



# MAS notes - Summary of the lectures of the multi-agent systems course, given in the second

Multi-agent systems (Vrije Universiteit Amsterdam)

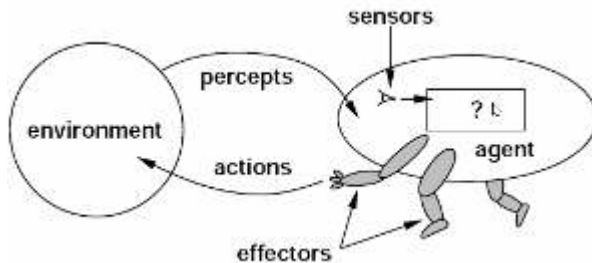
## Week 1, college 1

### This course:

- Knowledge representation and agents & robotics

### Intelligent agent paradigm

- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.



- - Processing is in ?
- Internally, an agent is agent = architecture + program
- Core business of AI: designing architectures and programs.

### Natural agents

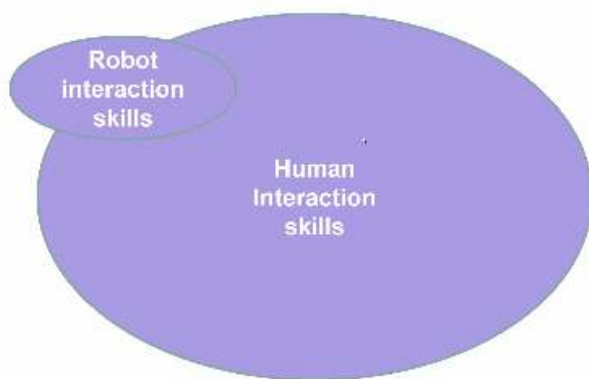
- Animals are agents
  - Can perceive their environment using their eyes, ears, skin, tongue and nose.
  - Can act upon their environments by their muscle motor systems, mounts (sound)
  - Cognitive skills: perception, attention, memory, language, learning and problem solving.
- Human agents
  - Can perceive their environment using their eyes, skin, tongue, nose.
  - Can act upon their environment by their muscle motor systems, mouth (speech)
  - Are very versatile (ability to adapt): can survive in almost any environment on earth.
  - Advanced cognitive skills: perception, attention, memory, language, learning, and problem solving.
- Plants are agents too
  - Can perceive their environment using their sensors.
  - Can act upon their environment by their several muscles and systems.
- Agents are everywhere
  - an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.
  - We are interested here in automated agents.
  - Websites also respond to their own environments.
- Robotic agents
  - Can perceive their environment using their cameras, microphones, touch sensors,...
  - Can act upon their environment by their motor and sound systems, displays.
  - Are currently not very versatile:
    - Can function in specific contexts.

- Do currently not yet have advanced cognitive skills:
  - Limited perception, attention, memory etc.

### Intelligent agents

- Reactive:
  - Ability to receive information and respond.
- Proactive:
  - Ability to take the initiative.
- Social: ability to communicate and cooperate.
- Autonomous:
  - Agents control their own processes.

### About being social



- Human interaction skills go way beyond the robot interactions skills yet.
  - But robots have their own repertoire for interaction, for example displays and lights.
  - Robots do not try to replicate humans necessary.

### Which are intelligent agents

- Plants are not really intelligent in intuition.
- It is not really clear what is intelligent and what not.

### Cognitive(-affective) agents

- An agent is anything that can be (usefully) viewed as a system that has:
  - Beliefs, desires, goals, intentions, plans, expectations, hopes, fears, joy,...
  - ...

### Intentional systems

- First-order
  - $\text{bel}(p)$ : the agent believes that  $p$
  - $\text{goal}(p)$ : the agent has a goal (or wants) that  $p$
- Second-order
  - $\text{bel}(a;\text{bel}(b;p))$ :  $a$  believes that  $p$
  - We now use more first-order

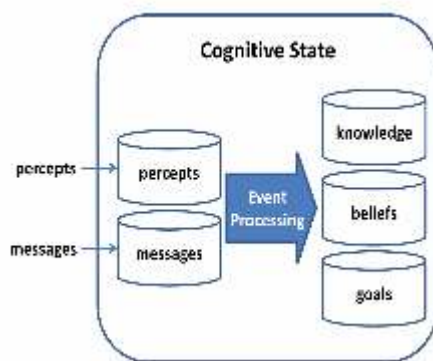
### Our notion of cognitive agent

- Agents with the following basic capabilities:

- Event processing
  - Process events like percepts and messages.
- Knowledge representation
  - Process events like percepts and messages.
  - It allows us to maintain a model of the environment and other agents.
- Decision-making:
  - Agent is able to select an action based on its beliefs, knowledge, and goals.

