightarrow (x \doteq 0 \lor x \doteq 5 \lor y \doteq 0 \lor y \doteq 5)$$
\[ Location\_toward([x, y], 0) \doteq [x + 1, y] \]
\[ Location\_toward([x, y], 90) \doteq [x, y + 1] \]
\[ Location\_toward([x, y], 180) \doteq [x - 1, y] \]
\[ Location\_toward([x, y], 270) \doteq [x, y - 1] \]

\begin{align*}
Orient.(\text{Agent}, s_0) & \doteq 90 \\
Orient.(p, \textbf{result}(a, s)) & \doteq d \iff \\
& (a \doteq \text{turn\_right} \land d \doteq \text{mod}(Orient.(p, s) - 90, 360)) \\
& \lor (a \doteq \text{turn\_left} \land d \doteq \text{mod}(Orient.(p, s) + 90, 360)) \\
& \lor (Orient.(p, s) \doteq d \land \neg (a \doteq \text{Turn\_right} \lor a \doteq \text{Turn\_left}))
\end{align*}

\textit{mod}(x, y) \text{ is the implemented “modulo”-function, } \\
\text{assigning a value between 0 and } y \text{ to each } \\
\text{variable } x.
Axioms about percepts, useful new notions:

\[ \text{Percept}([\text{Stench}, b, g, u, c], s) \rightarrow \text{Stench}(s) \]
\[ \text{Percept}([a, \text{Breeze}, g, u, c], s) \rightarrow \text{Breeze}(s) \]
\[ \text{Percept}([a, b, \text{Glitter}, u, c], s) \rightarrow \text{At\_Gold}(s) \]
\[ \text{Percept}([a, b, g, \text{Bump}, c], s) \rightarrow \text{At\_Wall}(s) \]
\[ \text{Percept}([a, b, g, u, \text{Scream}], s) \rightarrow \text{Wumpus\_dead}(s) \]

\[ \text{At}(\text{Agent}, l, s) \land \text{Breeze}(s) \rightarrow \text{Breezy}(l) \]
\[ \text{At}(\text{Agent}, l, s) \land \text{Stench}(s) \rightarrow \text{Smelly}(l) \]
\[ \text{Adjacent}(l_1, l_2) \leftrightarrow \exists d \; l_1 = \text{Location\_toward}(l_2, d) \]
\[ \text{Smelly}(l_1) \rightarrow \exists l_2 \; \text{At}(Wumpus, l_2, s) \wedge (l_2 = l_1 \lor \text{Adjacent}(l_1, l_2)) \]

\[ \text{Percept}([\text{none, none, g, u, c}], s) \wedge \text{At}(\text{Agent}, x, s) \wedge \text{Adjacent}(x, y) \rightarrow \text{OK}(y) \]
\[ (\neg \text{At}(Wumpus, x, t) \wedge \neg \text{Pit}(x)) \rightarrow \text{OK}(y) \]
\[ \text{At}(Wumpus, l_1, s) \wedge \text{Adjacent}(l_1, l_2) \rightarrow \text{Smelly}(l_2) \]
\[ \text{At}(\text{Pit}, l_1, s) \wedge \text{Adjacent}(l_1, l_2) \rightarrow \text{Breezy}(l_2) \]
Axioms to describe actions:

\[
\begin{align*}
\text{Holding}(\text{Gold}, \text{result}(\text{Grab}, s)) & \iff \text{At}_{-\text{Gold}}(s) \lor \text{Holding}(\text{Gold}, s) \\
\text{Holding}(\text{Gold}, \text{result}(\text{Release}, s)) & \iff \square \\
\text{Holding}(\text{Gold}, \text{result}(\text{Turn}\_\text{right}, s)) & \iff \text{Holding}(\text{Gold}, s) \\
\text{Holding}(\text{Gold}, \text{result}(\text{Turn}\_\text{left}, s)) & \iff \text{Holding}(\text{Gold}, s) \\
\text{Holding}(\text{Gold}, \text{result}(\text{Forward}, s)) & \iff \text{Holding}(\text{Gold}, s) \\
\text{Holding}(\text{Gold}, \text{result}(\text{Climb}, s)) & \iff \text{Holding}(\text{Gold}, s)
\end{align*}
\]

Each effect must be described carefully.
Axioms describing preferences among actions:

\[ \text{Great}(a, s) \rightarrow \text{Action}(a, s) \]
\[ (\text{Good}(a, s) \land \neg \exists b \text{Great}(b, s)) \rightarrow \text{Action}(a, s) \]
\[ (\text{Medium}(a, s) \land \neg \exists b (\text{Great}(b, s) \lor \text{Good}(b, s))) \rightarrow \text{Action}(a, s) \]
\[ (\text{Risky}(a, s) \land \neg \exists b (\text{Great}(b, s) \lor \text{Good}(b, s) \lor \text{Medium}(a, s))) \rightarrow \text{Action}(a, s) \]

\[ \text{At}(\text{Agent}, [1, 1], s) \land \text{Holding}(\text{Gold}, s) \rightarrow \text{Great}(\text{Climb}, s) \]
\[ \text{At}_{\text{Gold}}(s) \land \neg \text{Holding}(\text{Gold}, s) \rightarrow \text{Great}(\text{Grab}, s) \]
\[ \text{At}(\text{Agent}, l, s) \land \neg \text{Visited}(\text{Location}_\text{ahead}(\text{Agent}, s)) \land \neg \text{OK}(\text{Location}_\text{ahead}(\text{Agent}, s)) \rightarrow \text{Good}(\text{Forward}, s) \]
\[ \text{Visited}(l) \iff \exists s \text{At}(\text{Agent}, l, s) \]

The goal is not only to find the gold but also to return safely. We need additional axioms like

\[ \text{Holding}(\text{Gold}, s) \rightarrow \text{Go\_back}(s). \]
7.3 The Blocksworld
The Blocks World (BW) is one of the most popular domains in AI (first used in the 1970s). However, the setting is easy:
Domain

Blocks of various shapes, sizes and colors sitting on a table and on each other.

(Here: Quadratic blocks of equal size.)

Actions

Pick up a block and put it to another position (tower of blocks or table). Only the topmost blocks can be used.

How to formalize this? What language to use?
Choosing the predicates: First try.

1. One action: move. We add a ternary predicate $move(b, x, y)$: The block $b$ is moved from $x$ to $y$ if both $b$ and $y$ are clear (nothing on top of them).

2. So we need a predicate $clear(x)$.

3. One predicate $on(b, x)$ meaning block $b$ is on $x$.

4. But then $clear(x)$ can be defined by $\forall y \neg on(y, x)$.

Problem: What about the table?

If we view the table as a simple block: $clear(table)$ means that there is nothing on the table.
Choosing the predicates: Second try.

Keep the current framework, view the table as a block but add an additional binary predicate

\[ move\_to\_table(b, x), \]

meaning that the block \( b \) is moved from \( x \) to the table.

\( \text{clear}(x) \) is interpreted as \textbf{there is free place on} \( x \) \textbf{to put a block on} \( x \). This interpretation does also work for \( x \models \text{table} \).
Choosing the predicates: Third try.

**Definition 7.17 ($\mathcal{L}_{BW}$)**

The BW language $\mathcal{L}_{BW} \subseteq \mathcal{L}_{FOL}$ is the subset of FOL having a signature consisting of the two *binary* predicate symbols *above* and $\doteq$.

We define the following macros:

- $on(x, y) : above(x, y) \land \neg(\exists z(above(x, z) \land above(z, y)))$
- $onTable(x) : \neg(\exists y(above(x, y)))$
- $clear(x) : \neg(\exists y(above(y, x)))$

How do the $\mathcal{L}_{BW}$-structures look like?

Do they all make sense?
Our blocksworld has a very specific structure, which should be reflected in the models!

**Definition 7.18 (Chain)**

Let $A$ be a nonempty set. We say that $(A, <)$ is a chain if $<$ is a binary relation on $A$ which is

1. **irreflexive**,  
2. **transitive**, and  
3. **connected**, (i.e., for all $a, b \in A$ it holds that either $a < b$, $a = b$, or $a > b$).

(A chain is interpreted as a *tower of blocks*.)
Definition 7.19 (Ł_{BW}-BW-structure)

An **BW-structure** is a Ł_{BW}-structure \( A = (U, I) \) in which \( I(\text{above}) \) is a finite and disjoint union of chains over \( U \), i.e.

\[
I(\text{above}) = \bigcup_{(A, <) \in A'} \{ (a, b) \in A^2 \mid a > b \}
\]

where \( A' \) is a set of chains over \( U \) such that for all \( (A_1, <_1), (A_2, <_2) \in A' \) with \( (A_1, <_1) \neq (A_2, <_2) \) it holds that \( A_1 \cap A_2 = \emptyset \).

(Note: \( I(\text{above}) \) is transitive!)
Definition 7.20 (BW-theory)

The theory $Th(BW)$ consists of all $L_{BW}$-sentences which are true in all BW-structures.

Is the theory complete?
No, consider for example

$$\forall x, y (onTable(x) \land onTable(y) \rightarrow x \neq y)$$

What about planning in the blocksworld?
This should be done automatically as in the case of the Sudoku game or the puzzle.
Thus we need a FOL theorem prover.

An axiomatization is needed!
A complete axiomatization

The following set $\mathcal{AX}_{BW}$ of axioms was proposed by Cook and Liu (2003):

1. $\forall x \neg \text{above}(x, x)$
2. $\forall x \forall y \forall z \left( \text{above}(x, y) \land \text{above}(y, z) \right) \rightarrow \text{above}(x, z)$
3. $\forall x \forall y \forall z \left( \text{above}(x, y) \land \text{above}(x, z) \right) \rightarrow (y = z \lor \text{above}(y, z) \lor \text{above}(z, y))$
4. $\forall x \forall y \forall z \left( \text{above}(y, x) \land \text{above}(z, x) \right) \rightarrow (y = z \lor \text{above}(y, z) \lor \text{above}(z, y))$
5. $\forall x \left( \text{onTable}(x) \lor \exists y \left( \text{above}(x, y) \land \text{onTable}(y) \right) \right)$
6. $\forall x \left( \text{clear}(x) \lor \exists y \left( \text{above}(y, x) \land \text{clear}(y) \right) \right)$
7. $\forall x \forall y \left( \text{above}(x, y) \rightarrow (\exists z \text{on}(x, z) \land \exists z \text{on}(z, y)) \right)$

The last statement says that if an element is not on top ($y$) then there is a block above it, and if an element is not at the bottom ($x$) then there is an element below it.

