Artificial Intelligence 1
Winter Semester 2024/25

– Lecture Notes –
Part I: Getting Started with AI

Prof. Dr. Michael Kohlhase

Professur für Wissensrepräsentation und -verarbeitung
Informatik, FAU Erlangen-Nürnberg
Michael.Kohlhase@FAU.de

2025-02-06

This document contains Part I of the course notes for the course "Artificial Intelligence 1" held at FAU Erlangen-Nürnberg in the Winter Semesters 2016/17 ff. This part of the lecture notes sets the stage for the technical parts of the course by establishing a common framework (Rational Agents) that gives context and ties together the various methods discussed in the course. Other parts of the lecture notes can be found at `http://kwarc.info/teaching/AI/notes-*.pdf`.

Contents

After having seen what AI can do and where AI is being employed today (see ??), we will now

1. introduce a programming language to use in the course,
2. prepare a conceptual framework in which we can think about "intelligence" (natural and artificial), and
3. recap some methods and results from theoretical computer science that we will need throughout the course.

ad 1. Prolog: For the programming language we choose Prolog, historically one of the most influential "AI programming languages". While the other AI programming language: `Lisp` which gave rise to the functional programming programming paradigm has been superseded by typed languages like SML, Haskell, `Scala`, and F#, Prolog is still the prime example of the declarative programming paradigm. So using Prolog in this course gives students the opportunity to explore this paradigm. At the same time, Prolog is well-suited for trying out algorithms in symbolic AI the topic of this semester since it internalizes the more complex primitives of the algorithms presented here.

ad 2. Rational Agents: The conceptual framework centers around rational agents which combine aspects of purely cognitive architectures (an original concern for the field of AI) with the more recent realization that intelligence must interact with the world (embodied AI) to grow and learn. The cognitive architectures aspect allows us to place and relate the various algorithms and methods we will see in this course. Unfortunately, the "situated AI" aspect will not be covered in this course due to the lack of time and hardware.

ad 3. Topics of Theoretical Computer Science: When we evaluate the methods and algorithms introduced in AI-1, we will need to judge their suitability as agent functions. The main theoretical tool for that is complexity theory; we will give a short motivation and overview of the main methods and results as far as they are relevant for AI-1 in ??.

In the second half of the semester we will transition from search-based methods for problem solving to inference-based ones, i.e. where the problem formulation is described as expressions of a formal language which are transformed until an expression is reached from which the solution can be read off. Phrase structure grammars are the method of choice for describing such languages; we will introduce/recap them in ??.

Enough philosophy about "Intelligence" (Artificial or Natural)

- ▷ So far we had a nice philosophical chat, about "intelligence" et al.
- ▷ As of today, we look at technical stuff!
- ▷ Before we go into the algorithms and data structures proper, we will
  1. introduce a programming language for AI-1
  2. prepare a conceptual framework in which we can think about "intelligence" (natural and artificial), and
  3. recap some methods and results from theoretical computer science.

Chapter 3

Logic Programming

We will now learn a new programming paradigm: logic programming, which is one of the most influential paradigms in AI. We are going to study Prolog (the oldest and most widely used) as a concrete example of ideas behind logic programming and use it for our homeworks in this course.

As Prolog is a representative of a programming paradigm that is new to most students, programming will feel weird and tedious at first. But subtracting the unusual syntax and program organization logic programming really only amounts to recursive programming just as in functional programming (the other declarative programming paradigm). So the usual advice applies, keep staring at it and practice on easy examples until the pain goes away.

3.1 Introduction to Logic Programming and ProLog

Logic programming is a programming paradigm that differs from functional and imperative programming in the basic procedural intuition. Instead of transforming the state of the memory by issuing instructions (as in imperative programming), or computing the value of a function on some arguments, logic programming interprets the program as a body of knowledge about the respective situation, which can be queried for consequences.

This is actually a very natural conception of program; after all we usually run (imperative or functional) programs if we want some question answered. **Video Nuggets** covering this section can be found at `https://fau.tv/clip/id/21752` and `https://fau.tv/clip/id/21753`.

.

Logic Programming

▷ **Idea:** Use logic as a programming language!

▷ We state what we know about a problem (the program) and then ask for results (what the program would compute).

▷ **Example 3.1.1.**

Program	Leibniz is human Sokrates is human Sokrates is a greek Every human is fallible	$x + 0 = x$ If $x + y = z$ then $x + s(y) = s(z)$ 3 is prime
Query	Are there fallible greeks?	is there a z with $s(s(0)) + s(0) = z$
Answer	Yes, Sokrates!	yes $s(s(s(0)))$

▷ **How to achieve this?** Restrict a logic calculus sufficiently that it can be used as computational procedure.

▷ **Remark:** This idea leads a totally new programming paradigm: logic programming.

▷ **Slogan:** Computation = Logic + Control (Robert Kowalski 1973; [Kow97])

▷ We will use the programming language Prolog as an example.

We now formally define the language of Prolog, starting off the atomic building blocks.

Prolog Terms and Literals

▷ **Definition 3.1.2.** Prolog expresses knowledge about the world via

- ▷ constants denoted by lowercase strings,
- ▷ variables denoted by strings starting with an uppercase letter or _, and
- ▷ functions and predicates (lowercase strings) applied to terms.

▷ **Definition 3.1.3.** A Prolog term is

- ▷ a Prolog variable, or constant, or
- ▷ a Prolog function applied to terms.

A Prolog literal is a constant or a predicate applied to terms.

▷ **Example 3.1.4.** The following are

- ▷ Prolog terms: john, X, _, father(john), ...
- ▷ Prolog literals: loves(john,mary), loves(john,_), loves(john,wife_of(john)),...

Now we build up Prolog programs from those building blocks.

Prolog Programs: Facts and Rules

▷ **Definition 3.1.5.** A Prolog program is a sequence of clauses, i.e.

- ▷ facts of the form l., where l is a literal, (a literal and a dot)
- ▷ rules of the form h:−b_1,...,b_n., where $n > 0$. h is called the head literal (or simply head) and the b_i are together called the body of the rule.

A rule h:−b_1,...,b_n., should be read as *h (is true) if b_1 and ... and b_n are*.

▷ **Example 3.1.6.** Write "something is a car if it has a motor and four wheels" as car(X) :− has_motor(X),has_wheels(X,4). (variables are uppercase)
This is just an ASCII notation for $m(x) \wedge w(x,4) \Rightarrow car(x)$.

▷ **Example 3.1.7.** The following is a Prolog program:

```
human(leibniz).
human(sokrates).
```

```
greek(sokrates).
fallible(X):−human(X).
```

The first three lines are Prolog facts and the last a rule.

The whole point of writing down a knowledge base (a Prolog program with knowledge about the situation), if we do not have to write down *all* the knowledge, but a (small) subset, from which the rest follows. We have already seen how this can be done: with logic. For logic programming we will use a logic called "first-order logic" which we will not formally introduce here.

Prolog Programs: Knowledge bases

- **Intuition:** The knowledge base given by a Prolog program is the set of facts that can be derived from it under the if/and reading above.
- **Definition 3.1.8.** The knowledge base given by Prolog program is that set of facts that can be derived from it by Modus Ponens (MP), $\wedge I$ and instantiation.

$$\frac{A \quad A \Rightarrow B}{B}\,\mathrm{MP} \qquad \frac{A \quad B}{A \wedge B}\,\wedge I \qquad \frac{\mathbf{A}}{[\mathbf{B}/X](\mathbf{A})}\,\mathrm{Subst}$$

?? introduces a very important distinction: that between a Prolog program and the knowledge base it induces. Whereas the former is a finite, syntactic object (essentially a string), the latter may be an infinite set of facts, which represents the totality of knowledge about the world or the aspects described by the program.
As knowledge bases can be infinite, we cannot pre-compute them. Instead, logic programming languages compute fragments of the knowledge base by need; i.e. whenever a user wants to check membership; we call this approach querying: the user enters a query expression and the system answers yes or no. This answer is computed in a depth first search process.

Querying the Knowledge Base: Size Matters

- **Idea:** We want to see whether a fact is in the knowledge base.
- **Definition 3.1.9.** A query is a list of Prolog literals called goal literal (also subgoals or simply goals). We write a query as $?{-}A_1, \ldots, A_n.$ where A_i are goals.
- **Problem:** Knowledge bases can be big and even infinite. (cannot pre-compute)
- **Example 3.1.10.** The knowledge base induced by the Prolog program

```
nat(zero).
nat(s(X)) :− nat(X).
```

contains the facts nat(zero), nat(s(zero)), nat(s(s(zero))), ...

Querying the Knowledge Base: Backchaining

▷ **Definition 3.1.11.** Given a query Q: ?− $A_1, \ldots, A_n$. and rule R: h:− b_1,. . .,b_n, backchaining computes a new query by

1. finding terms for all variables in h to make h and A_1 equal and
2. replacing A_1 in Q with the body literals of R, where all variables are suitably replaced.

▷ Backchaining motivates the names goal/subgoal:

- ▷ the literals in the query are "goals" that have to be satisfied,
- ▷ backchaining does that by replacing them by new "goals".

▷ **Definition 3.1.12.** The Prolog interpreter keeps backchaining from the top to the bottom of the program until the query

- ▷ succeeds, i.e. contains no more goals, or (answer: **true**)
- ▷ fails, i.e. backchaining becomes impossible. (answer: false)

▷ **Example 3.1.13 (Backchaining).** We continue ??

```
?− nat(s(s(zero))).
?− nat(s(zero)).
?− nat(zero).
true
```


Note that backchaining replaces the current query with the body of the rule suitably instantiated. For rules with a long body this extends the list of current goals, but for facts (rules without a body), backchaining shortens the list of current goals. Once there are no goals left, the Prolog interpreter finishes and signals success by issuing the string **true**.

If no rules match the current subgoal, then the interpreter terminates and signals failure with the string false,

Querying the Knowledge Base: Failure

▷ If no instance of a query can be derived from the knowledge base, then the Prolog interpreter reports failure.

▷ **Example 3.1.14.** We vary ?? using 0 instead of zero.

```
?− nat(s(s(0))).
?− nat(s(0)).
?− nat(0).
FAIL
false
```


We can extend querying from simple yes/no answers to programs that return values by simply using variables in queries. In this case, the Prolog interpreter returns a substitution.

Querying the Knowledge base: Answer Substitutions

- **Definition 3.1.15.** If a query contains variables, then Prolog will return an answer substitution as the result to the query, i.e the values for all the query variables accumulated during repeated backchaining.
- **Example 3.1.16.** We talk about (Bavarian) cars for a change, and use a query with a variables

```
has_wheels(mybmw,4).
has_motor(mybmw).
car(X):−has_wheels(X,4),has_motor(X).
?− car(Y) % query
?− has_wheels(Y,4),has_motor(Y). % substitution X = Y
?− has_motor(mybmw). % substitution Y = mybmw
Y = mybmw % answer substitution
true
```


In ?? the first backchaining step binds the variable X to the query variable Y, which gives us the two subgoals has_wheels(Y,4),has_motor(Y). which again have the query variable Y. The next backchaining step binds this to mybmw, and the third backchaining step exhausts the subgoals. So the query succeeds with the (overall) answer substitution Y = mybmw. With this setup, we can already do the “fallible Greeks” example from the introduction.

PROLOG: Are there Fallible Greeks?

- **Program:**

```
human(leibniz).
human(sokrates).
greek(sokrates).
fallible(X):−human(X).
```

- **Example 3.1.17 (Query).** ?−fallible(X),greek(X).
- **Answer substitution:** $[\text{sokrates}/X]$

3.2 Programming as Search

In this section, we want to really use Prolog as a programming language, so let use first get our tools set up. **Video Nuggets** covering this section can be found at `https://fau.tv/clip/id/21754` and `https://fau.tv/clip/id/21827`.