### Cognitive state

- The internal state of a cognitive agent is called a cognitive state.
- Typically it includes
  - Event component
    - Percepts
    - Messages
  - Informational component:
    - Knowledge (static)
    - Beliefs (dynamic)
  - Motivational component (what the agent wants to achieve):
    - Goals



### Environments

- It is very important to emphasize that agents are situated in an environment.
- Percepts:
  - Agents to usually not see the real state of the environment but only receive percepts.
  - Designer:
    - The designer has to process percepts and possibly store them within the agent's memory.
  - Environment-properties:
    - As such it is very important to know the characteristics of an environment, before designing an agent.

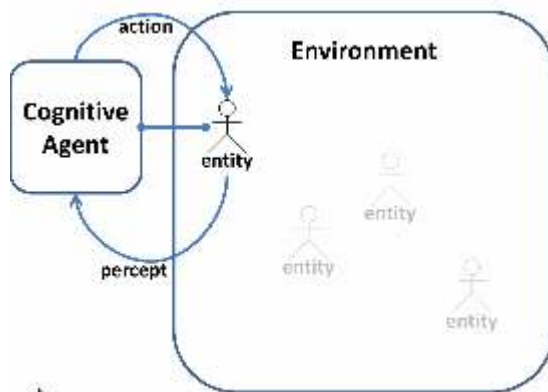
### Properties of environments

- Fully/partially observable: if the environment is not completely observable the agent will need internal states.
- Deterministic/stochastic:
  - Deterministic if completely determined by agent's action.

- If the environment is only partially observable, then it may appear stochastic (while it is deterministic).
- Static/dynamic:
  - The environment can change while an agent is deliberating.
- Discrete/continuous: if there is a limited number of percepts and actions the environment is discrete.
- Single/multi agents: is there just one agent or are there several interacting with each other.

### Cognitive agents and environments

- For cognitive agents we take the view that an environment offers controllable entities which are connected to cognitive agents.
- Possible view:
  - Cognitive agent is the mind
- controllable entity is the body.



- An action specification defines which actions are available to an agent and when.
  - It is defined by the environment.

### Interaction with environment and agents

- Observation: how can the agent observe its environment?
  - Passive
    - The agent receives the results of observations without taking any initiative or control to observe.
  - Active
    - The agent actively ini....

### Summary

- The (material) world state
- Performing observations in the world
- Execution of action in the world
- Performing communication with other agents
- An agent's own knowledge
- An agent's own assumptions (for example beliefs)
- An agent's own reasoning and acting processes.
- Other agent's...

### Cognitive agent programming

- A cognitive agent language is a programming language with events, beliefs, goals, actions, plans, etc as first class citizens.

### **Key language elements**

- Events received from environment (& agents)....

### **Prolog and GOAL**

- Knowledge representation language for inspecting and inferring from an agent's cognitive state.
  - Knowledge
  - Beliefs
  - Goals

### **Prolog compared to other programming languages**

- Procedural
- Object-oriented
- Functional
- Logic-based
- A logic-based programming language, focus on the description of a problem (and thus not on how to solve it)
  - Declarative programming vs. imperative/procedural programming.
- Prolog is based on a different programming paradigm, and thus requires a different way of thinking.

### **Imperative vs. declarative**

- The declarative approach describes the problem itself and the programming problem finds the solution.
- The imperative approach describes the whole process.

### **Why use prolog**

- Easy to represent knowledge through code.
- Uses problem solving based on reasoning.
- Meta-programming:
  - Given a description of a problem, decides steps itself.
- Ideal for keeping track of states.

### **Propositional logics**

- A declarative sentence (or proposition) is a statement that is true or false.