Is every $\mathcal{L}_{BW}$-BW-structure also a model for $\mathcal{AX}_{BW}$?
Lemma 7.21

\[ Cn(AX_{BW}) \subseteq Th(BW). \]

Proof: \( \rightsquigarrow \) Exercise

Indeed, both sets are identical:

Theorem 7.22 (Cook and Liu)

\[ Cn(AX_{BW}) = Th(BW). \]

Thus the axiomatization is sound and complete. Additionally, the theory is decidable!

\( \rightsquigarrow \) We are ready to use a theorem prover!
7.4 Higher order logic
Definition 7.23 (Second order logic $\mathcal{L}_{PL2}$)

The language $\mathcal{L}_{2\text{nd}OL}$ of second order logic consists of the language $\mathcal{L}_{FOL}$ and additionally

- for each $k \in \mathbb{N}_0$: $X^k_1, X^k_2, \ldots, X^k_n, \ldots$ a countable set $\text{RelVar}^k$ of $k$-ary predicate variables.

Thereby the set of terms gets larger:

- if $X^k$ is a $k$-ary predicate variable and $t_1, \ldots, t_k$ are terms, then $X^k(t_1, \ldots, t_k)$ is also a term

and also the set of formulae:

- if $X$ is a predicate variable, $\varphi$ a formula, then $(\exists X \varphi)$ and $(\forall X \varphi)$ are also formulae.

Not only elements of the universe can be quantified but also arbitrary subsets resp. $k$-ary relations.

The semantics do not change much – except for the new interpretation of formulae like $(\exists X \varphi)$, $(\forall X \varphi))$. 
We also require from $I_A$ that the new $k$-ary predicate variables are mapped onto $k$-ary relations on $U_A$.

- If $\varphi = \text{def } \forall X^k \psi$, then

\[
A(\varphi) = \text{def } \begin{cases} 
\top, & \text{if for all } R^k \subseteq U_A \times \cdots \times U_A : A[X^k/R^k](\psi) = \top, \\
\false, & \text{else.}
\end{cases}
\]

- If $\varphi = \text{def } \exists X^k \psi$, then

\[
A(\varphi) = \text{def } \begin{cases} 
\top, & \text{if there is a } R^k \subseteq U_A \times \cdots \times U_A \text{ with } A[X^k/R^k](\psi) = \top, \\
\false, & \text{else.}
\end{cases}
\]

We can quantify over arbitrary $n$-ary relations, not just over elements (like in first order logic).
Dedekind/Peano Characterization of \( \mathbb{N} \)

The natural numbers satisfy the following axioms:

- **Ax\(_1\)**: 0 is a natural number.
- **Ax\(_2\)**: For each natural number \( n \), there is exactly one successor \( S(n) \) of it.
- **Ax\(_3\)**: There is no natural number which has 0 as its successor.
- **Ax\(_4\)**: Each natural number is successor of at most one natural number.
- **Ax\(_5\)**: The set of natural numbers is the smallest set \( \mathbb{N} \) satisfying the following properties:
  1. \( 0 \in \mathbb{N} \),
  2. \( n \in \mathbb{N} \Rightarrow S(n) \in \mathbb{N} \).

The natural numbers are characterized up to isomorphy by these axioms.
How to formalize the last properties?

Language: We choose “0” as a constant and “\(S(\cdot)\)” as a unary function symbol.

Axiom for 0: \(\forall x \neg S(x) \equiv 0\).

Axiom for \(S\): \(\forall x \forall y (S(x) \equiv S(y) \rightarrow x \equiv y)\).

Axiom 5: \(\forall X ( (X(0) \land \forall x (X(x) \rightarrow X(S(x)))) \rightarrow \forall y X(y))\).

While the first two axioms are first-order, the last one is essentially second-order.
Different languages for $N_{PA}$

- $N_{PA} = (\mathbb{N}_0, 0^\mathbb{N}, 1^\mathbb{N}, +^\mathbb{N}, \divides^\mathbb{N})$
- $N_{PA} = (\mathbb{N}_0, 0^\mathbb{N}, S^\mathbb{N}, +^\mathbb{N}, \divides^\mathbb{N})$
- $N_{PA} = (\mathbb{N}_0, 0^\mathbb{N}, 1^\mathbb{N}, S^\mathbb{N}, +^\mathbb{N}, \divides^\mathbb{N})$
- $N_{PA} = (\mathbb{N}_0, 1^\mathbb{N}, +^\mathbb{N}, \divides^\mathbb{N})$

All these structures, resp. their formulae are interdefinable.
What do the following two 2nd order sentences mean?

\[ \forall x \forall y \, (x \equiv y \iff \forall X \, (X(x) \iff X(y))) \]

\[ \forall X \, (\forall x \exists ! y \, X(x, y) \land \forall x \forall y \forall z \, ((X(x, z) \land X(y, z)) \rightarrow x \equiv y) \rightarrow \forall x \exists y \, X(y, x)) \]
Answer:

The first sentence shows that equality can be defined in 2nd OL (in contrast to FOL). The second sentence holds in a structure iff it is finite. Note that this cannot be expressed in FOL.

While the semantics of $\mathcal{L}_{2nd\text{OL}}$ is a canonical extension of $\mathcal{L}_{FOL}$, this does not hold for the calculus level. It can be shown that the set of valid sentences in $\mathcal{L}_{2nd\text{OL}}$ is not even recursively enumerable.
Attention:

There is no **correct and complete calculus** for 2nd Order Logic!
8. Knowledge Engineering: Provers

- Theorem Proving
- Resolution
- Herbrand
- Variants of resolution
- SLD resolution
Content of this chapter:

**Provers:** We describe several *theorem provers* that are freely available and that can handle FOL.

**Resolution:** There are several *calculi* for implementing FOL. While the classical ones are not really feasible, *resolution* calculi can be efficiently implemented (and are the basis of most provers).

**Variants:** Resolution implementations often have huge search spaces. There are several variants that can be more efficiently implemented. *SLD resolution* leads to PROLOG as a programming language.
8.1 Theorem Proving
Automated Theorem Proving

Development of computer programs that show that some statement (the *conjecture*) is a logical consequence of a set of statements (*axioms* and *assumptions*).
First year math students usually prove that groups of order two are commutative.

Management consultants might formulate axioms that describe how organizations grow and interact, and from those axioms prove that organizational death rates decrease with age.
Examples (cont.)

- Hardware developers might validate the design of a circuit by proving a conjecture that describes a circuit’s performance, given axioms that describe the circuit itself.

- Lazy crosswords fans might prepare a database of words and rules of putting a crossword together, and generate the crosswords solution as a proof.
The Idea

- Specification of what we have (the system):
  Assumptions/axioms $A_1, \ldots, A_n$

- Specification of what we want: Goals $G_1, \ldots, G_k$

Now, we want to prove or disprove that

$A_1 \land \cdots \land A_n \models G_1 \land \cdots \land G_k$

\[\mapsto\] verification!

constructive proof \[\mapsto\] synthesis!
How can we use a theorem prover?

- **Correctness proving:** We prove
  \[ A_1 \land \cdots \land A_n \vdash G_1 \land \cdots \land G_k \]

- **Testing:** We look for counterexamples (models!)

- **Solution synthesis:** The constructed proof/model is the solution
  \[ \leadsto \text{software synthesis} \]
  \[ \leadsto \text{hardware synthesis} \]
  \[ \leadsto \text{plan generation etc.} \]
The language = logic

Often: Classical **First Order Logic**
Possibly: A non-classical logic or higher order logic
The system needs a **formal** description of the problem in some logical form.

- User prepares the description.
- Prover attempts to solve the problem.
- If **successful**: Proof is the output.
- If **unsuccessful**: User can provide guidance, simplify/split the problem, and/or revise the description.

This formality is both the underlying strength and handicap of ATP.
Pros of logical languages

- Allow a **precise formal statement** of the necessary information.
- **No ambiguity** in the statement of the problem (often the case when using natural language).
- Force the user to describe the problem precisely and accurately.
- The process often leads to a **clearer and deeper understanding** of the problem domain.
Cons of logical languages

- Require a precise formal statement of the necessary information
  \[\implies\text{many problems are not formalizable!}\]

Examples:
Creating a **good** user interface.
Creating a system which defends itself **reasonably well** against **unpredicted** situations.

- Even for domains which are in principle formalizable: **Inaccurate formalizations** are easy to obtain.
Cons of logical languages (cont.)

- Force the user to describe the problem precisely and accurately.
  \[
  \Rightarrow \text{considerable skills needed!}
  \]
- Computational intractability.
Computational complexity of the problem

- **Propositional logic:** co-NP-complete.
- **First-order logic:** recursively enumerable (re).
- **Higher-order logic:** Not even re.
Way out of the complexity

Computer-assisted theorem proving

Fully automatic theorem prover  \[\cdots\cdots\cdots\]  Proof checker
Interactive theorem provers require a human to give hints to the system.

The interaction may be at a very detailed level, where the user guides the inferences made by the system, or at a much higher level where the user determines intermediate lemmas to be proved on the way to the proof of a conjecture.

Often: the user defines a number of proof strategies proving toolbox (e.g., Isabelle).
Popular ATP techniques

- **First-order resolution with unification**, 
- Higher-order unification, 
- Model elimination, 
- Superposition and term rewriting, 
- Lean theorem proving, 
- Method of analytic tableaux, 
- Mathematical induction, 
- DPLL (Davis-Putnam-Logemann-Loveland algorithm).

All the previously mentioned complexity bounds still apply!
**CASC:** CADE ATP System Competition, a yearly contest of first-order provers for many important classes of first-order problems.

http://www.cs.miami.edu/~tptp/CASC/
Some important first-order provers (all have won at least one CASC competition division):

- E
- Gandalf
- Otter $\leadsto$ Prover9/Mace4
- SETHEO
- SPASS
- Vampire (won the "world cup for theorem provers" for eight years: 1999, 2001–2007)
- Waldmeister (won the CASC UEQ division for the last ten years: 1997-2006); recently embedded in Mathematica
Other Important Systems

- HOL
- Isabelle
- Mizar

Growing competition: Model checking (especially in logics of time, action and knowledge).
Example: animals, cats, and Garfield

Assumptions:
cat(x) -> animal(x).
cat(Garfield).