3.2.1 Running Prolog

We will now discuss how to use a Prolog interpreter to get to know the language. The SWI Prolog interpreter can be downloaded from `http://www.swi-prolog.org/`. To start the Prolog interpreter with pl or prolog or swipl from the shell. The SWI manual is available at `http://www.swi-prolog.org/pldoc/`

We will introduce working with the interpreter using unary natural numbers as examples: we first add the fact[1] to the knowledge base

```
unat(zero).
```

which asserts that the predicate unat[2] is **true** on the term zero. Generally, we can add a fact to the knowledge base either by writing it into a file (e.g. example.pl) and then "consulting it" by writing one of the following three commands into the interpreter:

```
[example]
consult('example.pl').
consult('example').
```

or by directly typing

```
assert(unat(zero)).
```

into the Prolog interpreter. Next tell Prolog about the following rule

```
assert(unat(suc(X)) :- unat(X)).
```

which gives the Prolog runtime an initial (infinite) knowledge base, which can be queried by

```
?- unat(suc(suc(zero))).
```

Even though we can use any text editor to program Prolog, but running Prolog in a modern editor with language support is incredibly nicer than at the command line, because you can see the whole history of what you have done. Its better for debugging too.

3.2.2 Knowledge Bases and Backtracking

Depth-First Search with Backtracking

- So far, all the examples led to direct success or to failure. (simple KB)
- **Definition 3.2.1 (Prolog Search Procedure).** The Prolog interpreter employs top-down, left-right depth first search, concretely, Prolog search:
  - works on the subgoals in left right order.
  - matches first query with the head literals of the clauses in the program in top-down order.
  - if there are no matches, fail and backtracks to the (chronologically) last backtrack point.
  - otherwise backchain on the first match, keep the other matches in mind for backtracking via backtrack points.

  We say that a goal G matches a head H, iff we can make them equal by replacing variables in H with terms.
- We can force backtracking to compute more answers by typing ;.

Note: With the Prolog search procedure detailed above, computation can easily go into infinite loops, even though the knowledge base could provide the correct answer. Consider for instance the simple program

[1] for "unary natural numbers"; we cannot use the predicate nat and the constructor function s here, since their meaning is predefined in Prolog

[2] for "unary natural numbers".

```
p(X):- p(X).
p(X):- q(X).
q(X).
```

If we query this with ?- p(john), then DFS will go into an infinite loop because Prolog expands by default the first predicate. However, we can conclude that p(john) is true if we start expanding the second predicate.

In fact this is a necessary feature and not a bug for a programming language: we need to be able to write non-terminating programs, since the language would not be Turing complete otherwise. The argument can be sketched as follows: we have seen that for Turing machines the halting problem is undecidable. So if all Prolog programs were terminating, then Prolog would be weaker than Turing machines and thus not Turing complete.

We will now fortify our intuition about the Prolog search procedure by an example that extends the setup from ?? by a new choice of a vehicle that could be a car (if it had a motor).

Backtracking by Example

▷ **Example 3.2.2.** We extend ??:

```
has_wheels(mytricycle,3).
has_wheels(myrollerblade,3).
has_wheels(mybmw,4).
has_motor(mybmw).
car(X):-has_wheels(X,3),has_motor(X). % cars sometimes have three wheels
car(X):-has_wheels(X,4),has_motor(X). % and sometimes four.
?- car(Y).
?- has_wheels(Y,3),has_motor(Y). % backtrack point 1
Y = mytricycle % backtrack point 2
?- has_motor(mytricycle).
FAIL % fails, backtrack to 2
Y = myrollerblade % backtrack point 2
?- has_motor(myrollerblade).
FAIL % fails, backtrack to 1
?- has_wheels(Y,4),has_motor(Y).
Y = mybmw
?- has_motor(mybmw).
Y=mybmw
true
```


In general, a Prolog rule of the form A:−B,C reads as *A, if B and C*. If we want to express *A if B or C*, we have to express this two separate rules A:−B and A:−C and leave the choice which one to use to the search procedure.

In ?? we indeed have two clauses for the predicate car/1; one each for the cases of cars with three and four wheels. As the three-wheel case comes first in the program, it is explored first in the search process.

Recall that at every point, where the Prolog interpreter has the choice between two clauses for a predicate, chooses the first and leaves a backtrack point. In ?? this happens first for the predicate car/1, where we explore the case of three-wheeled cars. The Prolog interpreter immediately has to choose again – between the tricycle and the rollerblade, which both have three wheels. Again, it chooses the first and leaves a backtrack point. But as tricycles do not have motors, the subgoal has_motor(mytricycle) fails and the interpreter backtracks to the chronologically nearest backtrack point (the second one) and tries to fulfill has_motor(myrollerblade). This fails again, and the next backtrack point is point 1 – note the stack-like organization of backtrack points which is in keeping with the depth-first search strategy – which chooses the case of four-wheeled cars. This ultimately succeeds as before with y=mybmw.

3.2.3 Programming Features

We now turn to a more classical programming task: computing with numbers. Here we turn to our initial example: adding unary natural numbers. If we can do that, then we have to consider Prolog a programming language.

Can We Use This For Programming?

- **Question:** What about functions? E.g. the addition function?
- **Question:** We cannot define functions, in Prolog!
- **Idea (back to math):** use a three-place predicate.
- **Example 3.2.3.** add(X,Y,Z) stands for X+Y=Z
- Now we can directly write the recursive equations $X+0=X$ (base case) and $X+s(Y)=s(X+Y)$ into the knowledge base.

```
add(X,zero,X).
add(X,s(Y),s(Z)) :− add(X,Y,Z).
```

- Similarly with multiplication and exponentiation.

```
mult(X,zero,zero).
mult(X,s(Y),Z) :− mult(X,Y,W), add(X,W,Z).

expt(X,zero,s(zero)).
expt(X,s(Y),Z) :− expt(X,Y,W), mult(X,W,Z).
```


Note: Viewed through the right glasses logic programming is very similar to functional programming; the only difference is that we are using $n+1$ ary relations rather than n ary function. To see how this works let us consider the addition function/relation example above: instead of a binary function + we program a ternary relation add, where relation add(X,Y,Z) means $X+Y=Z$. We start with the same defining equations for addition, rewriting them to relational style.

The first equation is straight-forward via our correspondence and we get the Prolog fact add(X,zero,X). For the equation $X+s(Y)=s(X+Y)$ we have to work harder, the straight-forward relational translation add(X,s(Y),s(X+Y)) is impossible, since we have only partially replaced the function + with the relation add. Here we take refuge in a very simple trick that we can always do in logic (and mathematics of course): we introduce a new name Z for the offending expression $X+Y$ (using a variable) so that we get the fact add(X,s(Y),s(Z)). Of course this is not universally true (remember that this fact would say that "$X+s(Y)=s(Z)$ for all X, Y, and Z"), so we have to extend it to a Prolog rule add(X,s(Y),s(Z)):−add(X,Y,Z). which relativizes to mean "$X+s(Y)=s(Z)$ for all X, Y, and Z with $X+Y=Z$".

Indeed the rule implements addition as a recursive predicate, we can see that the recursion relation is terminating, since the left hand sides have one more constructor for the successor function. The examples for multiplication and exponentiation can be developed analogously, but we have to use the naming trick twice.

We now apply the same principle of recursive programming with predicates to other examples to reinforce our intuitions about the principles.

More Examples from elementary Arithmetic

▷ **Example 3.2.4.** We can also use the add relation for subtraction without changing the implementation. We just use variables in the "input positions" and ground terms in the other two. (possibly very inefficient "generate and test approach")

```
?−add(s(zero),X,s(s(s(zero)))).
X = s(s(zero))
true
```

▷ **Example 3.2.5.** Computing the n^{th} Fibonacci number (0, 1, 1, 2, 3, 5, 8, 13,...; add the last two to get the next), using the addition predicate above.

```
fib(zero,zero).
fib(s(zero),s(zero)).
fib(s(s(X)),Y):−fib(s(X),Z),fib(X,W),add(Z,W,Y).
```

▷ **Example 3.2.6.** Using Prolog's internal floating-point arithmetic: a goal of the form ?− D **is** e. — where e is a ground arithmetic expression binds D to the result of evaluating e.

```
fib(0,0).
fib(1,1).
fib(X,Y):− D is X − 1, E is X − 2,fib(D,Z),fib(E,W), Y is Z + W.
```


Note: Note that the **is** relation does not allow "generate and test" inversion as it insists on the right hand being ground. In our example above, this is not a problem, if we call the fib with the first ("input") argument a ground term. Indeed, it matches the last rule with a goal ?− g,Y., where g is a ground term, then g−1 and g−2 are ground and thus D and E are bound to the (ground) result terms. This makes the input arguments in the two recursive calls ground, and we get ground results for Z and W, which allows the last goal to succeed with a ground result for Y. Note as well that re-ordering the bodys literal of the rule so that the recursive calls are called before the computation literals will lead to failure.

We will now add the primitive data structure of lists to Prolog; they are constructed by prepending an element (the head) to an existing list (which becomes the rest list or "tail" of the constructed one).

Adding Lists to Prolog

▷ **Definition 3.2.7.** In Prolog, lists are represented by list terms of the form

1. [a,b,c,...] for list literals, and
2. a first/rest constructor that represents a list with head F and rest list R as [F|R].

▷ **Observation:** Just as in functional programming, we can define list operations by recursion, only that we program with relations instead of with functions.

▷ **Example 3.2.8.** Predicates for member, append and reverse of lists in default Prolog representation.

```
member(X,[X|_]).
member(X,[_|R]):−member(X,R).

append([],L,L).
append([X|R],L,[X|S]):−append(R,L,S).
```

```
reverse([],[]).
reverse([X|R],L):−reverse(R,S),append(S,[X],L).
```


Logic programming is the third large programming paradigm (together with functional programming and imperative programming).

Relational Programming Techniques

▷ **Example 3.2.9.** Parameters have no unique direction "in" or "out"

```
?− rev(L,[1,2,3]).
?− rev([1,2,3],L1).
?− rev([1|X],[2|Y]).
```

▷ **Example 3.2.10.** Symbolic programming by structural induction:

```
rev([],[]).
rev([X|Xs],Ys) :− ...
```

▷ **Example 3.2.11.** Generate and test:

```
sort(Xs,Ys) :− perm(Xs,Ys), ordered(Ys).
```


From a programming practice point of view it is probably best understood as "relational programming" in analogy to functional programming, with which it shares a focus on recursion.
The major difference to functional programming is that "relational programming" does not have a fixed input/output distinction, which makes the control flow in functional programs very direct and predictable. Thanks to the underlying search procedure, we can sometime make use of the flexibility afforded by logic programming.
If the problem solution involves search (and depth first search is sufficient), we can just get by with specifying the problem and letting the Prolog interpreter do the rest. In **??** we just specify that list Xs can be sorted into Ys, iff Ys is a permutation of Xs and Ys is ordered. Given a concrete (input) list Xs, the Prolog interpreter will generate all permutations of Ys of Xs via the predicate perm/2 and then test them whether they are ordered.

This is a paradigmatic example of logic programming. We can (sometimes) directly use the specification of a problem as a program. This makes the argument for the correctness of the program immediate, but may make the program execution non optimal.

3.2.4 Advanced Relational Programming

It is easy to see that the running time of the Prolog program from **??** is not $\mathcal{O}(n\log_2(n))$ which is optimal for sorting algorithms. This is the flip side of the flexibility in logic programming. But Prolog has ways of dealing with that: the cut operator, which is a Prolog atom, which always succeeds, but which cannot be backtracked over. This can be used to prune the search tree in Prolog. We will not go into that here but refer the readers to the literature.

Specifying Control in Prolog

▷ *Remark 3.2.12.* The running time of the program from **??** is not $\mathcal{O}(n\log_2(n))$

which is optimal for sorting algorithms.

```
sort(Xs,Ys) :− perm(Xs,Ys), ordered(Ys).
```

▷ **Idea:** Gain computational efficiency by shaping the search!

Functions and Predicates in Prolog

▷ *Remark 3.2.13.* Functions and predicates have radically different roles in Prolog.

- ▷ Functions are used to represent data. (e.g. father(john) or s(s(zero)))
- ▷ Predicates are used for stating properties about and computing with data.

▷ *Remark 3.2.14.* In functional programming, functions are used for both. (even more confusing than in Prolog if you think about it)

▷ **Example 3.2.15.** Consider again the reverse predicate for lists below:
An input datum is e.g. [1,2,3], then the output datum is [3,2,1].

```
reverse([],[]).
reverse([X|R],L):−reverse(R,S),append(S,[X],L).
```

We "define" the computational behavior of the predicate rev, but the list constructors [. . .] are just used to construct lists from arguments.