### **Argument abstraction**

- For example:
  - If john wears his lucky underwear and it is not a wednesday, then his favourite team will win.
  - John's favourite team did not win.
  - John wears his luck underwear.
  - Therefore, it is a wednesday (inferred)
- In propositional logic we can abstract these statements as p, q and r.
  - The this is:

- if p and not q, then r
  - not r
  - p
  - therefore q.
- This formalization can be applied to all situations.
- Propositional logic gives us the syntax and semantics to define situations and statements.
- Validity of these arguments is due to their logical form.
  - ...

### Symbols of propositional logic

- We want to study logic without being distracted by the concrete contents of propositions.

#### Propositional variables

p, q, r, ...

#### Connectives

$\wedge$	'and'
$\vee$	'or'
$\oplus$	'either ... or ...' (XOR)
$\neg$	'not'
$\rightarrow$	'if ... then ...'
$\leftrightarrow$	'if and only if'

- 
- Not in the scope of propositional logic are constructs like:
  - For all, there exists
  - Must, may, always, eventually, I know that.

### Standard syntax

- Declarative sentences (propositions)
- Propositional formulas
  - Every propositional variable p, q, r is a formula
  - If p is a formula then so is  $\neg p$

### Truth value semantics

- Formulas of propositional logic are used to express declarative statements which are either true or false.
- We introduce the t...

### Remember truth tables for:

- Negation
- Conjunction
- Disjunction
- Exclusive disjunction
- Implication
- And remember that a valuation corresponds to one line in the truth table of a formula.
  - There are  $2^n$  lines where n is the number of variables

## First-order logic

- Logical structures are indicated as  $L = (U, S, I)$  where  $L$  is the name of the structure,  $U$  is a universe,  $S$  is its signature and  $I$  is an interpretation.
- Universe: the universe of objects about which a first-order logic theory is formulated, also called the domain of discourse.
- Signature: The set of non-logical symbols with which...

## Syntax first-order logic

- To create expressions in first order logic you can use:
  - Predicate and function symbols of the signature...

## Example:

### Signature S

#### • Function symbols:

- $o^0$
- $c^0$
- $s^0$

#### • Predicate symbols:

- $pink^1: U$
- $grey^1: U$
- $likes^2: U \times U$

### Universe U & mapping I



- That likes relation can go between 2 elephants.

### Informal

- Sam is pink
- Clyde is grey
- Clyde likes Oscar
- Oscar is either pink or grey, not both
- Oscar likes Sam

### Formal

- $pink(s)$
- $grey(c)$
- $likes(c, o)$
- $pink(o) \vee grey(o) \neg (pink(o) \wedge grey(o))$
- $likes(o, s)$

- There is a grey elephant that likes a pink one

To prove:  
 $\exists x \exists y (grey(x) \wedge pink(y) \wedge likes(x, y))$

## Example: informal $\rightarrow$ formal

- Define the universe
- Define the signature
  - The way in which you describe the universe.
- Define the interpretation
- Construct the formula

Signature and Interpretation:

Predicate symbols:

- $s^1$  student
- $t^1$  teacher
- $y^2$  younger than

Formula:

$\forall x (S(x) \rightarrow (\exists y (T(y) \wedge Y(x, y))))$



### **Universe, interpretation**

- Not for all exercise you need to specify the universe and/or the interpretation.
- For extract examples you can assume them to be the universe of everything and the interpretation....

### **First order logic**

- logical structures (universe, signature, interpretation)
- Predicate and function symbols
- Variables and constant
- Quantifiers
- Boolean operators
- Formulas and terms

### **Prolog and logic**

- Declarative programming paradigm
- Building block for propositional logic and first order logic:
  - Propositional values, logical connectives and propositional formulae.
  - Logical structure, predicates, functions and quantifiers.
  - Valuation and semantic relation
- Prolog applies theorem proving given a goal (formula) and a logical structure.
  - Central construct: unification.

### **Week 1, college 2**

- We use an example of redlipped batfishes.
- The aim of prolog is to
  - Represent knowledge efficiently through facts and rules.
  - Retrieve and infer knowledge from this representation, given a query.
- Logical principles are needed to enable such inference.