Goals:
exists x animal(x).
-(exists x animal(x)).

⇝ Ontologies, Semantic web
# Prover9 Syntax

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Connective</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>negation</td>
<td>–</td>
<td>(–p)</td>
</tr>
<tr>
<td>disjunction</td>
<td>[</td>
<td>(p</td>
</tr>
<tr>
<td>conjunction</td>
<td>&amp;</td>
<td>(p &amp; q &amp; r)</td>
</tr>
<tr>
<td>implication</td>
<td>–&gt;</td>
<td>(p –&gt; q)</td>
</tr>
<tr>
<td>equivalence</td>
<td>&lt;–&gt;</td>
<td>(p &lt;–&gt; q)</td>
</tr>
<tr>
<td>universal quant.</td>
<td>all</td>
<td>(all x all y p(x,y))</td>
</tr>
<tr>
<td>existential quant.</td>
<td>exists</td>
<td>(exists x p(x,y))</td>
</tr>
</tbody>
</table>
Prover9 Syntax (cont.)

- **Variables** start with (lower case) u through z
- Otherwise: **Constants**
- Free variables in clauses and formulas are assumed to be universally quantified at the outermost level

Prover9 uses **resolution with refutation**. Thus, its inference rules operate on clauses.

If non-clausal formulas are input, Prover9 immediately translates them to clauses by NNF, Skolemization, and CNF conversions.
Schubert’s Steamroller
Success Stories

Fields where ATP has been successfully used:

- **Logic and mathematics**.
- **Computer science and engineering** (software creation and verification, hardware design and verification – esp. integrated circuit design and verification, and knowledge based systems).
- **Social science**.
Mathematics

- **EQP**: Settling of the Robbins problem (open for 63 years!)
- **Otter**: Several results in quasi-groups; axiomatizations of algebraic structures.
- **Mizar**: Cross-verification of the Mathematical Library (ongoing).
- **Geometry Expert**: New results in Euclidean geometry.
- **NUPRL** helped to confirm Higman’s lemma and Gerard’s paradox.
Mathematics (cont.)

Results by dedicated provers (no general theorem provers):

- Proof of four color theorem (very controversial: The first mathematical proof which was impossible to verify by humans due to the enormous size of the program’s calculation).

- Solving the game of Connect Four.
Software creation

- **KIDS**: Used to derive scheduling algorithms that have outperformed currently used algorithms.
- **Amphion (NASA)**: Determining subroutines for satellite guidance
- **Certifiable Synthesis (NASA)**.
Software verification

- **KIV (Karlsruhe):** Verified an implementation of set functions, graph representations and algorithms, and a Prolog to WAM compiler.

- **PVS:** Diagnosis and scheduling algorithms for fault tolerant architectures; requirements specification for parts of the space shuttle flight control system.
Hardware verification

- Employed e.g. by IBM, Intel, and Motorola (especially since the Pentium FDIV bug).
- **ACL2**: Correctness proof of the floating point divide code for AMD5K86 microprocessor.
- **ANALYTICA**: Used to verify a division circuit that implements the floating point standard of IEEE.
- **RRL**: Verification of adder and multiplier circuits.
Hardware verification (cont.)

- **PVS**: Used to verify a microprocessor for aircraft flight control.
- **Nqthm**: Correctness proof of the FM9001 microprocessor.
- **HOL**: Hardware verification at Bell Laboratories.
Verification of critical section
8.2 Resolution
Definition 8.1 (Most general unifier: mgU)

Given a finite set of equations between terms or equations between literals. Then there is an algorithm which calculates a most general solution substitution (i.e. a substitution of the involved variables so that the left sides of all equations are syntactically identical to the right sides) or which returns fail. In the first case the most general solution substitution is defined (up to renaming of variables): it is called mgU, most general unifier.
\[ p(x, a) = q(y, b), \]

\[ p(g(a), f(x)) = p(g(y), z). \]

Basic substitutions are:

\[ [a/y, a/x, f(a)/z], [a/y, f(a)/x, f(f(a))/z], \]

and many more.

The \( m_{gu} \) is: \[ [a/y, f(x)/z]. \]
Given: \( f(x, g(h(y), y)) = f(x, g(z, a)) \)

The algorithm successively calculates the following sets of equations:

\[
\begin{align*}
\{ & x = x, \ g(h(y), y) = g(z, a) \} \\
\{ & g(h(y), y) = g(z, a) \} \\
\{ & h(y) = z, \ y = a \} \\
\{ & z = h(y), \ y = a \} \\
\{ & z = h(a), \ y = a \}
\end{align*}
\]

Thus the \( mgU \) is: \([x/x, a/y, h(a)/z]\).
The occur-check

Given: \( f(x, g(x)) = f(c, c) \)

Is there an mgU?
The algorithm gives the following:

\[ \{ x = c, \ g(x) = c \} \]

- But setting \( x = c \) is not a unifying substitution, because \( c \neq g(c) \).
- Therefore there is no mgU. And the algorithm has to do this check, called occur check, to test whether the substitution is really correct.
- However, this check is computationally expensive and many algorithms do not do it.
A resolution calculus for FOL

The resolution calculus is defined over the language $\mathcal{L}_{res} \subseteq \mathcal{L}_{FOL}$ where the set of well-formed formulae $\text{Fml}_{\mathcal{L}_{res}}$ consists of all disjunctions of the following form

$$A \lor \neg B \lor C \lor \ldots \lor \neg E,$$

i.e. the disjuncts are only atoms or their negations. No implications or conjunctions are allowed. These formulae are also called **clauses**. Such a clause is also written as the set

$$\{A, \neg B, C, \ldots, \neg E\}.$$

This means that the set-theoretic union of such sets corresponds again to a clause.

**Note**, that a clause now consists of atoms rather than constants, as it was the case of the resolution calculus for SL.
Definition 8.2 (Robinson’s resolution for FOL)

The resolution calculus consists of two rules:

\[ (\text{Res}) \quad \frac{C_1 \cup \{A_1\} \quad C_2 \cup \{\neg A_2\}}{(C_1 \cup C_2) \text{mgU}(A_1, A_2)} \]

where \( C_1 \cup \{A_1\} \) and \( C_2 \cup \{A_2\} \) are assumed to be disjunct wrt the variables, and the factorization rule

\[ (\text{Fac}) \quad \frac{C_1 \cup \{L_1, L_2\}}{(C_1 \cup \{L_1\}) \text{mgU}(L_1, L_2)} \]
Illustration of the resolution rule

Example 8.3

Consider the set

\[ M = \{ r(x) \lor \neg p(x), p(a), s(a) \} \]

and the question

\[ M \models \exists x (s(x) \land r(x)) \]
Definition 8.4 (Resolution Calculus for FOL)

We define the resolution calculus $\text{Robinson}_{\mathcal{L}_{res}}^{\text{FOL}} = \langle \emptyset, \{\text{Res}, \text{Fac}\} \rangle$ as follows. The underlying language is $\mathcal{L}_{res} \subseteq \mathcal{L}_{\text{FOL}}$ defined on Slide 627 together with the set of well-formed formulae $\text{Fml}_{\mathcal{L}_{res}}^{\text{Res}}$.

Thus there are no axioms and only two inference rules. The well-formed formulae are just clauses.

Question:

Is this calculus correct and complete?
Question:
Why do we need factorization?

Answer:
Consider

\[ M = \{ \{s(x_1), s(x_2)\}, \{\neg s(y_1), \neg s(y_2)\} \} \]

Resolving both clauses gives

\[ \{s(x_1)\} \cup \{\neg s(y_1)\} \]

or variants of it.
Resolving this new clause with one in \( M \) only leads to variants of the respective clause in \( M \).
Answer (continued):

□ can not be derived (using resolution only).

Factorization solves the problem, we can deduce both $s(x)$ and $\neg s(y)$, and from there the empty clause.

Theorem 8.5 (Resolution is refutation complete)

*Robinson’s resolution calculus* $\text{Robinson}^{\text{FOL}}_{\text{res}}$ is refutation complete: Given an unsatisfiable set, the empty clause can eventually be derived using resolution and factorization.
What to do if the formulae are not clauses?

In that case, we have to transform them into clauses of the required form.

Let us consider

**Example 8.6**

Let

\[ M = \{ \exists x \exists y \; Q(x) \land \neg Q(y), \; \forall x \exists y \; P(x, y) \rightarrow \neg Q(y) \} \]

How to deal with the existential quantifiers?
The first formula is easy to transform: we just add two new constants \( c, c' \) and instantiate them for the variables. This leads to \( Q(c) \land \neg Q(c') \), or the two clauses \( Q(c) \) and \( Q(c') \).

The second formula is more complicated, because there is no single \( y \). We have to take into account, that the \( y \) usually depends on the chosen \( x \). Therefore we introduce a function symbol \( f(x) \):

\[
\forall x \, P(x, f(x)) \rightarrow \neg Q(f(x)).
\]

Then we have to transform the formula into disjunctive form:

\[
\forall x \, \neg P(x, f(x)) \lor \neg Q(f(x)).
\]
Transformation into a set of clauses

Applying the technique of the last slide recursively, we can transform each set $M$ in a language $L$ into a set $M'$ of clauses in an extended language $L'$. It can be shown that the following holds.

**Theorem 8.7**

Let $M$ be a set of first-order sentences in a language $L$ and let $M'$ be its transform. Then

$M$ is satisfiable if and only if $M'$ is satisfiable.

Because of the refutation completeness of the resolution calculus, this property is all we need.
8.3 Herbrand
The introduced relation $T \models \phi$ states that each model of $T$ is also a model of $\phi$. But because there are many models with very large universes the following question comes up: can we restrict to particular models?

**Theorem 8.8 (Löwenheim-Skolem)**

$T \models \phi$ holds if and only if $\phi$ holds in all countable models of $T$.