▷ **Example 3.2.16 (Trees and Leaf Counting).** We represent (unlabelled) trees via the function t from tree lists to trees. For instance, a balanced binary tree of depth 2 is t([t([t([]),t([])]),t([t([]),t([])])]). We count leaves by

```
leafcount(t([]),1).
leafcount(t([V]),W) :− leafcount(V,W).
leafcount(t([X|R]),Y) :− leafcount(X,Z), leafcount(t(R),W), Y is Z + W.
```


For more information on Prolog

RTFM ($\widehat{=}$ "read the fine manuals")

▷ **RTFM Resources:** There are also lots of good tutorials on the web,

- ▷ I personally like [Fis; LPN],
- ▷ [Fla94] has a very thorough logic-based introduction,

▷ consult also the SWI Prolog Manual [SWI],

Chapter 4

Recap of Prerequisites from Math & Theoretical Computer Science

In this chapter we will briefly recap some of the prerequisites from theoretical computer science that are needed for understanding Artificial Intelligence 1.

4.1 Recap: Complexity Analysis in AI?

We now come to an important topic which is not really part of Artificial Intelligence but which adds an important layer of understanding to this enterprise: We (still) live in the era of Moore's law (the computing power available on a single CPU doubles roughly every two years) leading to an exponential increase. A similar rule holds for main memory and disk storage capacities. And the production of computer (using CPUs and memory) is (still) very rapidly growing as well; giving mankind as a whole, institutions, and individual exponentially grow of computational resources.

In public discussion, this development is often cited as the reason why (strong) AI is inevitable. But the argument is fallacious if all the algorithms we have are of very high complexity (i.e. at least exponential in either time or space). So, to judge the state of play in Artificial Intelligence, we have to know the complexity of our algorithms.

In this section, we will give a very brief recap of some aspects of elementary complexity theory and make a case of why this is a generally important for computer scientists.

A Video Nugget covering this section can be found at `https://fau.tv/clip/id/21839` and `https://fau.tv/clip/id/21840`.

To get a feeling what we mean by "fast algorithm", we do some preliminary computations.

Performance and Scaling

- ▷ Suppose we have three algorithms to choose from. (which one to select)
- ▷ Systematic analysis reveals performance characteristics.
- ▷ **Example 4.1.1.** For a computational problem of size n we have

	performance		
size	linear	quadratic	exponential
n	$100n\mu s$	$7n^2\mu s$	$2^n\mu s$
1	$100\mu s$	$7\mu s$	$2\mu s$
5	$.5ms$	$175\mu s$	$32\mu s$
10	$1ms$	$.7ms$	$1ms$
45	$4.5ms$	$14ms$	$1.1Y$
100	...	...	...
1 000	...	...	...
10 000	...	...	...
1 000 000	...	...	...

The last number in the rightmost column may surprise you. Does the run time really grow that fast? Yes, as a quick calculation shows; and it becomes much worse, as we will see.

What?! One year?

- $2^{10} = 1\,024$ ($1024\mu s \simeq 1ms$)
- $2^{45} = 35\,184\,372\,088\,832$ ($3.5\times10^{13}\mu s \simeq 3.5\times10^{7}s \simeq 1.1Y$)
- **Example 4.1.2.** We denote all times that are longer than the age of the universe with –

	performance		
size	linear	quadratic	exponential
n	$100n\mu s$	$7n^2\mu s$	$2^n\mu s$
1	$100\mu s$	$7\mu s$	$2\mu s$
5	$.5ms$	$175\mu s$	$32\mu s$
10	$1ms$	$.7ms$	$1ms$
45	$4.5ms$	$14ms$	$1.1Y$
< 100	$100ms$	$7s$	$10^{16}Y$
1 000	$1s$	$12min$	–
10 000	$10s$	$20h$	–
1 000 000	$1.6min$	$2.5mon$	–

So it does make a difference for larger computational problems what algorithm we choose. Considerations like the one we have shown above are very important when judging an algorithm. These evaluations go by the name of "complexity theory".
Let us now recapitulate some notions of elementary complexity theory: we are interested in the worst-case growth of the resources (time and space) required by an algorithm in terms of the sizes of its arguments. Mathematically we look at the functions from input size to resource size and classify them into "big-O" classes, abstracting from constant factors (which depend on the machine thealgorithm runs on and which we cannot control) and initial (algorithm startup) factors.

Recap: Time/Space Complexity of Algorithms

- We are mostly interested in worst-case complexity in AI-1.

▷ **Definition 4.1.3.** We say that an algorithm α that terminates in time $t(n)$ for all inputs of size n has running time $T(\alpha) := t$.

Let $S \subseteq \mathbb{N} \to \mathbb{N}$ be a set of natural number functions, then we say that α has time complexity in S (written $T(\alpha) \in S$ or colloquially $T(\alpha) = S$), iff $t \in S$. We say α has space complexity in S, iff α uses only memory of size $s(n)$ on inputs of size n and $s \in S$.

▷ Time/space complexity depends on size measures. (no canonical one)

▷ **Definition 4.1.4.** The following sets are often used for S in $T(\alpha)$:

Landau set	class name	rank	Landau set	class name	rank
$\mathcal{O}(1)$	constant	1	$\mathcal{O}(n^2)$	quadratic	4
$\mathcal{O}(\log_2(n))$	logarithmic	2	$\mathcal{O}(n^k)$	polynomial	5
$\mathcal{O}(n)$	linear	3	$\mathcal{O}(k^n)$	exponential	6

where $\mathcal{O}(g) = \{f \mid \exists k > 0. f \leq_a k \cdot g\}$ and $f \leq_a g$ (f is asymptotically bounded by g), iff there is an $n_0 \in \mathbb{N}$, such that $f(n) \leq g(n)$ for all $n > n_0$.

▷ **Lemma 4.1.5 (Growth Ranking).** *For $k' > 2$ and $k > 1$ we have*

$$\mathcal{O}(1) \subset \mathcal{O}(\log_2(n)) \subset \mathcal{O}(n) \subset \mathcal{O}(n^2) \subset \mathcal{O}(n^{k'}) \subset \mathcal{O}(k^n)$$

▷ **For AI-1:** I expect that given an algorithm, you can determine its complexity class. (next)

Advantage: Big-Oh Arithmetics

▷ **Practical Advantage:** Computing with Landau sets is quite simple. (good simplification)

▷ **Theorem 4.1.6 (Computing with Landau Sets).**

1. *If $\mathcal{O}(c \cdot f) = \mathcal{O}(f)$ for any constant $c \in \mathbb{N}$.* (drop constant factors)
2. *If $\mathcal{O}(f) \subseteq \mathcal{O}(g)$, then $\mathcal{O}(f + g) = \mathcal{O}(g)$.* (drop low-complexity summands)
3. *If $\mathcal{O}(f \cdot g) = \mathcal{O}(f) \cdot \mathcal{O}(g)$.* (distribute over products)

▷ These are not all of “big-Oh calculation rules”, but they're enough for most purposes

▷ **Applications:** Convince yourselves using the result above that

- ▹ $\mathcal{O}(4n^3 + 3n + 7^{1000n}) = \mathcal{O}(2^n)$
- ▹ $\mathcal{O}(n) \subset \mathcal{O}(n \cdot \log_2(n)) \subset \mathcal{O}(n^2)$

OK, that was the theory, ... but how do we use that in practice?

What I mean by this is that given an algorithm, we have to determine the time complexity.

This is by no means a trivial enterprise, but we can do it by analyzing the algorithm instruction by instruction as shown below.

Determining the Time/Space Complexity of Algorithms

▷ **Definition 4.1.7.** Given a function Γ that assigns variables v to functions $\Gamma(v)$ and α an imperative algorithm, we compute the

- ▷ time complexity $T_\Gamma(\alpha)$ of program α and
- ▷ the context $C_\Gamma(\alpha)$ introduced by α

by joint induction on the structure of α:

- ▷ constant: can be accessed in constant time
  If $\alpha = \delta$ for a data constant δ, then $T_\Gamma(\alpha) \in \mathcal{O}(1)$.
- ▷ variable: need the complexity of the value
  If $\alpha = v$ with $v \in \mathbf{dom}(\Gamma)$, then $T_\Gamma(\alpha) \in \mathcal{O}(\Gamma(v))$.
- ▷ application: compose the complexities of the function and the argument
  If $\alpha = \varphi(\psi)$ with $T_\Gamma(\varphi) \in \mathcal{O}(f)$ and $T_{\Gamma \cup C_\Gamma(\varphi)}(\psi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f \circ g)$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\varphi)}(\psi)$.
- ▷ assignment: has to compute the value $\rightsquigarrow$ has its complexity
  If α is $v{:=}\varphi$ with $T_\Gamma(\varphi) \in S$, then $T_\Gamma(\alpha) \in S$ and $C_\Gamma(\alpha) = \Gamma \cup (v,S)$.
- ▷ composition: has the maximal complexity of the components
  If α is $\varphi\,;\psi$, with $T_\Gamma(\varphi) \in P$ and $T_{\Gamma \cup C_\Gamma(\psi)}(\psi) \in Q$, then $T_\Gamma(\alpha) \in \max\,\{P, Q\}$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\psi)}(\psi)$.
- ▷ branching: has the maximal complexity of the condition and branches
  If α is **if**γ**then**φ**else**ψ**end**, with $T_\Gamma(\gamma) \in C$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in P$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in Q$, and then $T_\Gamma(\alpha) \in \max\,\{C, P, Q\}$ and $C_\Gamma(\alpha) = \Gamma \cup C_\Gamma(\gamma) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \cup C_{\Gamma \cup C_\Gamma(\gamma)}(\psi)$.
- ▷ looping: multiplies complexities
  If α is **while**γ**do**φ**end**, with $T_\Gamma(\gamma) \in \mathcal{O}(f)$, $T_{\Gamma \cup C_\Gamma(\gamma)}(\varphi) \in \mathcal{O}(g)$, then $T_\Gamma(\alpha) \in \mathcal{O}(f(n) \cdot g(n))$ and $C_\Gamma(\alpha) = C_{\Gamma \cup C_\Gamma(\gamma)}(\varphi)$.
- ▷ The time complexity $T(\alpha)$ is just $T_\emptyset(\alpha)$, where $\emptyset$ is the empty function.

▷ Recursion is much more difficult to analyze $\rightsquigarrow$ recurrences and Master's theorem.

As instructions in imperative programs can introduce new variables, which have their own time complexity, we have to carry them around via the introduced context, which has to be defined co-recursively with the time complexity. This makes ?? rather complex. The main two cases to note here are

- the variable case, which "uses" the context Γ and
- the assignment case, which extends the introduced context by the time complexity of the value.

The other cases just pass around the given context and the introduced context systematically.
Let us now put one motivation for knowing about complexity theory into the perspective of the job market; here the job as a scientist.

Please excuse the chemistry pictures, public imagery for CS is really just quite boring, this is what people think of when they say "scientist". So, imagine that instead of a chemist in a lab, it's me sitting in front of a computer.

Why Complexity Analysis? (General)

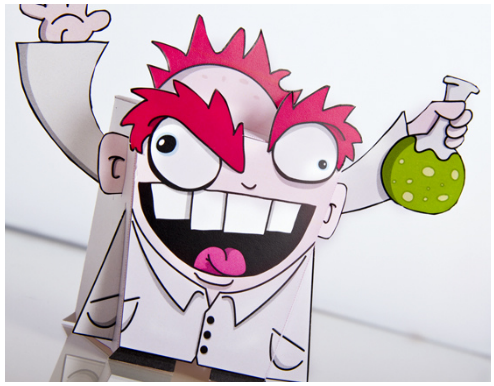
▷ **Example 4.1.8.** Once upon a time I was trying to invent an efficient algorithm.

- ▷ My first algorithm attempt didn't work, so I had to try harder.

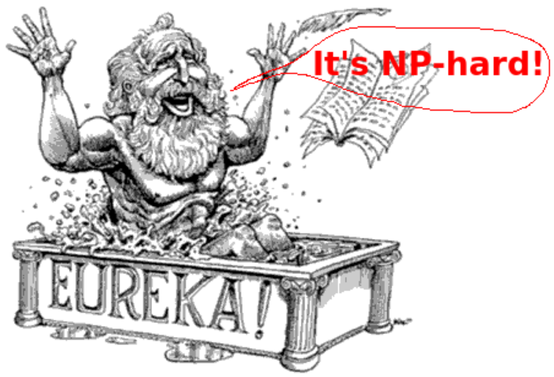
- ▷ But my 2nd attempt didn't work either, which got me a bit agitated.