### **Prolog: general framework**

1. Knowledge base
  - Facts
  - Rules
  - Stored in .pl files
- Queries
  - Posted from shell/script.
  - ?-

### **Knowledge base:**

- redlipped\_batfish(bertha).
- redlipped\_batfish(robert).
  - The things redlipped\_batfish is what we call a predicate.
  - This one has 1 value.
  - We say there is a redlipped\_batfish that is called bertha or robert.
- bertha.
  - This has no clear function and we will not use.

- Some queries:

```
?- redlipped_batfish(robert).
true.
?- redlipped_batfish(bertha).
true.
?- redlipped_batfish(robert), redlipped_batfish(bertha).
true.
?- redlipped_batfish(harco).
false.
?- robert.
ERROR: Unknown procedure: robert/0
?- bertha.
true.
```

- Now we add a shark harco with:

- shark(harco).

```
redlipped_batfish(bertha).
redlipped_batfish(robert).
shark(harco).
```

- We also now want to say that these things are actually fish.

- We want to query `?-fish(bertha).` true
- These things are all fish.
- The first solution is to add `fish(bertha)` and for all others.
  - Is very labour expensive.
- We can also give a rule:

- `fish(X) :- redlipped_batfish(X); shark(X).`

- This tells us:  $\forall x(\text{Redlipped\_Batfish}(x) \vee \text{Shark}(x)) \rightarrow \text{Fish}(x)$

- Says for all x, if x is a redlipped\_batfish or a shark, then it is a fish.

```
redlipped_batfish(bertha).
redlipped_batfish(robert).
shark(harco).
fish(X) :- redlipped_batfish(X); shark(X).
```

- So:

- Here we have 3 clauses.
  - A single line is a clause.
- 3 facts
  - Statements about the world.
- One rule
  - Inferences can be made over those.
- harco = atom
- X = variable
  - These are capital letters.
  - Can take different forms.
  - Are used in rules as quantification.
  - In queries when we include a variable, it gives us any case for which an atom is that thing.

```

?- redlipped_batfish(X).
X = bertha ;
X = robert.

```

- Prolog will infer bertha, harco and robert as fish.
- Everything ends with a dot.
  - In queries and statements
- A rule sign is :-
  - Left side is the head, right side is the body.
  - ; means or in the rule.
  - , means conjunction.
- How can we say that the batfish dwells in the galapagos islands?
  - We can say:
    - dwells\_in(X,galapagos\_islands) :- redlipped\_batfish(X).
  - Is a very generic way.
    - You want your rules to be reusable.
- You do not want to represent knowledge in a negative way.
  - So like saying not(goodswimmer(X)) :- redlipped\_batfish.
  - We can say for example goodswimmer(X) :- shark(X)
    - Implies that everything that is not in there is not a good swimmer.
- KB now looks like this with possible queries:

```

redlipped_batfish(bertha).
redlipped_batfish(robert).
shark(harco).

fish(X) :- redlipped_batfish(X); shark(X).
goodswimmer(X) :- shark(X).
dwells_in(X,galapagos_islands) :-
    redlipped_batfish(X).

```

```

?- dwells_in(robert, Y).
Y = galapagos_islands.
?- dwells_in(A, galapagos_islands).
A = bertha ;
A = robert.
?- dwells_in(harco, galapagos_islands).
false.
?- goodswimmer(robert).
false.
?- dwells_in(A, B).
A = bertha,
B = galapagos_islands ;
A = robert,
B = galapagos_islands.

```

- With queries asking for 2 queries, we can either give a variable and returns the things in the variable, so that's what you ask for.
- Now if we add waved\_albatross(ferdinand).
  - With dwells\_in(X,galapagos\_islands) :- waved\_albatross(X).
  - Then some more possible queries:

```

?- dwells_in(A,galapagos_islands), fish(A).
A = bertha ;
A = robert.
?- dwells_in(A,galapagos_islands), not(fish(A)).
A = ferdinand.
?- not(fish(A)).
false.

```

- 
- First one says give us everything that dwells in the galapagos islands and is a fish.
- Second one says the same but everything that is not a fish.
- We cannot ask prolog for everything that is not a certain thing.