By countable we mean the size of the universe of the model.
Quite often the universes of models (which we are interested in) consist exactly of the basic terms $\text{Term}_L(\emptyset)$. This leads to the following notion:

**Definition 8.9 (Herbrand model)**

A model $A$ is called **Herbrand model** with respect to a language if the universe of $A$ consists exactly of $\text{Term}_L(\emptyset)$ and the function symbols $f^k_i$ are interpreted as follows:

$$f^k_i^A : \text{Term}_L(\emptyset) \times \ldots \times \text{Term}_L(\emptyset) \to \text{Term}_L(\emptyset); \quad (t_1, \ldots, t_k) \mapsto f^k_i(t_1, \ldots, t_k)$$

We write $T \models_{\text{Herb}} \phi$ if each Herbrand model of $T$ is also a model of $\phi$. 

---

Prof. Dr. Jürgen Dix
Clausthal, SS 2011
Theorem 8.10 (Reduction to Herbrand models)

If $T$ is universal and $\phi$ existential, then the following holds:

$$T \models \phi \text{ if and only if } T \models_{\text{Herb}} \phi$$

Question:

Is $T \models_{\text{Herb}} \phi$ not much easier, because we have to consider only Herbrand models? Is it perhaps decidable?

No, truth in Herbrand models is highly undecidable.
The following theorem is the basic result for applying resolution. In a way it states that FOL can be somehow reduced to SL.

**Theorem 8.11 (Herbrand)**

Let $T$ be universal and $\phi$ without quantifiers. Then:

$$T \models \exists \phi \text{ if and only if } \text{there are } t_1, \ldots, t_n \in \text{Term}_L(\emptyset) \text{ with: } T \models \phi(t_1) \lor \ldots \lor \phi(t_n)$$

Or: Let $M$ be a set of clauses of FOL (formulae in the form $P_1(t_1) \lor \neg P_2(t_2) \lor \ldots \lor P_n(t_n)$ with $t_i \in \text{Term}_L(X)$). Then:

$M$ is unsatisfiable if and only if there is a finite and unsatisfiable set $M_{inst}$ of basic instances of $M$. 
In automatic theorem proving we are always interested in the question

\[ M \models \exists x_1, \ldots, x_n \bigwedge_i \phi_i \]

Then

\[ M \cup \{ \neg \exists x_1, \ldots, x_n \bigwedge_i \phi_i \} \]

is a set of clauses, which is unsatisfiable if and only if the relation above holds.
8.4 Variants of resolution
Our general goal is to derive an existentially quantified formula from a set of formulae:

\[ M \vdash \exists \varphi. \]

To use resolution we must form \( M \cup \{\neg \exists \varphi\} \) and put it into the form of clauses. This set is called input.

Instead of allowing arbitrary resolvents, we try to restrict the search space.
Example 8.12 (Unlimited Resolution)

Let $M := \{ r(x) \lor \neg p(x), p(a), s(a) \}$ and $
\square \leftarrow s(x) \land r(x)$ the query.
An unlimited resolution might look like this:

\[
\begin{array}{cccc}
  r(x) \lor \neg p(x) & p(a) & s(a) & \neg s(x) \lor \neg r(x) \\
  \hline
  r(a) & \neg s(x) & \neg r(a) & \\
  \hline
  \square
\end{array}
\]
**Input resolution:** in each resolution step one of the two parent clauses must be from the input. In our example:

\[
\begin{align*}
\neg s(x) \lor \neg r(x) & \quad s(a) \\
\hline
\neg r(a) & \quad r(x) \lor \neg p(x)
\end{align*}
\]

\[
\begin{align*}
\neg r(a) \quad p(a)
\end{align*}
\]

\[
\begin{align*}
\neg p(a)
\end{align*}
\]

**Linear resolution:** in each resolution step one of the two parent clauses must either be from the input or must be a successor of the other parent clause.
Theorem 8.13 (Completeness of resolution variants)

Linear resolution is refutation complete. Input resolution is correct but not refutation complete.

Idea:

Maybe input resolution is complete for a restricted class of formulae.
8.5 SLD resolution
**Definition 8.14 (Horn clause)**

A clause is called **Horn clause** if it contains at most one positive atom.

A Horn clause is called **definite** if it contains exactly one positive atom. It has the form

\[ A(t) \leftarrow A_1(t_1), \ldots, A_n(t_n). \]

A Horn clause without positive atom is called **query**:

\[ \Box \leftarrow A_1(t_1), \ldots, A_n(t_n). \]
Theorem 8.15 (Input resolution for Horn clauses)

Input resolution for Horn clauses is refutation complete.

Definition 8.16 (SLD resolution wrt $P$ and query $Q$)

SLD resolution with respect to a program $P$ and the query $Q$ is input resolution beginning with the query $\square \leftarrow A_1, \ldots, A_n$. Then one $A_i$ is chosen and resolved with a clause of the program. A new query emerges, which will be treated as before. If the empty clause $\square \leftarrow$ can be derived then SLD resolution was successful and the instantiation of the variables is called computed answer.
Theorem 8.17 (Correctness of SLD resolution)

Let $P$ be a definite program and $Q$ a query. Then each answer calculated for $P$ wrt $Q$ is correct.

Question:
Is SLD completely instantiated?

Definition 8.18 (Computation rule)

A computation rule $R$ is a function which assigns an atom $A_i \in \{A_1, \ldots, A_n\}$ to each query $\square \leftarrow A_1, \ldots, A_n$. This $A_i$ is the chosen atom against which we will resolve in the next step.

Note:
PROLOG always uses the leftmost atom.
In the following, we are illustrating SLD resolution on the following program:

\[
p(x, z) \leftarrow q(x, y), p(y, z)
\]

\[
p(x, x)
\]

\[
q(a, b)
\]

We would like to know, for which instances for \( x \), the fact \( p(x, b) \) follows from the above theory. Obviously, there are two solutions: \( x = a \) and \( x = b \) and these are the only ones.

We are now showing how to derive these solutions using SLD resolution.
8 Knowledge Engineering: Provers

8.5 SLD resolution

\[ p(x, b) \leftarrow q(x, y), p(y, b) \]

\[ p(b, b) \leftarrow q(b, u), p(u, b) \]

\[ b/x \]

\[ a/x \]

“Success”

“Failure”
8 Knowledge Engineering: Provers
8.5 SLD resolution

\[
\begin{align*}
&\leftarrow p(x, b) \\
&\quad \leftarrow q(x, y), p(y, b) \\
&\quad \quad \leftarrow q(x, y), q(y, u), p(u, b) \\
&\quad \quad \quad \leftarrow q(x, y), q(y, u), p(u, v), p(v, b) \\
&\quad \quad \quad \quad \leftarrow q(x, y), q(y, b), q(x, y), q(y, b) \\
&\quad \quad \quad \quad \quad \square [a/x] \\
&\quad \quad \quad \quad \quad \quad \leftarrow q(x, y), q(y, b), q(x, y), q(y, b) \\
&\quad \quad \quad \quad \quad \quad \quad \leftarrow q(x, y), q(y, b), q(x, y), q(y, b) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \leftarrow q(x, y), q(y, b), q(x, y), q(y, b) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \leftarrow q(x, a) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \leftarrow q(x, a) \\
&\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{"Failure"}
\end{align*}
\]
A SLD tree may have three different kinds of branches:

1. **infinite ones**,  
2. **branches ending with the empty clause** (and leading to an answer) and  
3. **failing branches** (dead ends).
Theorem 8.19 (Independence of computation rule)

Let \( R \) be a computation rule and \( \sigma \) an answer calculated wrt \( R \) (i.e. there is a successful SLD resolution). Then there is also a successful SLD resolution for each other computation rule \( R' \) and the answer \( \sigma' \) belonging to \( R' \) is a variant of \( \sigma \).
Theorem 8.20 (Completeness of SLD resolution)

Each correct answer substitution is subsumed through a calculated answer substitution. I.e.:

\[ P \models \forall Q \Theta \]

implies

SLD computes an answer \( \tau \) with:

\[ \exists \sigma : Q \tau \sigma = Q \Theta \]
Question:
How to find successful branches in a SLD tree?

Definition 8.21 (Search rule)
A search rule is a strategy to search for successful branches in SLD trees.

Note:
PROLOG uses depth-first-search.

A SLD resolution is determined by a computation rule and a search rule.
SLD trees for $P \cup \{Q\}$ are determined by the computation rule.

PROLOG is incomplete because of two reasons:

- depth-first-search
- incorrect unification (no occur check).
A third reason comes up if we also ask for finite and failed SLD resolutions:

- the computation rule must be fair, i.e. there must be a guarantee that each atom on the list of goals is eventually chosen.
<table>
<thead>
<tr>
<th>Programming versus knowledge engineering</th>
<th>knowledge engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>choose language</td>
<td>choose logic</td>
</tr>
<tr>
<td>write program</td>
<td>define knowledge base</td>
</tr>
<tr>
<td>write compiler</td>
<td>implement calculus</td>
</tr>
<tr>
<td>run program</td>
<td>derive new facts</td>
</tr>
</tbody>
</table>
9. Planning

- Planning vs. Problem-Solving
- STRIPS
- Partial-Order Planning
- Conditional Planning
- SHOP
- Extensions
Content of this chapter (1):

**Planning vs Search:** While planning can be seen as a purely search problem, available search methods are not feasible. We need to use knowledge about the problem and create a newly planning agent.

**STRIPS:** The STRIPS approach is based on using logical formulae as a representation language for planning.

**POP:** We describe a sound and complete partial order planning (POP) algorithm. POP is an action-based planner.
Content of this chapter (2):

**Conditional Planning:** Often a plan cannot be completely constructed a priori, because the environment is dynamic. The we need **sensing actions** that have to be checked at run-time.

**SHOP:** We introduce **HTN-planning**, which is based on **Hierarchical Task networks**. HTN planners use domain knowledge and are more expressive than action-based planners. **SHOP** is one of the most efficient HTN planners.

**Extensions:** We briefly describe **replanning** and how to combine it with conditional planning.
9.1 Planning vs. Problem-Solving
Motivation:

**problem-solving agent:** The effects of a static sequence of actions are determined.

**knowledge-based agent:** Actions can be chosen.

We try to merge both into a planning agent.
function SIMPLE-PLANNING-AGENT(percept) returns an action

static: KB, a knowledge base (includes action descriptions)
    p, a plan, initially NoPlan
    t, a counter, initially 0, indicating time

local variables: G, a goal
    current, a current state description

TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
current ← STATE-DESCRIPTION(KB, t)
if p = NoPlan then
    G ← ASK(KB, MAKE-GOAL-QUERY(t))
    p ← IDEAL-PLANNER(current, G, KB)
if p = NoPlan or p is empty then action ← NoOp
else
    action ← FIRST(p)
    p ← REST(p)
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
t ← t + 1
return action
Example 9.1 (Running Example)

We want to drink freshly made banana shake and drill some holes into a wall at home. Thus an agent needs to solve the following problem:

1. Get a quart of milk,
2. a bunch of bananas, and
3. a variable-speed cordless drill.
Question:
How does a problem-solving agent handle this?