- ▷ The 3rd attempt didn't work either...

- ▷ And neither the 4th. But then:



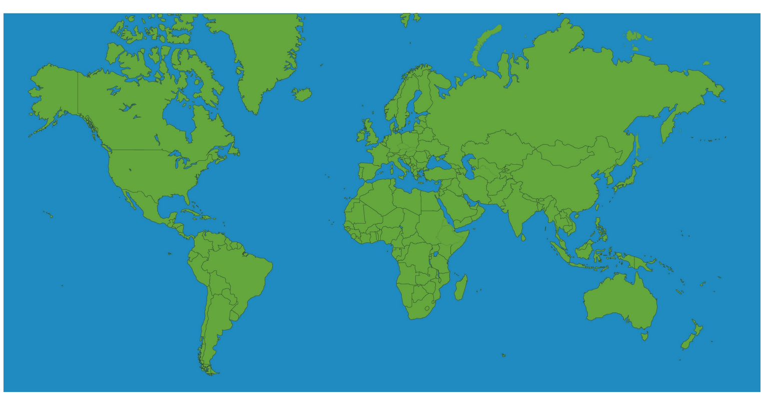
▷ Ta-da ... when, for once, I turned around and looked in the other direction–CAN one actually solve this efficiently? – NP hardness was there to rescue me.



The meat of the story is that there is no profit in trying to invent an algorithm, which we could have known that cannot exist. Here is another image that may be familiar to you.

Why Complexity Analysis? (General)

▷ **Example 4.1.9.** Trying to find a sea route east to India (from Spain) (does not exist)



▷ **Observation:** Complexity theory saves you from spending lots of time trying to invent algorithms that do not exist.

It's like, you're trying to find a route to India (from Spain), and you presume it's somewhere to the east, and then you hit a coast, but no; try again, but no; try again, but no; ... if you don't have a map, that's the best you can do. But NP hardness gives you the map: you can check that there actually is no way through here. But what is this notion of NP completeness alluded to above? We observe that we can analyze the complexity of problems by the complexity of the algorithms that solve them. This gives us a notion of what to expect from solutions to a given problem class, and thus whether efficient (i.e. polynomial time) algorithms can exist at all.

Reminder (?): $\mathbf{NP}$ and $\mathbf{PSPACE}$ (details ⇝ e.g. [GJ79])

- ▷ **Turing Machine:** Works on a tape consisting of cells, across which its Read/Write head moves. The machine has internal states. There is a transition function that specifies – given the current cell content and internal state – what the subsequent internal state will be, how what the R/W head does (write a symbol and/or move). Some internal states are accepting.
- ▷ Decision problems are in $\mathbf{NP}$ if there is a non deterministic Turing machine that halts with an answer after time polynomial in the size of its input. Accepts if *at least one* of the possible runs accepts.
- ▷ Decision problems are in $\mathbf{NPSPACE}$, if there is a non deterministic Turing machine that runs in space polynomial in the size of its input.
- ▷ **NP vs. PSPACE:** Non-deterministic polynomial space can be simulated in deterministic polynomial space. Thus $\mathbf{PSPACE} = \mathbf{NPSPACE}$, and hence (trivially) $\mathbf{NP} \subseteq \mathbf{PSPACE}$.

  It is commonly believed that $\mathbf{NP} \not\supseteq \mathbf{PSPACE}$. (similar to $\mathbf{P} \subseteq \mathbf{NP}$)

The Utility of Complexity Knowledge (NP-Hardness)

- ▷ **Assume:** In 3 years from now, you have finished your studies and are working in your first industry job. Your boss Mr. X gives you a problem and says *Solve It!*. By which he means, *write a program that solves it efficiently*.
- ▷ **Question:** Assume further that, after trying in vain for 4 weeks, you got the next meeting with Mr. X. How could knowing about NP hardness help?
- ▷ **Answer:** reserved for the plenary sessions ⇝ be there!

4.2 Recap: Formal Languages and Grammars

One of the main ways of designing rational agents in this course will be to define formal languages that represent the state of the agent environment and let the agent use various inference techniques to predict effects of its observations and actions to obtain a world model. In this section we recap the basics of formal languages and grammars that form the basis of a compositional theory for them.

The Mathematics of Strings

▷ **Definition 4.2.1.** An alphabet A is a finite set; we call each element $a \in A$ a character, and an n tuple $s \in A^n$ a string (of length n over A).

▷ **Definition 4.2.2.** Note that $A^0 = \{\langle\rangle\}$, where $\langle\rangle$ is the (unique) 0-tuple. With the definition above we consider $\langle\rangle$ as the string of length 0 and call it the empty string and denote it with ϵ.

▷ **Note:** Sets $\neq$ strings, e.g. $\{1, 2, 3\} = \{3, 2, 1\}$, but $\langle 1, 2, 3\rangle \neq \langle 3, 2, 1\rangle$.

▷ **Notation:** We will often write a string $\langle c_1, \ldots, c_n\rangle$ as "$\mathtt{c_1 \ldots c_n}$", for instance "abc" for $\langle \mathsf{a}, \mathsf{b}, \mathsf{c}\rangle$

▷ **Example 4.2.3.** Take $A = \{\mathtt{h}, \mathtt{1}, /\}$ as an alphabet. Each of the members $\mathtt{h}$, $\mathtt{1}$, and $/$ is a character. The vector $\langle /, /, \mathtt{1}, \mathtt{h}, \mathtt{1}\rangle$ is a string of length 5 over A.

▷ **Definition 4.2.4 (String Length).** Given a string s we denote its length with $|s|$.

▷ **Definition 4.2.5.** The concatenation $\mathrm{conc}(s, t)$ of two strings $s = \langle s_1, ..., s_n\rangle \in A^n$ and $t = \langle t_1, ..., t_m\rangle \in A^m$ is defined as $\langle s_1, ..., s_n, t_1, ..., t_m\rangle \in A^{n+m}$.

We will often write $\mathrm{conc}(s, t)$ as $s + t$ or simply st

▷ **Example 4.2.6.** $\mathrm{conc}(\text{"text"}, \text{"book"}) = \text{"text"} + \text{"book"} = \text{"textbook"}$

We have multiple notations for concatenation, since it is such a basic operation, which is used so often that we will need very short notations for it, trusting that the reader can disambiguate based on the context.

Now that we have defined the concept of a string as a sequence of characters, we can go on to give ourselves a way to distinguish between good strings (e.g. programs in a given programming language) and bad strings (e.g. such with syntax errors). The way to do this by the concept of a formal language, which we are about to define.

Formal Languages

▷ **Definition 4.2.7.** Let A be an alphabet, then we define the sets $A^+ := \bigcup_{i \in \mathbb{N}^+} A^i$ of nonempty string and $A^* := A^+ \cup \{\epsilon\}$ of strings.

▷ **Example 4.2.8.** If $A = \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, then $A^* = \{\epsilon, \mathsf{a}, \mathsf{b}, \mathsf{c}, \mathsf{aa}, \mathsf{ab}, \mathsf{ac}, \mathsf{ba}, \ldots, \mathsf{aaa}, \ldots\}$.

▷ **Definition 4.2.9.** A set $L \subseteq A^*$ is called a formal language over A.

▷ **Definition 4.2.10.** We use $\mathsf{c}^{[n]}$ for the string that consists of the character c repeated n times.

▷ **Example 4.2.11.** $\#^{[5]} = \langle \#, \#, \#, \#, \#\rangle$

▷ **Example 4.2.12.** The set $M := \{\mathsf{ba}^{[n]} \mid n \in \mathbb{N}\}$ of strings that start with character b followed by an arbitrary numbers of a's is a formal language over $A = \{\mathsf{a}, \mathsf{b}\}$.

▷ **Definition 4.2.13.** Let $L_1, L_2, L \subseteq \Sigma^*$ be formal languages over Σ.

- ▷ Intersection and union: $L_1 \cap L_2$, $L_1 \cup L_2$.
- ▷ Language complement L: $\overline{L} := \Sigma^* \backslash L$.
- ▷ The language concatenation of L_1 and L_2: $L_1 L_2 := \{uw \,|\, u \in L_1,\ w \in L_2\}$. We often use $L_1 L_2$ instead of $L_1 L_2$.
- ▷ Language power L: $L^0 := \{\epsilon\}$, $L^{n+1} := LL^n$, where $L^n := \{\mathtt{w_1} \ldots \mathtt{w_n} \,|\, w_i \in L$, for $i = 1 \ldots n\}$, (for $n \in \mathbb{N}$).
- ▷ language Kleene closure L: $L^* := \bigcup_{n \in \mathbb{N}} L^n$ and also $L^+ := \bigcup_{n \in \mathbb{N}^+} L^n$.
- ▷ The reflection of a language L: $L^R := \{w^R \,|\, w \in L\}$.

There is a common misconception that a formal language is something that is difficult to understand as a concept. This is not true, the only thing a formal language does is separate the “good” from the bad strings. Thus we simply model a formal language as a set of stings: the “good” strings are members, and the “bad” ones are not.

Of course this definition only shifts complexity to the way we construct specific formal languages (where it actually belongs), and we have learned two (simple) ways of constructing them: by repetition of characters, and by concatenation of existing languages. As mentioned above, the purpose of a formal language is to distinguish “good” from “bad” strings. It is maximally general, but not helpful, since it does not support computation and inference. In practice we will be interested in formal languages that have some structure, so that we can represent formal languages in a finite manner (recall that a formal language is a subset of A^*, which may be infinite and even undecidable – even though the alphabet A is finite).

To remedy this, we will now introduce phrase structure grammars (or just grammars), the standard tool for describing structured formal languages.

Phrase Structure Grammars (Theory)

▷ **Recap:** A formal language is an arbitrary set of symbol sequences.

▷ **Problem:** This may be infinite and even undecidable even if A is finite.

▷ **Idea:** Find a way of representing formal languages with structure finitely.

▷ **Definition 4.2.14.** A phrase structure grammar (also called type 0 grammar, unrestricted grammar, or just grammar) is a tuple $\langle N, \Sigma, P, S\rangle$ where

- ▷ N is a finite set of nonterminal symbols,
- ▷ Σ is a finite set of terminal symbols, members of $\Sigma \cup N$ are called symbols.
- ▷ P is a finite set of production rules: pairs $p := h \rightarrow b$ (also written as $h \Rightarrow b$), where $h \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $b \in (\Sigma \cup N)^*$. The string h is called the head of p and b the body.
- ▷ $S \in N$ is a distinguished symbol called the start symbol (also sentence symbol).

The sets N and Σ are assumed to be disjoint. Any word $w \in \Sigma^*$ is called a terminal word.

▷ **Intuition:** Production rules map strings with at least one nonterminal to arbitrary other strings.

▷ **Notation:** If we have n rules $h \to b_i$ sharing a head, we often write $h \to b_1 \mid \ldots \mid b_n$ instead.

We fortify our intuition about these – admittedly very abstract – constructions by an example and introduce some more vocabulary.

Phrase Structure Grammars (cont.)

▷ **Example 4.2.15.** A simple phrase structure grammar G:

$$\begin{array}{rcl} S & \to & NP\ Vi \\ NP & \to & Article\ N \\ Article & \to & \textbf{the} \mid \textbf{a} \mid \textbf{an} \\ N & \to & \textbf{dog} \mid \textbf{teacher} \mid \ldots \\ Vi & \to & \textbf{sleeps} \mid \textbf{smells} \mid \ldots \end{array}$$

Here S, is the start symbol, NP, $Article$, N, and Vi are nonterminals.

▷ **Definition 4.2.16.** A production rule whose head is a single non-terminal and whose body consists of a single terminal is called lexical or a lexical insertion rule.

Definition 4.2.17. The subset of lexical rules of a grammar G is called the lexicon of G and the set of body symbols the vocabulary (or alphabet). The nonterminals in their heads are called lexical categories of G.

▷ **Definition 4.2.18.** The non-lexicon production rules are called structural, and the nonterminals in the heads are called phrasal or syntactic categories.