## Prolog syntax

- :- defines an implication ( $a \rightarrow B$  corresponds to  $B :- A$ ).
- , defines a conjunction (in right-hand side of rule)
- ; defines a disjunction (but can also be multiple rules)
- An atom is one of the following:
  - A sequence of letters, digits or underscores that starts with a lower-case letter.
    - Like father, or hOUSE\_2
  - A sequence of letters, digits or underscores that are enclosed in single quotes.
    - Like "HOUSE 2", 'car' or 'House'
  - A string of special characters.
    - Like :- or ,
  - A number is any sequence of numbers, possibly containing a dot.
  - Both atoms and numbers are called constants.
- A variable is a sequence of letters, digits or underscores that starts with an upper-case letter or an underscore.
  - For example X, Car, HOUSE\_2, \_father, House.
  - The variable \_ is called the anonymous variable.
- A term is defined recursively:
  - Each constant and variable is a term.
  - $f(t_1, \dots, t_k)$  is a term if each  $t_i$  is a term and  $f$  is an atom.
    - Number of values  $t_k$  is the amount of things a term takes.
    - This is called a compound term with functor  $f$  and arity  $k$ , which is often referred to in the notation  $f/k$  (signature)
    - For example woman/1 or :-/2.
    - A term is ground when it contains no variables.
- Any constant or compound term is known as a predicate.
- If immediately followed by a dot (.), it is called a fact (knowledge)
- Otherwise, it can be a rule if:
  - Given by  $A_0 :- A_1, \dots, A_n$ . with  $n \geq 0$  and each  $A_i$  a predicate.
  - $A_0$  is called the head of the rule,  $A_1, \dots, A_n$  the body.
- A clause is any fact or rule.
  - $p(X) :- f(X), r(X)$  (prolog)
  - $f(X) \wedge r(X) \rightarrow p(X)$  (FOL)
  - clauses are always universal.

- Here  $P(X)$  holds for all  $X$ .

22	constant, term, predicate
$p(X) :- f(X), z(Y).$	rule
$f(x(A), z(B)).$	fact, compound term, predicate
Robert	variable, term, predicate

- 
- A prolog program is a finite sequence of clauses.
- The set of all facts and rules that use a certain predicate  $P$  is called the definition of that predicate.
  - If there is none in a certain program,  $P$  is called undefined (cannot be queried).
- A built-in predicate is defined in SWI Prolog itself.
  - For example `fail/0`, `true/0`, `consult/1`
- A query is defined as `?- p.` where  $p$  is some predicate, known then as a goal.
  - Queries are existential (where the knowledgebase is universal).
    - $?-p(X)$  corresponds to for some  $X$   $p(X)$  in FOL

### Unification in prolog

- Is all about such a query: `?-ab = ab`
  - It will say true.
- Unification is key to prolog and interchange between variables.
- We need to define when 2 predicates match.
- Matching is Prolog's method for unification.
  - We generally still say unification when referring to Prolog's matching.
- 2 terms  $t_1$  and  $t_2$  match iff:
  - $t_1$  and  $t_2$  are identical constants (starting with lowercase letter).
  - If  $t_1$  is a variable and  $t_2$  is not, then  $t_1$  is instantiated with  $t_2$ .
    - The variable will take the same form as what it means.
    - The same applies for the reverse or when both are variables.
  - If  $t_1$  and  $t_2$  are compound terms (like `dwells` in we saw earlier), they match if:
    - They have the same outermost functor.
      - For example  $t_1 = f(r_1, \dots, r_k)$  and  $t_2 = (s_1, \dots, s_k)$
    - Each term  $r_i$  matches with each term  $s_1$ .
    - The variable instantiations are compatible.
      - A variable cannot be assigned different terms which are not variables.
- `=/2` is the built-in matching predicate (so not equality operator).
  - Binds variables using the most general unifier.
- `\=/2` can be used to check for terms that do not match.

### Examples:

- `?- ab = ab` aligns with the first rule, and will say true.
- `?- X = a` aligns with the second rule
  - We see some instantiation going on.
  - Then prolog will say  $X = a$
- With `?- X = f(a)`, the output will be  $X = f(a)$ .