Start

- Go To Pet Store
- Go To School
- Go To Supermarket
- Go To Sleep
- Read A Book
- Sit in Chair
- Etc. Etc. ...

- Buy Tuna Fish
- Buy Arugula
- Buy Milk
- Buy A Dog
- Talk to Parrot
- Sit Some More
- Read A Book

... → Finish
Planning in the situation calculus:

**Initial state:** \( \text{At(Home, } S_0) \), \( \neg \text{Have(Milk, } S_0) \), 
\( \neg \text{Have(Bananas, } S_0) \), \( \neg \text{Have(Drill, } S_0) \).

**Goal:** \( \exists s \ (\text{At(Home, } s) \land \text{Have(Milk, } s) \land \neg \text{Have(Bananas, } s) \land \text{Have(Drill, } s)) \).

**Axioms:** e.g. “buy milk”:

\[
\forall a, s \ \text{Have(Milk, result}(a, s)) \leftrightarrow (a = \text{Buy(Milk)} \land \text{At(Supermarket, } s)) \lor (\text{Have(Milk, } s) \land a \neq \text{Drop(Milk)})
\]
We also need a term \( \text{Result}'(l, s) \): the situation that would emerge when the sequence of actions \( l \) is executed in \( s \).

\[
\forall s \quad \text{Result}'([], s) := s \\
\forall a, p, s \quad \text{Result}'([a|p], s) := \text{Result}'(p, \text{result}(a, s))
\]
The task is now to find a $p$ with

$\text{At(Home, Result}'(p, S_0)) \land$

$\text{Have(Milk, Result}'(p, S_0)) \land$

$\text{Have(Bananas, Result}'(p, S_0)) \land$

$\text{Have(Drill, Result}'(p, S_0))$.

**Problem solved!**

This is the solution of our problem! **We start a theorem prover and collect the answers!**
Problems:

- $T \vdash \phi$ is only **semi-decidable** for FOL.
- If $p$ is a plan then so are $[Empty\_Action|p]$ and $[A, A^{-1}|p]$.

Result:
We should not resort to a **general prover**, but to one which is **specially designed** for our domain. We also should **restrict the language**.
We should make clear the difference between

- shipping a goal to a planner,
- asking a query to a theorem prover.

In the first case we look for a plan so that after the execution of the plan the goal holds.

In the second case we ask if the query can be made true wrt the KB: $KB \models \exists x \phi(x)$.

The dynamics is in the terms. The logic itself is static.
9.2 STRIPS
STRIPS stands for Stanford Research Institute Problem Solver.

**states:** conjunctions of function-free *ground-atoms (positive literals).*

**goal:** conjunctions of function-free literals

**actions:** STRIPS-operations consist of three components

1. *description,* name of the action
2. *precondition,* conjunction of atoms
3. *postcondition,* conjunction of literals
At(here), Path(here, there)

Go(there)

At(there), \neg At(here)
Example 9.2 (Air Cargo Transportation)

Three actions: *Load*, *Unload*, *Fly*. Two predicates

\[ \text{In}(c, p): \text{cargo } c \text{ is inside plane } p, \]
\[ \text{At}(x, a): \text{object } x \text{ is at airport } a. \]

Is cargo \( c \) at airport \( a \) when it is loaded in a plane \( p \)?
\[\text{Init}(\text{At}(C_1, \text{SFO}) \land \text{At}(C_2, \text{JFK}) \land \text{At}(P_1, \text{SFO}) \land \text{At}(P_2, \text{JFK})
\land \text{Cargo}(C_1) \land \text{Cargo}(C_2) \land \text{Plane}(P_1) \land \text{Plane}(P_2)
\land \text{Airport}(\text{JFK}) \land \text{Airport}(\text{SFO}))
\]

\[\text{Goal}(\text{At}(C_1, \text{JFK}) \land \text{At}(C_2, \text{SFO}))\]

\[\text{Action}(\text{Load}(c, p, a),
\quad \text{PRECOND}: \text{At}(c, a) \land \text{At}(p, a) \land \text{Cargo}(c) \land \text{Plane}(p) \land \text{Airport}(a)
\quad \text{EFFECT}: \neg \text{At}(c, a) \land \text{In}(c, p))\]

\[\text{Action}(\text{Unload}(c, p, a),
\quad \text{PRECOND}: \text{In}(c, p) \land \text{At}(p, a) \land \text{Cargo}(c) \land \text{Plane}(p) \land \text{Airport}(a)
\quad \text{EFFECT}: \text{At}(c, a) \land \neg \text{In}(c, p))\]

\[\text{Action}(\text{Fly}(p, \text{from}, \text{to}),
\quad \text{PRECOND}: \text{At}(p, \text{from}) \land \text{Plane}(p) \land \text{Airport}(\text{from}) \land \text{Airport}(\text{to})
\quad \text{EFFECT}: \neg \text{At}(p, \text{from}) \land \text{At}(p, \text{to}))\]
Example 9.3 (Spare Tire)

Four actions:

1. Remove(spare, trunk),
2. Remove(flat, axle),
3. PutOn,
4. LeaveOvernight.

One predicate $At(x, a)$ meaning object $x$ is at location $a$.

Is the following a STRIPS description?
\[\text{Init}(\text{At}(\text{Flat}, \text{Axle}) \land \text{At}(\text{Spare}, \text{Trunk}))\]

\[\text{Goal}(\text{At}(\text{Spare}, \text{Axle}))\]

\[\text{Action}(\text{Remove}(\text{Spare}, \text{Trunk}),\]
  \hspace{1em} \text{Precond}: \text{At}(\text{Spare}, \text{Trunk})
  \hspace{1em} \text{Effect}: \neg \text{At}(\text{Spare}, \text{Trunk}) \land \text{At}(\text{Spare}, \text{Ground}))\]

\[\text{Action}(\text{Remove}(\text{Flat}, \text{Axle}),\]
  \hspace{1em} \text{Precond}: \text{At}(\text{Flat}, \text{Axle})
  \hspace{1em} \text{Effect}: \neg \text{At}(\text{Flat}, \text{Axle}) \land \text{At}(\text{Flat}, \text{Ground}))\]

\[\text{Action}(\text{PutOn}(\text{Spare}, \text{Axle}),\]
  \hspace{1em} \text{Precond}: \text{At}(\text{Spare}, \text{Ground}) \land \neg \text{At}(\text{Flat}, \text{Axle})
  \hspace{1em} \text{Effect}: \neg \text{At}(\text{Spare}, \text{Ground}) \land \text{At}(\text{Spare}, \text{Axle}))\]

\[\text{Action}(\text{LeaveOvernight},\]
  \hspace{1em} \text{Precond}:\]
  \hspace{2em} \text{Effect}: \neg \text{At}(\text{Spare}, \text{Ground}) \land \neg \text{At}(\text{Spare}, \text{Axle}) \land \neg \text{At}(\text{Spare}, \text{Trunk})
  \hspace{2em} \neg \text{At}(\text{Flat}, \text{Ground}) \land \neg \text{At}(\text{Flat}, \text{Axle}))\]
Example 9.4 (Blocks World)

- One action: $\textit{Move}$. $\textit{Move}(b, x, y)$ moves the block $b$ from $x$ to $y$ if both $b$ and $y$ are clear.
- One predicate $\textit{On}(b, x)$ meaning block $b$ is on $x$ ($x$ can be another block or the table).

How to formulate that a block is clear?

“$b$ is clear”: $\forall x \neg \textit{On}(x, b)$.

Not allowed in STRIPS.
Therefore we introduce another predicate $\text{Clear}(y)$.

What about $\text{Move}(b, x, y)$ defined by

**Precondition:** $\text{On}(b, x) \land \text{Clear}(b) \land \text{Clear}(y)$ and

**Effect:** $\text{On}(b, y) \land \text{Clear}(x) \land \neg \text{On}(b, x) \land \neg \text{Clear}(y)$?
\begin{align*}
\text{Init}(\text{On}(A, \text{Table}) & \land \text{On}(B, \text{Table}) \land \text{On}(C, \text{Table}) \\
& \land \text{Block}(A) \land \text{Block}(B) \land \text{Block}(C) \\
& \land \text{Clear}(A) \land \text{Clear}(B) \land \text{Clear}(C))) \\
\text{Goal}(\text{On}(A, B) & \land \text{On}(B, C)) \\
\text{Action}(\text{Move}(b, x, y), \\
\text{PRECOND}: \text{On}(b, x) & \land \text{Clear}(b) \land \text{Clear}(y) \land \text{Block}(b) \land \\
(b \neq x) & \land (b \neq y) \land (x \neq y), \\
\text{EFFECT}: \text{On}(b, y) & \land \text{Clear}(x) \land \neg \text{On}(b, x) \land \neg \text{Clear}(y)) \\
\text{Action}(\text{MoveToTable}(b, x), \\
\text{PRECOND}: \text{On}(b, x) & \land \text{Clear}(b) \land \text{Block}(b) \land (b \neq x), \\
\text{EFFECT}: \text{On}(b, \text{Table}) & \land \text{Clear}(x) \land \neg \text{On}(b, x))
\end{align*}
ADL (Action Description Language) and its many derivatives are extensions of STRIPS:

**States:** both positive and negative literals in states.

**OWA:** Open World assumption (not CWA)

**Effects:** Add all positive and negative literals, and delete their negations.

**Quantification:** Quantified variables in goals are allowed.

**Goals:** disjunction and negation also allowed.

**when** $P$: Conditional effects allowed.
(a) **Progression** (forward) and (b) **Regression** (backward) state-space search
\[ \text{At}(\text{here}), \text{Path}(\text{here, there}) \]

\[ \text{Go}(\text{there}) \]

\[ \text{At}(\text{there}), \neg \text{At}(\text{here}) \]
Definition 9.5 (Applicable Operator)

An operator $Op$ is **applicable in a state** $s$ if there is some way to instantiate the variables in $Op$ so that every one of the preconditions of $Op$ is true in $s$: 

$$\text{Precond}(Op) \subseteq s.$$ 

In the resulting state, all the positive literals in $\text{Effect}(Op)$ hold, as do all the literals that held in $s$, except for those that are negative literals in $\text{Effect}(Op)$. 
Frame problem is handled implicitly: literals not mentioned in effects remain unchanged (persistence).