Now we look at just how a grammar helps in analyzing formal languages. The basic idea is that a grammar accepts a word, iff the start symbol can be rewritten into it using only the rules of the grammar.

Phrase Structure Grammars (Theory)

▷ **Idea:** Each symbol sequence in a formal language can be analyzed/generated by the grammar.

▷ **Definition 4.2.19.** Given a phrase structure grammar $G := \langle N, \Sigma, P, S\rangle$, we say G derives $t \in (\Sigma \cup N)^*$ from $s \in (\Sigma \cup N)^*$ in one step, iff there is a production rule $p \in P$ with $p = h \to b$ and there are $u, v \in (\Sigma \cup N)^*$, such that $s = suhv$ and $t = ubv$. We write $s \to_G^p t$ (or $s \to_G t$ if p is clear from the context) and use $\to_G^*$ for the reflexive transitive closure of $\to_G$. We call $s \to^*{}_G t$ a G derivation of t from s.

TEST1: $\begin{array}{ccc} A & \to_G & B \\ C & \to_G & D \end{array}$

TEST2: $\begin{array}{lll} A & \to_G & B \\ & \to_G & C \\ & \to_G & D \end{array}$ TEST3: $\begin{array}{lll} s & \to_{G_2} & asb \\ & \to_{G_2} & aaSbb \\ & \to_{G_2} & aaaSbbb \\ & \to_{G_2} & aaaaSbbbb \\ & \to_{G_2} & aaaabbbb \end{array}$

▷ **Definition 4.2.20.** Given a phrase structure grammar $G := \langle N, \Sigma, P, S \rangle$, we say that $s \in (N \cup \Sigma)^*$ is a sentential form of G, iff $S {\to^*}_G s$. A sentential form that does not contain nontermials is called a sentence of G, we also say that G accepts s. We say that G rejects s, iff it is not a sentence of G.

▷ **Definition 4.2.21.** The language $\mathbf{L}(G)$ of G is the set of its sentences. We say that $\mathbf{L}(G)$ is generated by G.

Definition 4.2.22. We call two grammars equivalent, iff they have the same languages.

Definition 4.2.23. A grammar G is said to be universal if $\mathbf{L}(G) = \Sigma^*$.

▷ **Definition 4.2.24.** Parsing, syntax analysis, or syntactic analysis is the process of analyzing a string of symbols, either in a formal or a natural language by means of a grammar.

Again, we fortify our intuitions with ??.

Phrase Structure Grammars (Example)

▷ **Example 4.2.25.** In the grammar G from ??:

1. *Article* **teacher** *Vi* is a sentential form,

$$\begin{array}{lll} S & \to_G & NP\ Vi \\ & \to_G & Article\ N\ Vi \\ & \to_G & Article\ \mathbf{teacher}\ Vi \end{array}$$

2. *The teacher sleeps* is a sentence.

$$\begin{array}{lll} S & \to_G^* & Article\ \mathbf{teacher}\ Vi \\ & \to_G & \mathbf{the\ teacher}\ Vi \\ & \to_G & \mathbf{the\ teacher\ sleeps} \end{array}$$

$$\begin{array}{lll} S & \to & NP\ Vi \\ NP & \to & Article\ N \\ Article & \to & \mathbf{the} \mid \mathbf{a} \mid \mathbf{an} \mid \ldots \\ N & \to & \mathbf{dog} \mid \mathbf{teacher} \mid \ldots \\ Vi & \to & \mathbf{sleeps} \mid \mathbf{smells} \mid \ldots \end{array}$$

Note that this process indeed defines a formal language given a grammar, but does not provide an efficient algorithm for parsing, even for the simpler kinds of grammars we introduce below.

Grammar Types (Chomsky Hierarchy [Cho65])

▷ **Observation:** The shape of the grammar determines the "size" of its language.

▷ **Definition 4.2.26.** We call a grammar:

1. context-sensitive (or type 1), if the bodies of production rules have no less symbols than the heads,
2. context-free (or type 2), if the heads have exactly one symbol,
3. regular (or type 3), if additionally the bodies are empty or consist of a nonterminal, optionally followed by a terminal symbol.

By extension, a formal language L is called context-sensitive/context-free/regular (or type 1/type 2/type 3 respectively), iff it is the language of a respective grammar. Context-free grammars are sometimes CFGs and context-free languages CFLs.

▷ **Example 4.2.27 (Context-sensitive).** The language $\{a^{[n]}b^{[n]}c^{[n]}\}$ is accepted by

$$\begin{array}{rcl} S & \rightarrow & \mathbf{a}\;\mathbf{b}\;\mathbf{c} \mid A \\ A & \rightarrow & \mathbf{a}\;A\;B\;\mathbf{c} \mid \mathbf{a}\;\mathbf{b}\;\mathbf{c} \\ \mathbf{c}\;B & \rightarrow & B\;\mathbf{c} \\ \mathbf{b}\;B & \rightarrow & \mathbf{b}\;\mathbf{b} \end{array}$$

▷ **Example 4.2.28 (Context-free).** The language $\{a^{[n]}b^{[n]}\}$ is accepted by $S \rightarrow \mathbf{a}\;S\;\mathbf{b} \mid \epsilon$.

▷ **Example 4.2.29 (Regular).** The language $\{a^{[n]}\}$ is accepted by $S \rightarrow S\;\mathbf{a}$

▷ **Observation:** Natural languages are probably context-sensitive but parsable in real time! (like languages low in the hierarchy)

While the presentation of grammars from above is sufficient in theory, in practice the various grammar rules are difficult and inconvenient to write down. Therefore computer science – where grammars are important to e.g. specify parts of compilers – has developed extensions – notations that can be expressed in terms of the original grammar rules – that make grammars more readable (and writable) for humans. We introduce an important set now.

Useful Extensions of Phrase Structure Grammars

▷ **Definition 4.2.30.** The Bachus Naur form or Backus normal form (BNF) is a metasyntax notation for context-free grammars.

It extends the body of a production rule by mutiple (admissible) constructors:

- ▷ alternative: $s_1 \mid \ldots \mid s_n$,
- ▷ repetition: s^* (arbitrary many s) and s^+ (at least one s),
- ▷ optional: $[s]$ (zero or one times),
- ▷ grouping: $(s_1 ; \ldots ; s_n)$, useful e.g. for repetition,
- ▷ character sets: $[s\text{−}t]$ (all characters c with $s \leq c \leq t$ for a given ordering on the characters), and
- ▷ complements: $[^\wedge s_1,\ldots,s_n]$, provided that the base alphabet is finite.

- **Observation:** All of these can be eliminated, .e.g (⇝ many more rules)
  - replace $X \to Z\ (s^*)\ W$ with the production rules $X \to Z\ Y\ W$, $Y \to \epsilon$, and $Y \to Y\ s$.
  - replace $X \to Z\ (s^+)\ W$ with the production rules $X \to Z\ Y\ W$, $Y \to s$, and $Y \to Y\ s$.

We will now build on the notion of BNF grammar notations and introduce a way of writing down the (short) grammars we need in AI-1 that gives us even more of an overview over what is happening.

An Grammar Notation for AI-1

- **Problem:** In grammars, notations for nonterminal symbols should be
  - short and mnemonic (for the use in the body)
  - close to the official name of the syntactic category (for the use in the head)
- In AI-1 we will only use context-free grammars (simpler, but problem still applies)
- **in AI-1:** I will try to give "grammar overviews" that combine those, e.g. the grammar of first-order logic.

variables	X	$\in$	$\mathcal{V}_1$	
function constants	f^k	$\in$	Σ_k^f	
predicate constants	p^k	$\in$	$\Sigma^p{}_k$	
terms	t	$::=$	X	variable
		$\mid$	f^0	constant
		$\mid$	$f^k(t_1,\ldots,t_k)$	application
formulae	$\mathbf{A}$	$::=$	$p^k(t_1,\ldots,t_k)$	atomic
		$\mid$	$\neg\mathbf{A}$	negation
		$\mid$	$\mathbf{A}_1 \wedge \mathbf{A}_2$	conjunction
		$\mid$	$\forall X.\mathbf{A}$	quantifier

We will generally get by with context-free grammars, which have highly efficient into parsing algorithms, for the formal language we use in this course, but we will not cover the algorithms in AI-1.

4.3 Mathematical Language Recap

We already clarified above that we will use mathematical language as the main vehicle for specifying the concepts underlying the AI algorithms in this course.

In this section, we will recap (or introduce if necessary) an important conceptual practice of modern mathematics: the use of mathematical structures.

Mathematical Structures

- **Observation:** Mathematicians often cast classes of complex objects as mathematical structures.

▷ We have just seen an example of a mathematical structure: (repeated here for convenience)

▷ **Definition 4.3.1.** A phrase structure grammar (also called type 0 grammar, unrestricted grammar, or just grammar) is a tuple $\langle N, \Sigma, P, S\rangle$ where

- ▷ N is a finite set of nonterminal symbols,
- ▷ Σ is a finite set of terminal symbols, members of $\Sigma \cup N$ are called symbols.
- ▷ P is a finite set of production rules: pairs $p := h \rightarrow b$ (also written as $h \Rightarrow b$), where $h \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $b \in (\Sigma \cup N)^*$. The string h is called the head of p and b the body.
- ▷ $S \in N$ is a distinguished symbol called the start symbol (also sentence symbol).

The sets N and Σ are assumed to be disjoint. Any word $w \in \Sigma^*$ is called a terminal word.

▷ **Intuition:** All grammars share structure: they have four components, which again share struccture, which is further described in the definition above.

▷ **Observation:** Even though we call production rules "pairs" above, they are also mathematical structures $\langle h, b\rangle$ with a funny notation $h \rightarrow b$.

Note that the idea of mathematical structures has been picked up by most programming languages in various ways and you should therefore be quite familiar with it once you realize the parallelism.

Mathematical Structures in Programming

▷ **Observation:** Most programming languages have some way of creating "named structures". Referencing components is usually done via "dot notation".

▷ **Example 4.3.2 (Structs in C).** C data structures for representing grammars:

```
struct grule {
  char[][] head;
  char[][] body;
}
struct grammar {
  char[][] nterminals;
  char[][] termininals;
  grule[] grules;
  char[] start;
}
int main() {
  struct grule r1;
  r1.head = "foo";
  r1.body = "bar";
}
```

▷ **Example 4.3.3 (Classes in OOP).** Classes in object-oriented programming languages are based on the same ideas as mathematical structures, only that OOP adds powerful inheritance mechanisms.

Even if the idea of mathematical structures may be familiar from programming, it may be quite intimidating to some students in the mathematical notation we will use in this course. Therefore will – when we get around to it – use a special overview notation in AI-1. We introduce it below.

In AI-1 we use a mixture between Math and Programming Styles

- In AI-1 we use mathematical notation, ...
- **Definition 4.3.4.** A structure signature combines the components, their "types", and accessor names of a mathematical structure in a tabular overview.
- **Example 4.3.5.**

$$\text{grammar} = \left\langle \begin{array}{lll} N & \mathbf{Set} & \text{nonterminal symbols,} \\ \Sigma & \mathbf{Set} & \text{terminal symbols,} \\ P & \{h \rightarrow b \mid \ldots\} & \text{production rules,} \\ S & N & \text{start symbol} \end{array} \right\rangle$$

$$\text{production rule} \quad h \rightarrow b = \left\langle \begin{array}{lll} h & (\Sigma \cup N)^*, N, (\Sigma \cup N)^* & \text{head,} \\ b & (\Sigma \cup N)^* & \text{body} \end{array} \right\rangle$$

  Read the first line "N **Set** nonterminal symbols" in the structure above as "N is in an (unspecified) set and is a nonterminal symbol".

  Here – and in the future – we will use **Set** for the class of sets $\rightsquigarrow$ "N is a set".

- I will try to give structure signatures where necessary.