- X has taken a value.
  - Is again the second rule.
- $?- f(X, g(a)) = f(b, g(Z))$ .
  - Will return  $X = b$  and  $Z = a$ .
- $?- f(X, a) = f(b, X)$ .
  - Returns FALSE.
- $?- X + 3 = 5 + Y$ 
  - Will return  $X = 5$  and  $Y = 3$ .
  - Again some instantiation is going on.
  - No mathematics going on, just unification.
- But  $?- X + 2 = 5$  will return false!
  - It does not see a variable to the right.
- $?- \text{term}(X) = \text{term}(Y)$ . will return  $X = Y$ .
- $?- \text{term}(X, X) = \text{term}(Y, Z)$ . returns  $Y = Z$ .

### Deduction

- Suppose we are given the following program and ask the following query:

```
q(a).
r(a).
p(X) :- q(X), r(X).
```

```
?- p(a).
```

- 
- First of all  $p(a)$  is matched with  $p(X)$ .
- Then start with deriving the body of the rule:  $q(a), r(a)$ .
- Both goals are facts in the knowledge base, so true will be returned.

### Searching for proofs

- Searching for a proof of a query, or an answer to a query is a multiple step process:
  - Ask a query
  - Find appropriate rule
  - Prove body of rule
  - Continue until no alternative remain.

### Three principles

- Backward chaining:
  - Start from the current goal and select the rule/fact which allows to derive the goal.
  - Then try to derive the rule's body (from left to right).
- Linear:
  - Traverse the logic program from top to bottom (select appropriate rules).
- Backtracking:
  - If the current goal cannot further be matched with any rule, try other alternative rules in step 2.
    - For example the 2nd topmost, 3rd topmost etc.
  - Backtrack to the closest possible choice point and try an alternative.
  - May happen when you have multiple rules for the same name that can be unified for the current query.
- Example:

```

sport(jack).
relaxed(alice).
healthy(X) :- sport(X).
happy(X) :- healthy(X).
happy(X) :- relaxed(X).

```

```
?- happy(alice).
```

- 
- By the linear principle we start with the first rule (4th line).
  - Then it will try to prove the rule of the body, which is healthy(X) (3rd line)
  - Then it searches sport(alice).
  - It leads to a failure.
    - Then prolog will go back and look if there are any alternatives.
    - Then it will see the 5th line which leads to success.
- Example for backward chaining compound queries:
 

```
goal1(t3) :- p1(t4), p2(t5).
goal1(t1) = goal1(t3).
```

```
?- goal1(t1), goal2(t2).
```

  - 
  - Find the first topmost rule such that head of rule matches with goal1(t1) for an instantiation.
  - Continue with new query (body replacement)
    - **?- p1(t4) $\sigma$ , p2(t5) $\sigma$ , goal2(t2).**
    - Note: (sub)goals are treated from left to right.
- When can we stop?
  - query?-
    - No more goals to deduce (empty query).
    - Success.
  - Query ?- p(t),...and:
    - No rule's head matches with p(t)
    - No fact matches with p(t)
      - What now? failure, backtrack to most recent choice point and try alternative rule.
      - No alternatives left: terminate and fail.
    - A failure can coexist with some successful output.
      - As something can lead to several outputs.

### Standardisation

- If a new rule like head(X,Y) :- left(X), right(Y). is selected, prolog:
  - Introduces new variable names and replaces old ones uniformly.
    - Unification is done over and over.
  - E.g., an instantiated rule looks as follows:
 

```
head(_G4, _G453) :- left(_G4), right(_G453).
```
  - - In the search a variable is instantiated with an abstract value.
- With the trace function, you can see what prolog actually takes as steps.

### Search tree example

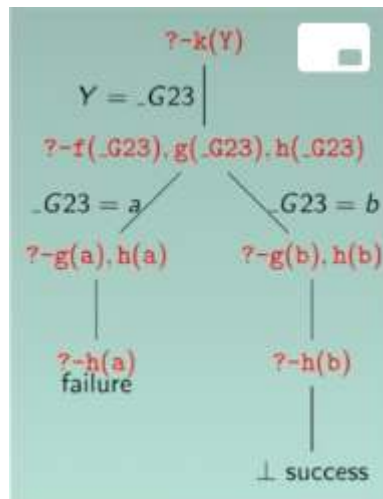
```

f(a).
f(b).
g(a).
g(b).
h(b).
k(X) :- f(X), g(X), h(X).