Effect is sometimes split into add and delete lists.

Up to now we can consider this as being problem-solving.

We use STRIPS as an representation-formalism and search a solution-path:

- nodes in the search-tree ≈ situations
- solution paths ≈ plans.
Idea:

**Perform a search in the space of all plans!**

Begin with a partial plan and extend it successively.

Therefore we need operators which operate on plans. We distinguish two types:

- **refinement-op:** constraints are attached to a plan. Then a plan represents the set of all complete plans (analogously $Cn(T)$ for $MOD(T)$).

- **modification-op:** all others.
Question:
How do we represent plans?

Answer:
We have to consider two things:

- **instantiation of variables**: instantiate only if necessary, i.e. always choose the mgU
- **partial order**: refrain from the exact ordering (reduces the search-space)
A plan is formally defined as a data structure consisting of the following four components:

- A set of **plan steps**. Each step is one of the operators for the problem.

- A set of **step ordering constraints**. Each one of the form $S_i < S_j$: $S_i$ has to be executed before $S_j$.

- A set of **variable binding constants**. Each one of the form $v = x$: $v$ is a variable in some step, and $x$ is either a constant or another variable.

- A set of **causal links**. Each one of the form $S_i \xrightarrow{c} S_j$: $S_i$ achieves $c$ for $S_j$. 
The initial plan consists of two steps, called START and FINISH.
The initial plan of a problem is:

\[
\text{Plan(Steps : } \{S_1 : START, S_2 : FINISH\} \\
\text{Orderings : } \{S_1 \prec S_2\} \\
\text{Bindings : } \emptyset \\
\text{Links : } \emptyset
\)
Partial Order Plan:

- Start
- Left Sock
- Right Sock
- Left SockOn
- Right SockOn
- Left Shoe
- Right Shoe
- LeftShoeOn, RightShoeOn
- Finish

Total Order Plans:

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish

- Start
  - Right Sock
  - Left Sock
  - Right SockOn
  - Left SockOn
  - Right Shoe
  - Left Shoe
  - Right ShoeOn
  - Left ShoeOn
  - Right Shoe
  - Left Shoe
  - Finish
Question: What is a solution?

Answer: Considering only **fully instantiated, linearly ordered plans**: *checking is easy.*

But our case is far more complicated:

**Definition 9.7 (Solution of a Plan)**
A solution is a **complete** and **consistent** plan.

**complete**: each precondition of each step is achieved by some other step,

**consistent**: there are no contradictions in the ordering or binding constraints.
More precisely:

- "\( S_i \) achieves \( c \) for \( S_j \)" means
  - \( c \in \text{Precondition}(S_j) \),
  - \( c \in \text{Effect}(S_i) \),
  - \( S_i \prec S_j \),
  - \( \exists S_k : \neg c \in \text{Effect}(S_k) \) with \( S_i \prec S_k \prec S_j \) in any linearization of the plan.

- "no contradictions" means
  - neither \( (S_i \prec S_j \) and \( S_j \prec S_i \) nor \( (v = A \) and \( v = B \) for different constants \( A, B \)).

Note: these propositions may be derivable, because \( \prec \) and \( = \) are transitive.
9.3 Partial-Order Planning
We consider the banana-milk example. The operators are \textit{Buy} and \textit{Go}.

\begin{center}
\begin{tikzpicture}
  \node[rectangle, draw] (start) at (0,0) {Start};
  \node[rectangle, draw] (finish) at (0,-2) {Finish};
  \draw[->] (start) -- node[above] {At(Home) Sells(SM,Banana)} (finish);
  \draw[->] (start) -- node[above] {Sells(SM,Milk) Sells(HWS,Drill)} (finish);
  \draw[->] (start) -- node[above] {Have(Drill) Have(Milk)} (finish);
  \draw[->] (start) -- node[above] {Have(Banana) At(Home)} (finish);
\end{tikzpicture}
\end{center}

\textbf{Causal links are protected!}
9.3 Partial-Order Planning

At(SM), Sells(SM, Bananas)

At(SM), Sells(SM, Milk)

At(HWS), Sells(HWS, Drill)

Have(Drill), Have(Milk), Have(Bananas), At(Home)

Finish

Go(HWS)

Buy(Drill)

Go(SM)

Buy(Milk)

Buy(Bananas)

At(Home)

At(HWS), Sells(HWS, Drill)

At(SM), Sells(SM, Milk)

At(SM), Sells(SM, Bananas)

Have(Drill), Have(Milk), Have(Bananas), At(Home)

Finish

Go(SM)
The last partial plan **cannot be extended**. How do we determine this? By determining that a causal link is threatened and the threat cannot be resolved.

The way to resolve the threat is to **add ordering constraints** (this will not always work!).
Question:

We have to introduce a Go-step in order to ensure the last precondition. But how can we ensure the precondition of the Go-step?

Now there are a lot of threats and many of them are unresolvable. This leads to

![Diagram of planning process](image-url)
9 Planning

9.3 Partial-Order Planning

At(Home)  Have(Milk)  At(Home)  Have(Ban.)  Have(Drill)  Buy(Drill)  Buy(Milk)  Buy(Ban.)  Go(Home)  Go(HWS)  Go(SM)  Finish

Start

At(HWS)  Sells(HWS,Drill)

Buy(Drill)

At(HWS)

Go(SM)

At(SM)  Sells(SM,Milk)  At(SM)  Sells(SM,Ban.)

Buy(Milk)  Buy(Ban.)

At(SM)

Go(Home)

Have(Milk)  At(Home)  Have(Ban.)  Have(Drill)

Finish

Prof. Dr. Jürgen Dix
Clausthal, SS 2011
This leads to the following algorithm

- In each round the plan is extended in order to **ensure the precondition of a step**. This is done by choosing an appropriate operator.
- The respective **causal link is introduced**. Threats are resolved through ordering constraints (two cases: the new step threatens existing ones or the existing ones threaten the new one).
- If there is **no operator** or the threat **cannot** be resolved then perform **backtracking**.

**Theorem 9.8 (POP)**

*POP is complete and correct.*
function POP(initial, goal, operators) returns plan

    plan ← MAKE-MINIMAL-PLAN(initial, goal)
    loop do
        if SOLUTION?(plan) then return plan
        Sneed, c ← SELECT-SUBGOAL(plan)
        CHOOSE-OPERATOR(plan, operators, Sneed, c)
        RESOLVE-THREATS(plan)
    end

function SELECT-SUBGOAL(plan) returns Sneed, c

    pick a plan step Sneed from STEPS(plan)
    with a precondition c that has not been achieved
    return Sneed, c

procedure CHOOSE-OPERATOR(plan, operators, Sneed, c)

    choose a step Sadd from operators or STEPS(plan) that has c as an effect
    if there is no such step then fail
    add the causal link Sadd ↷ Sneed to LINKS(plan)
    add the ordering constraint Sadd △ Sneed to ORDERINGS(plan)
    if Sadd is a newly added step from operators then
        add Sadd to STEPS(plan)
        add Start △ Sadd △ Finish to ORDERINGS(plan)

procedure RESOLVE-THREATS(plan)

    for each Sthreat that threatens a link Si ↷ Sj in LINKS(plan) do
        choose either
            Promotion: Add Sthreat △ Si to ORDERINGS(plan)
            Demotion: Add Sj △ Sthreat to ORDERINGS(plan)
        if not CONSISTENT(plan) then fail
    end
So far we did not consider variable-substitutions.

**Question:**

Suppose $S_1$ ensures the $At(home)$ precondition of a step $S_2$ and there is a concurrent step $S_3$ with the postcondition $\neg At(x)$. Is this a threat?
We call such a threat **possible** and ignore it for the time being, but keep it in mind. If $x$ is later instantiated with *home* then a real threat is there which has to be resolved.
procedure CHOOSE-OPERATOR\(\textit{plan}, \textit{operators}, S_{\text{need}}, c\)  

\begin{align*}
\textbf{choose} \quad \text{a step } S_{\text{add}} \text{ from } \textit{operators} \text{ or } \textit{STEPS}(\textit{plan}) \text{ that has } c_{\text{add}} \text{ as an effect} \\
\text{such that } u = \text{UNIFY}(c, c_{\text{add}}, \text{BINDINGS}(\textit{plan})) \\
\text{if} \text{ there is no such step} \\
\text{then fail} \\
\text{add } u \text{ to } \text{BINDINGS}(\textit{plan}) \\
\text{add } S_{\text{add}} \xrightarrow{c} S_{\text{need}} \text{ to } \text{LINKS}(\textit{plan}) \\
\text{add } S_{\text{add}} \prec S_{\text{need}} \text{ to } \text{ORDERINGS}(\textit{plan}) \\
\text{if } S_{\text{add}} \text{ is a newly added step from } \textit{operators} \text{ then} \\
\quad \text{add } S_{\text{add}} \text{ to } \textit{STEPS}(\textit{plan}) \\
\quad \text{add } \text{Start} \prec S_{\text{add}} \prec \text{Finish} \text{ to } \text{ORDERINGS}(\textit{plan}) \\
\end{align*}

procedure RESOLVE-THREATS(\textit{plan})  

\begin{align*}
\text{for each } S_i \xrightarrow{c} S_j \text{ in } \text{LINKS}(\textit{plan}) \text{ do} \\
\text{for each } S_{\text{threat}} \text{ in } \textit{STEPS}(\textit{plan}) \text{ do} \\
\text{for each } c' \text{ in } \textit{EFFECT}(S_{\text{threat}}) \text{ do} \\
\quad \text{if } \text{SUBST}(\text{BINDINGS}(\textit{plan}), c) = \text{SUBST}(\text{BINDINGS}(\textit{plan}), \neg c') \text{ then} \\
\qquad \text{choose either} \\
\qquad \quad \textit{Promotion}: \text{Add } S_{\text{threat}} \prec S_i \text{ to } \text{ORDERINGS}(\textit{plan}) \\
\qquad \quad \textit{Demotion}: \text{Add } S_j \prec S_{\text{threat}} \text{ to } \text{ORDERINGS}(\textit{plan}) \\
\quad \text{if not } \text{CONSISTENT}(\textit{plan}) \text{ then fail} \\
\end{align*}
STRIPS was originally designed for SHAKEY, a small and mobile robot. SHAKEY is described through 6 Operators: $Go(x)$, $Push(b, x, y)$, $Climb(b)$, $Down(b)$, $Turn_{On}(ls)$, $Turn_{Off}(ls)$.