Chapter 5

Rational Agents: a Unifying Framework for Artificial Intelligence

In this chapter, we introduce a framework that gives a comprehensive conceptual model for the multitude of methods and algorithms we cover in this course. The framework of rational agents accommodates two traditions of AI.
Initially, the focus of AI research was on symbolic methods concentrating on the mental processes of problem solving, starting from Newell/Simon's "physical symbol hypothesis":

> *A physical symbol system has the necessary and sufficient means for general intelligent action.* [NS76]

Here a symbol is a representation an idea, object, or relationship that is physically manifested in (the brain of) an intelligent agent (human or artificial).
Later – in the 1980s – the proponents of embodied AI posited that most features of cognition, whether human or otherwise, are shaped – or at least critically influenced – by aspects of the entire body of the organism. The aspects of the body include the motor system, the perceptual system, bodily interactions with the environment (situatedness) and the assumptions about the world that are built into the structure of the organism. They argue that symbols are not always necessary since

> *The world is its own best model. It is always exactly up to date. It always has every detail there is to be known. The trick is to sense it appropriately and often enough.* [Bro90]

The framework of rational agents initially introduced by Russell and Wefald in [RW91] – accommodates both, it situates agents with percepts and actions in an environment, but does not preclude physical symbol systems – i.e. systems that manipulate symbols as agent functions. Russell and Norvig make it the central metaphor of their book "Artificial Intelligence – A modern approach" [RN03], which we follow in this course.

5.1 Introduction: Rationality in Artificial Intelligence

We now introduce the notion of rational agents as entities in the world that act optimally (given the available information). We situate rational agents in the scientific landscape by looking at variations of the concept that lead to slightly different fields of study.

What is AI? Going into Details

▷ **Recap:** AI studies how we can make the computer do things that humans can still do better at the moment. (humans are proud to be rational)

▷ **What is AI?:** Four possible answers/facets: Systems that

think like humans	think rationally
act like humans	act rationally

expressed by four different definitions/quotes:

	Humanly	**Rational**
Thinking	"*The exciting new effort to make computers think ... machines with human-like minds*" [Hau85]	"*The formalization of mental faculties in terms of computational models*" [CM85]
Acting	"*The art of creating machines that perform actions requiring intelligence when performed by people*" [Kur90]	"*The branch of CS concerned with the automation of appropriate behavior in complex situations*" [LS93]

▷ **Idea:** Rationality is performance-oriented rather than based on imitation.

So, what does modern AI do?

▷ **Acting Humanly:** Turing test, not much pursued outside Loebner prize
- ▷ $\widehat{=}$ building pigeons that can fly so much like real pigeons that they can fool pigeons
- ▷ Not reproducible, not amenable to mathematical analysis

▷ **Thinking Humanly:** $\leadsto$ Cognitive Science.
- ▷ How do humans think? How does the (human) brain work?
- ▷ Neural networks are a (extremely simple so far) approximation

▷ **Thinking Rationally:** Logics, Formalization of knowledge and inference
- ▷ You know the basics, we do some more, fairly widespread in modern AI

▷ **Acting Rationally:** How to make good action choices?
- ▷ Contains logics (one possible way to make intelligent decisions)
- ▷ We are interested in making good choices in practice (e.g. in AlphaGo)

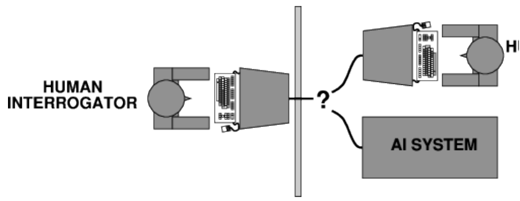
We now discuss all of the four facets in a bit more detail, as they all either contribute directly to our discussion of AI methods or characterize neighboring disciplines.

Acting humanly: The Turing test

▷ Introduced by Alan Turing (1950) "Computing machinery and intelligence" [Tur50]:

▷ "Can machines think?" ⟶ "Can machines behave intelligently?"

▷ **Definition 5.1.1.** The Turing test is an operational test for intelligent behavior based on an imitation game over teletext (arbitrary topic)



▷ It was predicted that by 2000, a machine might have a 30% chance of fooling a lay person for 5 minutes.

▷ **Note:** In [Tur50], Alan Turing

- ▷ anticipated all major arguments against AI in following 50 years and
- ▷ suggested major components of AI: knowledge, reasoning, language understanding, learning

▷ **Problem:** Turing test is not reproducible, constructive, or amenable to mathematical analysis!

Thinking humanly: Cognitive Science

▷ **1960s:** "cognitive revolution": information processing psychology replaced prevailing orthodoxy of behaviorism.

▷ Requires scientific theories of internal activities of the brain

▷ What level of abstraction? "Knowledge" or "circuits"?

▷ **How to validate?:** Requires

1. Predicting and testing behavior of human subjects or (top-down)
2. Direct identification from neurological data. (bottom-up)

▷ **Definition 5.1.2.** Cognitive science is the interdisciplinary, scientific study of the mind and its processes. It examines the nature, the tasks, and the functions of cognition.

▷ **Definition 5.1.3.** Cognitive neuroscience studies the biological processes and aspects that underlie cognition, with a specific focus on the neural connections in the brain which are involved in mental processes.

▷ Both approaches/disciplines are now distinct from AI.

▷ Both share with AI the following characteristic: *the available theories do not explain (or engender) anything resembling human-level general intelligence*

▷ Hence, all three fields share one principal direction!

Thinking rationally: Laws of Thought

▷ Normative (or prescriptive) rather than descriptive

▷ Aristotle: what are correct arguments/thought processes?

▷ Several Greek schools developed various forms of logic: *notation* and *rules of derivation* for thoughts; may or may not have proceeded to the idea of mechanization.

▷ Direct line through mathematics and philosophy to modern AI

▷ **Problems:**

1. Not all intelligent behavior is mediated by logical deliberation
2. What is the purpose of thinking? What thoughts *should* I have out of all the thoughts (logical or otherwise) that I *could* have?

Acting Rationally

▷ **Idea:** Rational behavior $\widehat{=}$ doing the right thing!

▷ **Definition 5.1.4.** Rational behavior consists of always doing what is expected to maximize goal achievement given the available information.

▷ Rational behavior does not necessarily involve thinking e.g., blinking reflex — but thinking should be in the service of rational action.

▷ **Aristotle:** *Every art and every inquiry, and similarly every action and pursuit, is thought to aim at some good.* (Nicomachean Ethics)

The Rational Agents

▷ **Definition 5.1.5.** An agent is an entity that perceives and acts.

▷ **Central Idea:** This course is about designing agent that exhibit rational behavior, i.e. for any given class of environments and tasks, we seek the agent (or class of agents) with the best performance.

▷ **Caveat:** *Computational limitations make perfect rationality unachievable*
$\rightsquigarrow$ design best program for given machine resources.

5.2 Agents and Environments as a Framework for AI

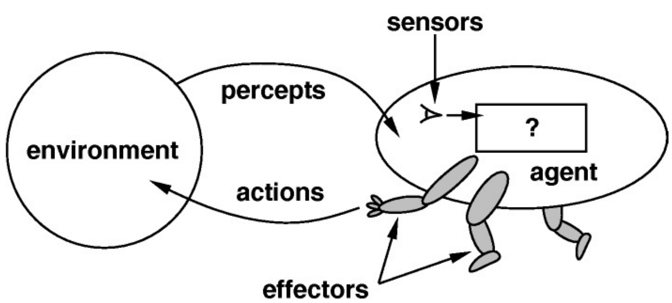
A Video Nugget covering this section can be found at `https://fau.tv/clip/id/21843`.
Given the discussion in the previous section, especially the ideas that "behaving rationally" could be a suitable – since operational – goal for AI research, we build this into the paradigm "rational agents" introduced by Stuart Russell and Eric H. Wefald in [RW91].

Agents and Environments

▷ **Definition 5.2.1.** An agent is anything that

- ▷ perceives its environment via sensors (a means of sensing the environment)
- ▷ acts on it with actuators (means of changing the environment).

Definition 5.2.2. Any recognizable, coherent employment of the actuators of an agent is called an action.



▷ **Example 5.2.3.** Agents include humans, robots, softbots, thermostats, etc.

▷ **remark:** The notion of an agent and its environment is intentionally designed to be inclusive. We will classify and discuss subclasses of both later

One possible objection to this is that the agent and the environment are conceptualized as separate entities; in particular, that the image suggests that the agent itself is not part of the environment. Indeed that is intended, since it makes thinking about agents and environments easier and is of little consequence in practice. In particular, the offending separation is relatively easily fixed if needed.
Let us now try to express the agent/environment ideas introduced above in mathematical language to add the precision we need to start the process towards the implementation of rational agents.

Modeling Agents Mathematically and Computationally

▷ **Definition 5.2.4.** A percept is the perceptual input of an agent at a specific time instant.

▷ **Definition 5.2.5.** Any recognizable, coherent employment of the actuators of an agent is called an action.

▷ **Definition 5.2.6.** The agent function f_a of an agent a maps from percept histories to actions:

$$f_a : \mathcal{P}^* \to \mathcal{A}$$

▷ We assume that agents can always perceive their own actions. (but not necessarily their consequences)

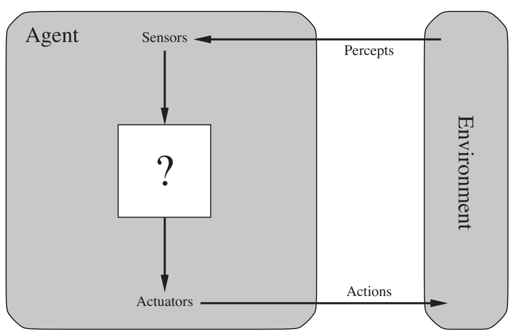
▷ **Problem:** Agent functions can become very big and may be uncomputable. (theoretical tool only)

▷ **Definition 5.2.7.** An agent function can be implemented by an agent program that runs on a (physical or hypothetical) agent architecture.

Here we already see a problem that will recur often in this course: The mathematical formulation gives us an abstract specification of what we want (here the agent function), but not directly a way of how to obtain it. Here, the solution is to choose a computational model for agents (an agent architecture) and see how the agent function can be implemented in a agent program.

Agent Schema: Visualizing the Internal Agent Structure

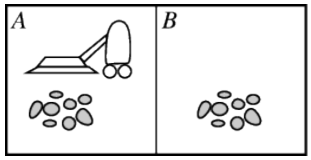
▷ **Agent Schema:** We will use the following kind of agent schema to visualize the internal structure of an agent:



Different agents differ on the contents of the white box in the center.

Let us fortify our intuition about all of this with an example, which we will use often in the course of the AI-1 course.

Example: Vacuum-Cleaner World and Agent



- percepts: location and contents, e.g., $[A, Dirty]$
- actions: $Left$, $Right$, $Suck$, $NoOp$

Percept sequence	Action
$[A, Clean]$	$Right$
$[A, Dirty]$	$Suck$
$[B, Clean]$	$Left$
$[B, Dirty]$	$Suck$
$[A, Clean]$, $[A, Clean]$	$Right$
$[A, Clean]$, $[A, Dirty]$	$Suck$
$[A, Clean]$, $[B, Clean]$	$Left$
$[A, Clean]$, $[B, Dirty]$	$Suck$
$[A, Dirty]$, $[A, Clean]$	$Right$
$[A, Dirty]$, $[A, Dirty]$	$Suck$
$\vdots$	$\vdots$
$[A, Clean]$, $[A, Clean]$, $[A, Clean]$	$Right$
$[A, Clean]$, $[A, Clean]$, $[A, Dirty]$	$Suck$
$\vdots$	$\vdots$

- **Science Question:** What is the *right* agent function?
- **AI Question:** Is there an agent architecture and agent program that implements it.

The first implementation idea inspired by the table in last slide would just be table lookup algorithm.

Table-Driven Agents

- **Idea:** We can just implement the agent function as a lookup table and lookup actions.
- We can directly implement this:

```
function Table−Driven−Agent(percept) returns an action
  persistent table /∗ a table of actions indexed by percept sequences ∗/
  var percepts /∗ a sequence, initially empty ∗/
  append percept to the end of percepts
  action := lookup(percepts, table)
  return action
```

- **Problem:** Why is this not a good idea?
  - The table is much too large: even with n binary percepts whose order of occurrence does not matter, we have 2^n rows in the table.
  - Who is supposed to write this table anyways, even if it "only" has a million entries?