```

?- k(Y)

- 
- Use trace and read the output.
- How this is drawn in search tree:



### Prolog search tree

- Given a Prolog P and a query Q the just described procedure results in a tree with the following properties:
  - The root is labelled with query Q.
  - Nodes are labelled with queries that prolog has to derive.
  - Child nodes are ordered (according to the position of the rule in the program).
  - Edges are labelled with variable instantiations
- We call this tree the search tree of P and Q (sometimes also called proof tree).
- See example in lecture.
  - Remember here that you traverse all possible assignments of variables.

### Search tree branches

- A search tree can have 3 different types of branches:
  - Infinite ones.

- You would get from the following query:

Consider the following program and query:

```

s(a).
r(b).
p(X) :- r(X).
p(X) :- s(X).
p(X) :- t(X).
t(X) :- t(X).

```

?- p(a).

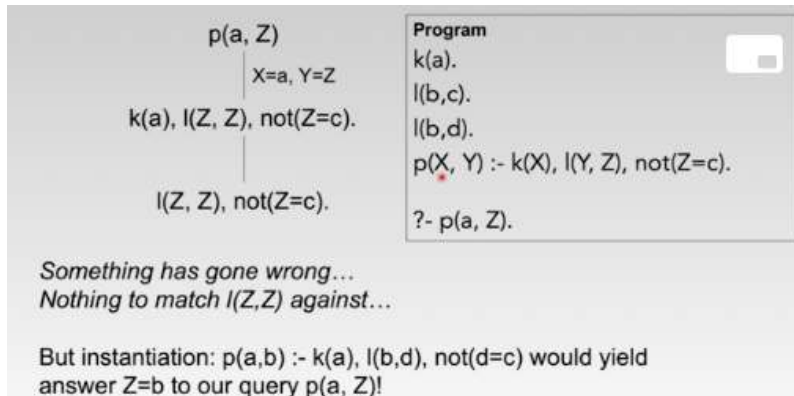
■ The query yields a failing, successful, and an infinite branch (in that order).

- In the last rule, it searches for t(X), ends up in the same rule and searches for t(X) again.

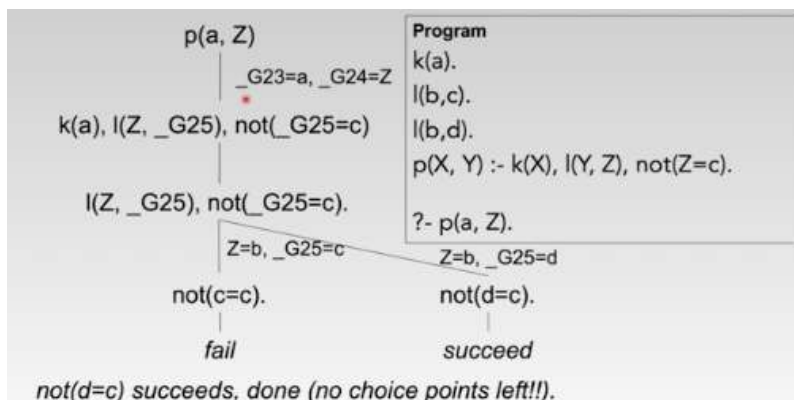


- Never make these rules.
  - Successful branches ending with the empty clause (and leading to an answer), and
  - Failing branches (dead ends).
  - Endpoint is always infinite, failure or success.

### Search tree - variables example



- Match with rule, but make sure you take care to rename variables used in rule to avoid variable clashes.



## Week 2, college 1

### Recursion

- Recursion in procedural/object-oriented programming:
  - A method of solving a problem where the solution depends on solutions to smaller instances of the same problem.
  - A recursive definition is defined in terms of itself.
  - Important to include termination on condition.
- Recursion in prolog
  - Compact form to define predicates.
  - Let the computer do the work.
  - Danger of non-termination.