To turn the lights on or off SHAKEY has to stand on a box. $On(Shakey, floor)$ is a precondition of the Go-action so that SHAKEY does not fall off.
9 Planning

9.3 Partial-Order Planning

Room 4
- Door 4

Room 3
- Door 3
- Shakey

Corridor

Room 2
- Door 2

Room 1
- Door 1
- Box1
- Box2
- Box3
- Box4

Ls1
- Ls2
- Ls3
- Ls4
POP for ADL
9.3 Partial-Order Planning

- Remove(Spare, Trunk)
- At(Spare, Trunk)
- PutOn(Spare, Axle)
- At(Spare, Ground)
- At(Flat, Axle)
- Finish
- Start
- LeaveOvernight
- At(Flat, Axle)
- At(Spare, Trunk)
- At(Flat, Axle)
- At(Spare, Axle)
- At(Spare, Ground)
- At(Spare, Trunk)
9.3 Partial-Order Planning

Start

Remove(Spare, Trunk)
At(Spare, Trunk)
Remove(Flat, Axle)
At(Flat, Axle)
PutOn(Spare, Axle)
At(Spare, Ground)
At(Flat, Axle)
Finish
At(Spare, Axle)
9.4 Conditional Planning
Question:
What to do if the world is not fully accessible? Where to get milk? Milk-price has doubled and we do not have enough money.

Idea:
Introduce new sensing actions to query certain conditions and to react accordingly. (How much does milk cost?)
A flat tire

\(\text{Op( Action : Remove}(x)\),
\text{Prec : On}(x),
\text{Effect : Off}(x) \land \text{Clear}_\text{Hub} \land \neg \text{On}(x)\)

\(\text{Op( Action : Put}_\text{on}(x),
\text{Prec : Off}(x) \land \text{Clear}_\text{Hub},
\text{Effect : On}(x) \land \neg \text{Clear}_\text{Hub} \land \neg \text{Off}(x)\)

\(\text{Op( Action : Inflate}(x),
\text{Prec : Intact}(x) \land \text{Flat}(x),
\text{Effect : Inflated}(x) \land \neg \text{Flat}(x)\)

\(\text{goal: On}(x) \land \text{Inflated}(x),\)
\(\text{initial state: Inflated}(\text{Spare}) \land \text{Intact}(\text{Spare}) \land
\text{Off}(\text{Spare}) \land \text{On}(\text{Tire}_1) \land \text{Flat}(\text{Tire}_1).\)
Question:
What does POP deliver?

Answer:
Because of $\text{Intact}(\text{Tire}_1)$ not being present POP delivers the plan

\[ \text{[Remove}(\text{Tire}_1), \text{Put\_on}(\text{Spare})]. \]
Question:
Would you also do it like that?

Answer:
A better way would be a **conditional** plan:

If \( \text{Intact}(\text{Tire}_1) \) then: \( \text{Inflate}(\text{Tire}_1) \).
Therefore we have to allow *conditional* steps in the phase of plan-building:

**Definition 9.9 (Conditional Step)**

A *conditional step* in a plan has the form

$$\text{If}([\text{Condition}], [\text{Then}_\text{Part}], [\text{Else}_\text{Part}])$$
The respective planning-agent looks like this:

```plaintext
function CONDITIONAL-PLANNING-AGENT( percept) returns an action
    static: KB, a knowledge base (includes action descriptions)
        p, a plan, initially NoPlan
        t, a counter, initially 0, indicating time
        G, a goal

    TELL(KB, MAKE-PERCEPT-SENTENCE( percept, t))
    current ← STATE-DESCRIPTION(KB, t)
    if p = NoPlan then p ← CPOP(current, G, KB)
    if p = NoPlan or p is empty then action ← NoOp
    else
        action ← FIRST(p)
        while CONDITIONAL?(action) do
            if ASK(KB, CONDITION-PART[action]) then p ← APPEND(THEN-PART[action], REST(p))
            else p ← APPEND(ELSE-PART[action], REST(p))
            action ← FIRST(p)
        end
        p ← REST(p)
    TELL(KB, MAKE-ACTION-SENTENCE(action, t))
    t ← t + 1
    return action
```

The agent has to know if the respective if-condition holds or not when executing the plan (at runtime). Therefore we introduce new checking actions:

\[
\text{Op} \begin{align*}
\text{Action} : & \quad \text{Check} \_\text{Tire}(x), \\
\text{Prec} : & \quad \text{Tire}(x), \\
\text{Effect} : & \quad \text{"We know if } \text{Intact}(x) \text{ holds or not."}
\end{align*}
\]

**Definition 9.10 (Context)**

We associate a **context** with each step: the set of conditions which have to hold before executing a step.
9 Planning

9.4 Conditional Planning

Inflated(Spare)
On(Tire1)
Flat(Tire1)
Start Finish
(True)
Inflated(x)
On(x)

Intact(Tire1)
Inflated(Spare)
On(Tire1)
Flat(Tire1)
Start Finish
Inflate(Tire1)
(True)

Prof. Dr. Jürgen Dix Clausthal, SS 2011

Clausthal, SS 2011 721
Here POP would backtrack, because $\text{Intact}(Tire_1)$ cannot be shown.

Hence we introduce a new type of links: conditional links.
9 Planning

9.4 Conditional Planning

Start

- On(Tire1)
- Flat(Tire1)
- Inflated(Spare)

Check(Tire1)

Finish

- On(Tire1)
- Inflated(Tire1)
- Intact(Tire1)

Inflate(Tire1)

Inflated(Spare)

Intact(Tire1)

Prof. Dr. Jürgen Dix
Clausthal, SS 2011
We have to cover each case:
Question:
What if new contexts emerge in the second case?

Answer:
Then we have to introduce new copies of the FINISH step: There must be a complete distinction of cases in the end.
Question:

How can we make $\text{Inflated}(x)$ true in the FINISH step? Adding the step $\text{Inflate}(\text{Tire}_1)$ would not make sense, because the preconditions are inconsistent in combination with the context.
At last $\text{On}(\text{Spare})$.

Attention:

At first “True” is the context of $\text{Remove}(\text{Tire}_1)$, $\text{Put}_\text{on}(\text{Spare})$. But $\text{Remove}(\text{Tire}_1)$ threatens the $\text{On}(\text{Tire}_1)$ step (precondition of the first Finish).
We can resolve this threat by making the contexts incompatible. The respective contexts are inherited by the following steps.

More exactly:

Search a conditional step the precondition of which makes the contexts incompatible and thereby resolves threats.
function CPOP(initial, goals, operators) returns plan

plan ← MAKE-PLAN(initial, goals)
loop do

Termination:
if there are no unsatisfied preconditions
and the contexts of the finish steps are exhaustive
then return plan

Alternative context generation:
if the plans for existing finish steps are complete and have contexts C₁ … Cₙ then
add a new finish step with a context ¬(C₁ ∨ … ∨ Cₙ)
this becomes the current context

Subgoal selection and addition:
find a plan step S_need with an open precondition c

Action selection:
choose a step S_add from operators or STEPS(plan) that adds c or
knowledge of c and has a context compatible with the current context
if there is no such step
then fail
add S_add ¬→ S_need to LINKS(plan)
add S_add < S_need to ORDERINGS(plan)
if S_add is a newly added step then
add S_add to STEPS(plan)
add Start < S_add < Finish to ORDERINGS(plan)

Threat resolution:
for each step S_threat that potentially threatens any causal link Sᵢ ¬→ Sⱼ with a compatible context do
choose one of
Promotion: Add S_threat < Sᵢ to ORDERINGS(plan)
Demotion: Add Sⱼ < S_threat to ORDERINGS(plan)
Conditioning:
find a conditional step S_cond possibly before both S_threat and Sⱼ, where
1. the context of S_cond is compatible with the contexts of S_threat and Sⱼ;
2. the step has outcomes consistent with S_threat and Sⱼ, respectively
add conditioning links for the outcomes from S_cond to S_threat and Sⱼ
augment and propagate the contexts of S_threat and Sⱼ
if no choice is consistent
then fail
end
end
9.5 SHOP
Up to now: Action-based planning (POP, CPOP)

- Each state of the world is represented by a set of atoms, and each action corresponds to a deterministic state transition.
- Search space is still huge.
- HTN planning has been proved to be more expressive than action-based planning.
- Moreover, HTN planning algorithms have been experimentally proved to be more efficient than their action-based counterparts.
Each **state** of the world is represented by a **set of atoms**, and each action corresponds to a deterministic state transition.

**Hierarchical Task Networks**: HTN planners differ from classical planners in **what** they plan for, and **how** they plan for it.
Classical HTN planning (dating back to mid-70ies) focused on particular application domains: production-line scheduling, crisis management and logistics, planning and scheduling for spacecraft, equipment configuration, manufacturability analysis, evacuation planning, and the game of bridge.

There are also domain-independent HTN planners: Nonlin, Sipe-2, O-Plan, UMCP, SHOP, ASHOP, and SHOP2.

We focus on SHOP.
Features of HTN planning

- Why is HTN planning superior to classical action-based planning?
- The domain knowledge and the notion of decomposing a task network while satisfying the given constraints enable the planner to focus on a much smaller portion of the search space.
Features of SHOP

- SHOP is based on ordered task decomposition.
- SHOP plans for tasks in the same order that they will later be executed.
- Therefore we know the current state of the world at each step in the planning process.
- This eliminates a great deal of uncertainty about the world.
- It helps to incorporate inferencing and reasoning power into the planning system.
- It also enables to call external programs (e.g. to perform numeric computations).
SHOP needs the following

Knowledge: about the domain. Can be given as a set of axioms from which other facts can be deduced.