Example: Vacuum-Cleaner Agent Program

- A much better implementation idea is to trigger actions from specific percepts.
- **Example 5.2.8 (Agent Program).**

```
procedure Reflex−Vacuum−Agent [location,status] returns an action
```

if $status$ = Dirty **then return** Suck
else if $location$ = A **then return** Right
else if $location$ = B **then return** Left

▷ This is the kind of agent programs we will be looking for in AI-1.

5.3 Good Behavior $\rightsquigarrow$ Rationality

Now we try understand the mathematics of rational behavior in our quest to make the rational agents paradigm implementable and take steps for realizing AI. **A Video Nugget** covering this section can be found at `https://fau.tv/clip/id/21844`.

Rationality

▷ **Idea:** Try to design agents that are successful! (aka. "do the right thing")

▷ **Problem:** What do we mean by "successful", how do we measure "success"?

▷ **Definition 5.3.1.** A performance measure is a function that evaluates a sequence of environments.

▷ **Example 5.3.2.** A performance measure for a vacuum cleaner could

- ▹ award one point per "square" cleaned up in time T?
- ▹ award one point per clean "square" per time step, minus one per move?
- ▹ penalize for $> k$ dirty squares?

▷ **Definition 5.3.3.** An agent is called rational, if it chooses whichever action maximizes the expected value of the performance measure given the percept sequence to date.

▷ **Critical Observation:** We only need to maximize the expected value, not the actual value of the performance measure!

▷ **Question:** Why is rationality a good quality to aim for?

Let us see how the observation that we only need to maximize the expected value, not the actual value of the performance measure affects the consequences.

Consequences of Rationality: Exploration, Learning, Autonomy

▷ **Note:** A rational agent need not be perfect:

- ▹ It only needs to maximize expected value (rational $\neq$ omniscient)
  - ▹ need not predict e.g. very unlikely but catastrophic events in the future
- ▹ Percepts may not supply all relevant information (rational $\neq$ clairvoyant)
  - ▹ if we cannot perceive things we do not need to react to them.
  - ▹ but we may need to try to find out about hidden dangers (exploration)

- Action outcomes may not be as expected (rational ≠ successful)
  - but we may need to take action to ensure that they do (more often) (learning)
- **Note:** Rationality may entail exploration, learning, autonomy (depending on the environment / task)
- **Definition 5.3.4.** An agent is called autonomous, if it does not rely on the prior knowledge about the environment of the designer.
- Autonomy avoids fixed behaviors that can become unsuccessful in a changing environment. (anything else would be irrational)
- The agent may have to learn all relevant traits, invariants, properties of the environment and actions.

For the design of agent for a specific task – i.e. choose an agent architecture and design an agent program, we have to take into account the performance measure, the environment, and the characteristics of the agent itself; in particular its actions and sensors.

PEAS: Describing the Task Environment

- **Observation:** To design a rational agent, we must specify the task environment in terms of performance measure, environment, actuators, and sensors, together called the PEAS components.
- **Example 5.3.5.** When designing an automated taxi:
  - **Performance measure:** safety, destination, profits, legality, comfort, ...
  - **Environment:** US streets/freeways, traffic, pedestrians, weather, ...
  - **Actuators:** steering, accelerator, brake, horn, speaker/display, ...
  - **Sensors:** video, accelerometers, gauges, engine sensors, keyboard, GPS, ...
- **Example 5.3.6 (Internet Shopping Agent).** The task environment:
  - Performance measure: price, quality, appropriateness, efficiency
  - Environment: current and future WWW sites, vendors, shippers
  - Actuators: display to user, follow URL, fill in form
  - Sensors: HTML pages (text, graphics, scripts)

The PEAS criteria are essentially a laundry list of what an agent design task description should include.

Examples of Agents: PEAS descriptions

Agent Type	Performance measure	Environment	Actuators	Sensors
Chess/Go player	win/loose/draw	game board	moves	board position
Medical diagnosis system	accuracy of diagnosis	patient, staff	display questions, diagnoses	keyboard entry of symptoms
Part-picking robot	percentage of parts in correct bins	conveyor belt with parts, bins	jointed arm and hand	camera, joint angle sensors
Refinery controller	purity, yield, safety	refinery, operators	valves, pumps, heaters, displays	temperature, pressure, chemical sensors
Interactive English tutor	student's score on test	set of students, testing accuracy	display exercises, suggestions, corrections	keyboard entry

Agents

▷ Which are agents?

(A) James Bond.

(B) Your dog.

(C) Vacuum cleaner.

(D) Thermometer.

▷ **Answer:** reserved for the plenary sessions ⇝ be there!

5.4 Classifying Environments

A Video Nugget covering this section can be found at `https://fau.tv/clip/id/21869`.
It is important to understand that the kind of the environment has a very profound effect on the agent design. Depending on the kind, different kinds of agents are needed to be successful. So before we discuss common kind of agents in **??**, we will classify kinds environments.

Environment types

▷ **Observation 5.4.1.** *Agent design is largely determined by the type of environment it is intended for.*

▷ **Problem:** There is a vast number of possible kinds of environments in AI.

▷ **Solution:** Classify along a few "dimensions". (independent characteristics)

▷ **Definition 5.4.2.** For an agent a we classify the environment e of a by its type, which is one of the following. We call e

1. fully observable, iff the a's sensors give it access to the complete state of the environment at any point in time, else partially observable.

2. deterministic, iff the next state of the environment is completely determined by the current state and a's action, else stochastic.
3. episodic, iff a's experience is divided into atomic episodes, where it perceives and then performs a single action. Crucially, the next episode does not depend on previous ones. Non-episodic environments are called sequential.
4. dynamic, iff the environment can change without an action performed by a, else static. If the environment does not change but a's performance measure does, we call e semidynamic.
5. discrete, iff the sets of e's state and a's actions are countable, else continuous.
6. single-agent, iff only a acts on e; else multi-agent (when must we count parts of e as agents?)

Some examples will help us understand the classification of environments better.

Environment Types (Examples)

▷ **Example 5.4.3.** Some environments classified:

	Solitaire	Backgammon	Internet shopping	Taxi
fully observable	No	Yes	No	No
deterministic	Yes	No	Partly	No
episodic	No	Yes	No	No
static	Yes	Semi	Semi	No
discrete	Yes	Yes	Yes	No
single-agent	Yes	No	Yes (except auctions)	No

▷ **Note:** Take the example above with a grain of salt. There are often multiple interpretations that yield different classifications and different agents. (agent designer's choice)

▷ **Example 5.4.4.** Seen as a multi-agent game, chess is deterministic, as a single-agent game, it is stochastic.

▷ **Observation 5.4.5.** *The real world is (of course) a partially observable, stochastic, sequential, dynamic, continuous, and multi-agent environment. (worst case for AI)*

▷ **Preview:** We will concentrate on the "easy" environment types (fully observable, deterministic, episodic, static, and single-agent) in AI-1 and extend them to "realworld"-compatible ones in AI-2.

In the AI-1 course we will work our way from the simpler environment types to the more general ones. Each environment type wil need its own agent types specialized to surviving and doing well in them.

5.5 Types of Agents

We will now discuss the main types of agents we will encounter in this course, get an impression of the variety, and what they can and cannot do. We will start from simple reflex agents, add state, and utility, and finally add learning. **A Video Nugget** covering this section can be found at `https://fau.tv/clip/id/21926`.

Agent Types

▷ **Observation:** So far we have described (and analyzed) agents only by their behavior (cf. agent function $f: \mathcal{P}^* \rightarrow \mathcal{A}$).

▷ **Problem:** This does not help us to build agents. (the goal of AI)

▷ To build an agent, we need to fix an agent architecture and come up with an agent program that runs on it.

▷ **Preview:** Four basic types of agent architectures in order of increasing generality:

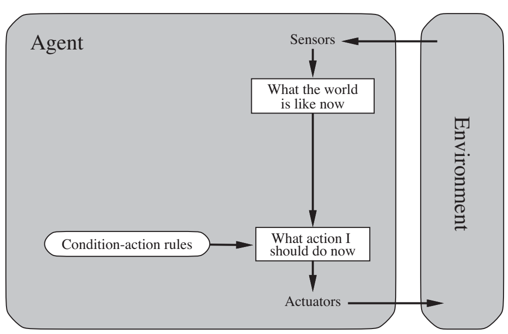
1. simple reflex agents
2. model-based agents
3. goal-based agents
4. utility-based agents

All these can be turned into learning agents.

Simple reflex agents

▷ **Definition 5.5.1.** A simple reflex agent is an agent a that only bases its actions on the last percept: so the agent function simplifies to $f_a: \mathcal{P} \rightarrow \mathcal{A}$.

▷ **Agent Schema:**



▷ **Example 5.5.2 (Agent Program).**

procedure Reflex−Vacuum−Agent [location,status] **returns** an action
  if status = Dirty **then** . . .

Simple reflex agents (continued)

- **General Agent Program:**

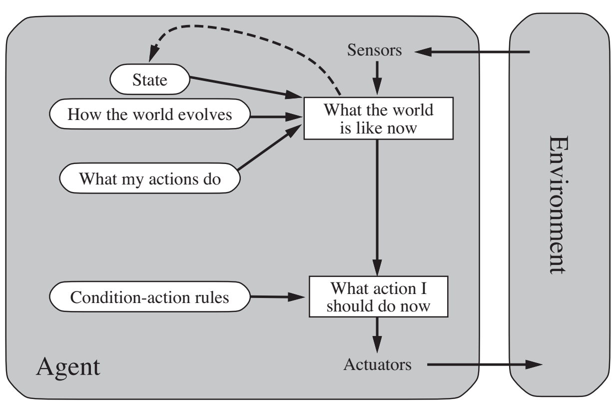
```
function Simple–Reflex–Agent (percept) returns an action
  persistent: rules /* a set of condition–action rules*/

  state := Interpret–Input(percept)
  rule := Rule–Match(state,rules)
  action := Rule–action[rule]
  return action
```

- **Problem:** Simple reflex agents can only react to the perceived state of the environment, not to changes.
- **Example 5.5.3.** Automobile tail lights signal braking by brightening. A simple reflex agent would have to compare subsequent percepts to realize.
- **Problem:** Partially observable environments get simple reflex agents into trouble.
- **Example 5.5.4.** Vacuum cleaner robot with defective location sensor $\leadsto$ infinite loops.

Model-based Reflex Agents: Idea

- **Idea:** Keep track of the state of the world we cannot see in an internal model.
- **Agent Schema:**



Model-based Reflex Agents: Definition

- **Definition 5.5.5.** A model-based agent is an agent whose actions depend on
  - a world model: a set $\mathcal{S}$ of possible states.

- ▷ a sensor model S that given a state s and a percepts p determines a new state $S(s, p)$.
- ▷ a transition model $\mathcal{T}$, that predicts a new state $\mathcal{T}(s, a)$ from a state s and an action a.
- ▷ An action function f that maps (new) states to an actions.

If the world model of a model-based agent A is in state s and A has taken action a, A will transition to state $s' = \mathcal{T}(S(p, s), a)$ and take action $a' = f(s')$.

▷ **Note:** As different percept sequences lead to different states, so the agent function $f_a : \mathcal{P}^* \to \mathcal{A}$ no longer depends only on the last percept.

▷ **Example 5.5.6 (Tail Lights Again).** Model-based agents can do the ?? if the states include a concept of tail light brightness.

Model-Based Agents (continued)

▷ **Observation 5.5.7.** *The agent program for a model-based agent is of the following form:*

```
function Model−Based−Agent (percept) returns an action
  var state /* a description of the current state of the world */
  persistent rules /* a set of condition−action rules */
  var action /* the most recent action, initially none */

  state := Update−State(state,action,percept)
  rule := Rule−Match(state,rules)
  action := Rule−action(rule)
  return action
```

▷ **Problem:** Having a world model does not always determine what to do (rationally).

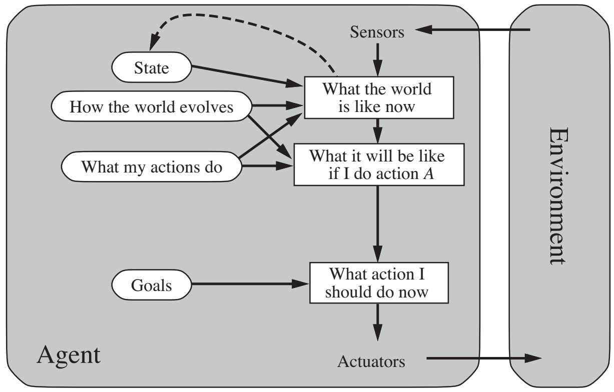
▷ **Example 5.5.8.** Coming to an intersection, where the agent has to decide between going left and right.