Operators: they describe what needs to be done to fulfill a primitive task.

Methods: often a task can not be fulfilled in one single step. In that case, the task needs to be reduced to other (new) tasks. Methods are prescriptions for how to decompose some compound (abstract) task into a totally ordered sequence of subtasks, along with various restrictions that must be satisfied in order for the method to be applicable.
More than one method may be applicable to the same task. Thus several methods can be used.

The SHOP algorithm nondeterministically chooses an applicable method.

This method is instantiated to decompose the task into (several) subtasks.

This goes on recursively.

The deterministic implementation of the SHOP algorithm uses depth-first backtracking: If the constraints on the subtasks prevent the plan from being feasible, then the implementation will backtrack and try other methods.
travel(UMD, MIT) → buy ticket(BWI, Logan) → travel(UMD, BWI) →
get taxi → ride taxi(UMD, BWI) → pay driver →
fly(BWI, Logan) → travel(Logan, MIT) →
get taxi → ride taxi(Logan, MIT) → pay driver
Beware!

- The planner may need to **recognise and resolve** interactions among the subtasks. (travel to airport: arrive in time)
- It is not always obvious **which method** to use.
- If it is not possible to solve the subtasks produced by one method, **SHOP will backtrack** and try another method instead.
SHOP (3 sorts of atoms)

- **Rigid Atoms:** Atoms whose truth values never change during planning. They appear in states, but not in the effects of planning operators nor in the heads of Horn clauses.

- **Primary Atoms:** Atoms that can appear in states and in the effects of planning operators, but **cannot appear in the heads of Horn clauses.**

- **Secondary Atoms:** These are the ones whose truth values are **inferred** rather than being stated explicitly. They **can appear in the heads** of Horn clauses, but cannot appear in states nor in the effects of planning operators.
**SHOP (States, axioms)**

**Definition 9.11 (States (S), Axioms (AX))**

A **state** S is a set of ground primary atoms. An **axiom** is an expression of the form

\[ a \leftarrow l_1, \ldots, l_n, \]

where \( a \) is a **secondary atom** and the \( l_1, \ldots, l_n \) are literals that constitute either **primary or secondary** atoms.

Axioms need not be ground.
SHOP starts with a state and modifies this state using add/delete lists.

Axioms are used only to check whether the preconditions of methods are satisfied.

A precondition might not be explicitly satisfied (an atom is not contained in \( S \)), but might be caused by \( S \) and the axioms.

The precise definition of this relation “caused by” is given as follows.
**Definition 9.12 (Literal caused by \((S, AX)\))**

A literal \(l\) is caused by \((S, AX)\) if \(l\) is true in the unique model of \(S \cup AX\).
A *task list* is a list of tasks, like the following:

```
```

A *ground task list* is a task list that consists of only ground tasks, like the following:

```
(!get-taxi umd) (!ride-taxi umd mit) (!pay-driver umd mit))
```
**SHOP (Operator)**

**Definition 9.13 (Operator: \((\text{Op } h \epsilon_{\text{del}} \epsilon_{\text{add}})\)**

An operator is an expression of the form \((\text{Op } h \epsilon_{\text{del}} \epsilon_{\text{add}})\), where \(h\) (the head) is a primitive task and \(\epsilon_{\text{add}}\) and \(\epsilon_{\text{del}}\) are lists of primary atoms (called the add- and delete-lists, respectively). The set of variables in the atoms in \(\epsilon_{\text{add}}\) and \(\epsilon_{\text{del}}\) must be a subset of the set of variables in \(h\).

- Unlike the operators used in action-based planning, ours have no preconditions.
- Preconditions are not needed for operators in our formulation, because they occur in the methods that invoke the operators.
As an example, here is a possible implementation of the `get-taxi` operator:

```
(:Op (!get-taxi ?x)
  ((taxi-called-to ?x))
  ((taxi-standing-at ?x)))
```
SHOP (Decomposing primitive tasks)

Operators are used in decomposition of primitive tasks during planning:

**Definition 9.14 (Decomposition of Primitive Tasks)**

Let $t$ be a primitive task, and let $\text{Op} = (\text{Op } h \, \epsilon_{\text{del}} \, \epsilon_{\text{add}})$ be an operator. Suppose that $\theta$ is a unifier for $h$ and $t$. Then the ground operator instance $(\text{Op})\theta$ is applicable to $t$, in which case we define the decomposition of $t$ by $\text{Op}$ to be $(\text{Op})\theta$. 
The *decomposition of a primitive task* by an operator results in a ground instance of that operator – i.e., it results in an *action that can be applied in a state* of the world. We now define the result of such an application.
A plan is a list of heads of ground operator instances. A plan $\pi$ is called a \textbf{simple plan} if it consists of the head of just one ground operator instance. Given a simple plan $\pi = (h)$, we define $\text{result}(S, \pi)$ to be the set $S \setminus \epsilon_{\text{del}} \cup \epsilon_{\text{add}}$, obtained by deleting from $S$ all atoms in $\epsilon_{\text{del}}$ and by adding all ground instances of atoms in $\epsilon_{\text{add}}$. If $\pi = (h_1, h_2, \ldots, h_n)$ is a plan and $S$ is a state, then the \textbf{result} of applying $\pi$ to $S$ is the state

$$\text{result}(S, \pi) = \text{result}(\text{result}(\ldots(\text{result}(S, h_1), h_2), \ldots), h_n).$$
In SHOP, a method specifies a possible way to accomplish a compound task.

The set of methods relevant for a particular compound task can be seen as a recursive definition of that task.
Definition 9.16 (Method: \((\text{Meth } h \rho t)\))

A **method** is an expression of the form \((\text{Meth } h \rho t)\) where \(h\) (the method’s **head**) is a compound task, \(\rho\) (the method’s **preconditions**) is a conjunction of literals and \(t\) is a totally-ordered list of subtasks, called the **decomposition list** of the method.

The set of variables that appear in the decomposition list of a method must be a subset of the variables in \(h\) (the head of the method) and \(\rho\) (the preconditions of the method).
Here is a possible implementation of the travel-by-taxi method:

(:Meth (travel ?x ?y)
  ((smaller-distance ?x ?y))
  (!get-taxi ?x) (!ride-taxi ?x ?y) (!pay-driver ?x ?y)))
**SHOP (Decomposing compound tasks)**

**Definition 9.17 (Decomposition of Compound Tasks)**

Let $t$ be a compound task, $S$ be the current state, $\text{Meth} = (\text{Meth \ h \ \rho \ t})$ be a method, and $\mathcal{AX}$ be an axiom set. Suppose that $\theta$ is a unifier for $h$ and $t$, and that $\theta'$ is a unifier such that all literals in $\rho \theta \theta'$ are caused wrt. $S$ and $\mathcal{AX}$ (see Definition 9.12).

Then, the ground method instance $(\text{Meth})\theta \theta'$ is applicable to $t$ in $S$, and the result of applying it to $t$ is the ground task list $r = (t)\theta \theta'$. The task list $r$ is the decomposition of $t$ by $\text{Meth}$ in $S$. 
SHOP (Planning Problem)

Definition 9.18 (Planning Domain Description)

A planning domain description $\mathcal{D}$ is a triple consisting of (1) a set of axioms, (2) a set of operators such that no two operators have the same head, and (3) a set of methods.

A planning problem is a triple $(S, t, \mathcal{D})$, where $S$ is a state, $t = (t_1, t_2, \ldots, t_k)$ is a ground task list, and $\mathcal{D}$ is a planning domain description.
Definition 9.19 (Solutions)

Let \( P = (S, t, D) \) be a planning problem and \( \pi = (h_1, h_2, \ldots, h_n) \) be a plan. Then, \( \pi \) is a solution for \( P \), if any of the following is true:

- **Case 1:** \( t \) and \( \pi \) are both empty, (i.e., \( k = 0 \) and \( n = 0 \));
- **Case 2:** \( t = (t_1, t_2, \ldots, t_k) \), \( t_1 \) is a ground primitive task, \( (h_1) \) is the decomposition of \( t_1 \), and \( (h_2 \ldots h_n) \) solves \( (\text{result}(S, (h_1)), (t_2, \ldots, t_k), D) \);
- **Case 3:** \( t = (t_1, t_2, \ldots, t_k) \), \( t_1 \) is a ground compound task, and there is a decomposition \( (r_1 \ldots r_j) \) of \( t_1 \) in \( S \) such that \( \pi \) solves \( (S, (r_1, \ldots, r_j, t_2, \ldots, t_k), D) \).

The planning problem \( (S, t, D) \) is solvable if there is a plan that solves it.
Edge labellings $m_i(t)$ (resp. $o(t)$) represent a method (resp. an operator) application to a task $t$, which is compound (resp. primitive).
9.6 Extensions
1. A plan can fail because of the following reasons:

- Actions may have *unexpected effects*, but these can be enumerated (as a disjunction).
- The *unexpected effects* are known. Then we have to replan.
function \textsc{replanning-agent}(\textit{percept}) \textbf{returns} an action

\textbf{static:} $KB$, a knowledge base (includes action descriptions)

$p$, an annotated plan, initially $NoPlan$

$q$, an annotated plan, initially $NoPlan$

$G$, a goal

\textsc{tell}(\textit{kb}, \textsc{make-percept-sentence}(\textit{percept}, t))

current $\leftarrow$ \textsc{state-description}(\textit{kb}, t)

\textbf{if} $p$ = $NoPlan$ \textbf{then}

\textbf{if} $p$ = $NoPlan$ or $p$ is empty \textbf{then} \textbf{return} $NoOp$

\textbf{if} \textsc{preconditions}(\textit{p}) not currently true in $KB$ \textbf{then}

\textbf{if} $p' \leftarrow$ \textsc{choose-best-continuation}(\textit{current}, $q$)

$p' \leftarrow$ \textsc{append}(\textsc{planner}(\textit{current}, \textsc{preconditions}(p'), kb), p')

$q' \leftarrow p$

\textit{action} $\leftarrow$ \textsc{first}(p)

$p' \leftarrow$ \textsc{rest}(p)

\textbf{return} \textit{action}

\textit{I.e. we perceive} and then \textbf{plan only if something has changed.}
2. Combine **replanning** and **conditional planning**. Planning and execution are integrated.