Goal-based Agents

▷ **Problem:** A world model does not always determine what to do (rationally).

▷ **Observation:** Having a goal in mind does! (determines future actions)

▷ **Agent Schema:**



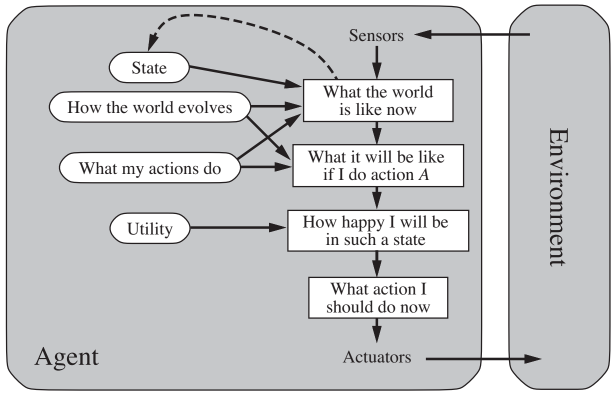
Goal-based agents (continued)

- **Definition 5.5.9.** A goal-based agent is a model-based agent with transition model T that deliberates actions based on 3 and a world model: It employs
  - a set $\mathcal{G}$ of goals and a goal function f that given a (new) state s' selects an action a to best reach $\mathcal{G}$.

  The action function is then $s \mapsto f(T(s), \mathcal{G})$.
- **Observation:** A goal-based agent is more flexible in the knowledge it can utilize.
- **Example 5.5.10.** A goal-based agent can easily be changed to go to a new destination, a model-based agent's rules make it go to exactly one destination.

Utility-based Agents

- **Definition 5.5.11.** A utility-based agent uses a world model along with a utility function that models its preferences among the states of that world. It chooses the action that leads to the best expected utility.
- **Agent Schema:**



Utility-based vs. Goal-based Agents

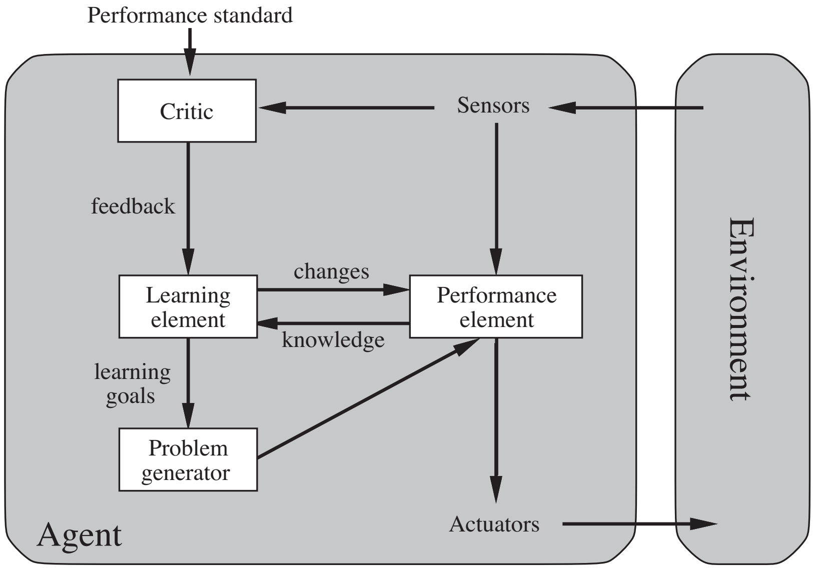
- **Question:** What is the difference between goal-based and utility-based agents?
- **Utility-based Agents are a Generalization:** We can always force goal-directedness by a utility function that only rewards goal states.
- **Goal-based Agents can do less:** A utility function allows rational decisions where mere goals are inadequate:
  - conflicting goals (utility gives tradeoff to make rational decisions)
  - goals obtainable by uncertain actions (utility $\times$ likelihood helps)

Learning Agents

- **Definition 5.5.12.** A learning agent is an agent that augments the performance element – which determines actions from percept sequences with
  - a learning element which makes improvements to the agent's components,
  - a critic which gives feedback to the learning element based on an external performance standard,
  - a problem generator which suggests actions that lead to new and informative experiences.
- The performance element is what we took for the whole agent above.

Learning Agents

- **Agent Schema:**



Learning Agents: Example

- ▷ **Example 5.5.13 (Learning Taxi Agent).** It has the components
  - ▷ Performance element: the knowledge and procedures for selecting driving actions. (this controls the actual driving)
  - ▷ critic: observes the world and informs the learning element (e.g. when passengers complain brutal braking)
  - ▷ Learning element modifies the braking rules in the performance element (e.g. earlier, softer)
  - ▷ Problem generator might experiment with braking on different road surfaces
- ▷ The learning element can make changes to any "knowledge components" of the diagram, e.g. in the
  - ▷ model from the percept sequence (how the world evolves)
  - ▷ success likelihoods by observing action outcomes (what my actions do)
- ▷ **Observation:** here, the passenger complaints serve as part of the "external performance standard" since they correlate to the overall outcome – e.g. in form of tips or blacklists.

Domain-Specific vs. General Agents

Domain-Specific Agent	vs.	General Agent
 Duell Kasparov gegen Deep Blue (1997): *Demütigende Niederlage*	vs.	
Solver specific to a particular problem ("domain").	vs.	Solver based on *description* in a general problem-description language (e.g., the rules of any board game).
More efficient.	vs.	Much less design/maintenance work.

▷ What kind of agent are you?

5.6 Representing the Environment in Agents

We now come to a very important topic, which has a great influence on agent design: how does the agent represent the environment. After all, in all agent designs above (except the simple reflex agent) maintain a notion of world state and how the world state evolves given percepts and actions. The form of this model crucially influences the algorithms we can build. **A Video Nugget** covering this section can be found at `https://fau.tv/clip/id/21925`.

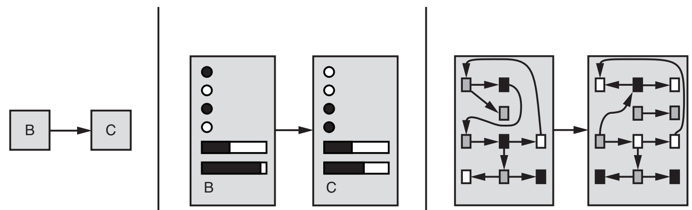
Representing the Environment in Agents

- We have seen various components of agents that answer questions like
  - *What is the world like now?*
  - *What action should I do now?*
  - *What do my actions do?*
- **Next natural question:** How do these work? (see the rest of the course)
- **Important Distinction:** How the agent implements the world model.
- **Definition 5.6.1.** We call a state representation
  - atomic, iff it has no internal structure (black box)
  - factored, iff each state is characterized by attributes and their values.
  - structured, iff the state includes representations of objects, their properties and relationships.
- **Intuition:** From atomic to structured, the representations agent designer more flexibility and the algorithms more components to process.
- **Also** The additional internal structure will make the algorithms more complex.

Again, we fortify our intuitions with a an illustration and an example.

Atomic/Factored/Structured State Representations

▷ **Schematically:** We can visualize the three kinds by



(a) Atomic (b) Factored (b) Structured

▷ **Example 5.6.2.** Consider the problem of finding a driving route from one end of a country to the other via some sequence of cities.

- ▷ In an atomic representation the state is represented by the name of a city.
- ▷ In a factored representation we may have attributes "gps-location", "gas",. . . (allows information sharing between states and uncertainty)
- ▷ But how to represent a situation, where a large truck blocking the road, since it is trying to back into a driveway, but a loose cow is blocking its path. (attribute "TruckAheadBackingIntoDairyFarmDrivewayBlockedByLooseCow" is unlikely)
- ▷ In a structured representation, we can have objects for trucks, cows, etc. and their relationships. (at "run-time")

Note: The set of states in atomic representations and attributes in factored ones is determined at design time, while the objects and their relationships in structured ones are discovered at "runtime".

Here – as always when we evaluate representations – the crucial aspect to look out for are the idendity conditions: when do we consider two representations equal, and when can we (or more crucially algorithms) distinguish them.

For instance for factored representations, make world representations equal, iff the values of the attributes – that are determined at agent design time and thus immutable by the agent – are all equual. So the agent designer has to make sure to add all the attributes to the chosen representation that are necessary to distinguish environments that the agent program needs to treat differently.

It is tempting to think that the situation with atomic representations is easier, since we can "simply" add enough states for the necesssary distictions, but in practice this set of states may have to be infinite, while in factored or structured representations we can keep representations finite.

5.7 Rational Agents: Summary

Summary

▷ Agents interact with environments through actuators and sensors.

- ▷ The agent function describes what the agent does in all circumstances.
- ▷ The performance measure evaluates the environment sequence.
- ▷ A perfectly rational agent maximizes expected performance.
- ▷ Agent programs implement (some) agent functions.
- ▷ PEAS descriptions define task environments.
- ▷ Environments are categorized along several dimensions: fully observable? deterministic? episodic? static? discrete? single-agent?
- ▷ Several basic agent architectures exist: reflex, model-based, goal-based, utility-based

Corollary: We are Agent Designers!

- ▷ **State:** We have seen (and will add more details to) different
  - ▷ agent architectures,
  - ▷ corresponding agent programs and algorithms, and
  - ▷ world representation paradigms.
- ▷ **Problem:** Which one is the best?
- ▷ **Answer:** That really depends on the environment type they have to survive/thrive in! The agent designer – i.e. you – has to choose!
  - ▷ The course gives you the necessary competencies.
  - ▷ There is often more than one reasonable choice.
  - ▷ Often we have to build agents and let them compete to see what really works.

- ▷ **Consequence:** The rational agents paradigm used in this course challenges you to become a good agent designer.

Bibliography

[Bro90] Rodney Brooks. In: *Robotics and Autonomous Systems* 6.1–2 (1990), pp. 3–15. DOI: 10.1016/S0921-8890(05)80025-9.

[Cho65] Noam Chomsky. *Syntactic structures*. Den Haag: Mouton, 1965.

[CM85] Eugene Charniak and Drew McDermott. *Introduction to Artificial Intelligence*. Addison Wesley, 1985.

[Fis] John R. Fisher. *prolog :- tutorial*. URL: https://www.cpp.edu/~jrfisher/www/prolog_tutorial/ (visited on 10/10/2019).

[Fla94] Peter Flach. Wiley, 1994. ISBN: 0471 94152 2. URL: https://github.com/simply-logical/simply-logical/releases/download/v1.0/SL.pdf.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. BN book: Freeman, 1979.

[Hau85] John Haugeland. *Artificial intelligence: the very idea*. Massachusetts Institute of Technology, 1985.

[Kow97] Robert Kowalski. "Algorithm = Logic + Control". In: *Communications of the Association for Computing Machinery* 22 (1997), pp. 424–436.

[Kur90] Ray Kurzweil. *The Age of Intelligent Machines*. MIT Press, 1990. ISBN: 0-262-11121-7.

[LPN] *Learn Prolog Now!* URL: http://lpn.swi-prolog.org/ (visited on 10/10/2019).

[LS93] George F. Luger and William A. Stubblefield. *Artificial Intelligence: Structures and Strategies for Complex Problem Solving*. World Student Series. The Benjamin/Cummings, 1993. ISBN: 9780805347852.

[NS76] Alan Newell and Herbert A. Simon. "Computer Science as Empirical Inquiry: Symbols and Search". In: *Communications of the ACM* 19.3 (1976), pp. 113–126. DOI: 10.1145/360018.360022.

[RN03] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 2nd ed. Pearso n Education, 2003. ISBN: 0137903952.

[RW91] S. J. Russell and E. Wefald. *Do the Right Thing — Studies in limited Rationality*. MIT Press, 1991.

[SWI] *SWI Prolog Reference Manual*. URL: https://www.swi-prolog.org/pldoc/refman/ (visited on 10/10/2019).

[Tur50] Alan Turing. "Computing Machinery and Intelligence". In: *Mind* 59 (1950), pp. 433–460.