### Examples of recursion in prolog

- Identify implicit relations in knowledge base.
  - Inheritance
  - Public transport connection

- Syntactic relations
- Building/deducing structure through unification.
  - Successions
  - Equations
  - List processing.
- Base clause
  - Descendent(X,Y) :- child(X,Y).
- Recursive clause
  - Descendent(X,Y) :- child(X,Z), descendent(Z,Y).
  - Is recursive because it calls upon itself.
- So instead of this

```
child(sara, frank).
child(paul, ann).
child(ann, sara).
descendent(X,Y) :- child(X,Y).
descendent(X,Y) :- child(X,Z), child(Z,Y).
descendent(X,Y) :- child(X, W), child(W, Z), child(Z, Y).
```

- 
- We can have this:
- **descendent(X, Y) :- child(X, Z), descendent(Z, Y).**

### Reachable

- There is recursion if there is reachability of predicates.
- A predicate pred2 is reachable from predicate pred1 in a program if:
  - There is a rule in P the head of which uses pred1 and the body of which contains pred2; or
  - There is a predicate pred3 such that pred3 is reachable from pred1 in P, and pred2 is reachable from pred3 in P.
    - Is pretty indirect.
- A predicate pred is defined recursively if there is at least one rule in the definition of pred in P from which pred itself is reachable.
  - For example:
    - The predicate [1 with p(X) :- P(X), q(X). is defined recursively.
  - The following example is a recursive indirect definition:

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

- Then we say it is a recursive predicate.

- Stop clause:

```
on(b,a).
on(c,b).
above(X,Y) :- on(X,Y).
above(X,Y) :- on(X,Z), above(Z,Y).
```

- A prolog program always checks first if the base clause can be satisfied.

- The second one on top is the stop clause.
- At the point where prolog comes upon a fact where the given X and Y variables are included, then the search tree stops.
  - From there it can track back again.
  - The recursion goes on until it finds that final statement.

### Inductive definitions

- Consider inductive definition of the natural numbers:
  - 0 is a natural number.
  - If n is a natural number, then n+1 is a natural number.
  - There are no more numbers than those obtained by 1 and 2.
- The definition consists of 3 parts:
  - The base case
  - The induction step, and
  - A maximality condition
- For prolog it often works like that too:
  - Basic clause:
    - Think about boundary case if A is a child of B, then A is a descendent of B.
  - Inductive clause:
    - Think about the general case children of descendants are also descendants.
  - We don't have a concrete extremal case.
    - Think about negation as failure and how prolog derives new facts:
      - If something cannot be derived given the current program then it is considered not to hold.

### Example's and do's and don'ts

- Example:



- What it will do here (the rule's below are switched out of usual order) is depth first search.
  - The final fact that closes everything is searched.
  - Here it will ends up with anna = A and emily = B.
- Never have a recursive clause that call upon itself directly when no variable has been instantiated yet.
  - Like in here:
 

```
desc(X,Y) :- desc(Z,Y), child(X,Z).
desc(X,Y) :- child(X,Y).
```

### Recursion: rules of thumb

- Recursion is powerful and dangerous.
- Try to avoid left-recursion.
  - It is usually better to make the recursive call as late as possible.
- Recursive call as far as possible to the right of a rule.
- Rule and predicate ordering is important.
- Try to put base clause before recursive clause.

### Non-termination

- Recursion is powerful but can yield non-terminating behavior.
- Like this example:



- The query yields a successful and an infinite branch.
- Prolog searches the program in a depth first search fashion with backtracking.
  - Starting at root node and exploring as far as possible along each branch before backtracking.
- Prolog is both incomplete and incorrect because:
  - Prolog uses depth-first-search:
    - It can get lost in an infinite branch.
  - By default prolog uses incorrect unification.
    - No occurs check,  $X = f(X)$
    - For efficiency reasons.
    - It is not displaying the logic behaviour that you expect.

### Derivation; declarative and procedural meaning

- A goal G can be derived or proven from a program P if prolog finds a successful trace in the search tree of P for G.
- Declarative meaning:
  - Logical meaning of a program, forget about prolog's search strategy.
- Procedural meaning:
  - Answer computed by Prolog.
  - The way a goal is derived by prolog.
- The declarative and procedural meaning of a program can be different.
  - One of them can make something unusable while the other meaning is okay.
- Incompleteness:
  - Ordering effects ordering of solutions.

### Concluding remarks


- Classic approach to recursion:
  - Base clause
  - Recursive clause
- Success and efficiency of prolog proof search depends on the ordering of a program:
  - Recursion comes at the risk of infinite loops.

- Often, a good way to design a prolog program is to follow the following steps:
  - Understand the problem.
  - Think about a declarative solution, and come up with a logic program.
  - Refine the program and take into consideration Prolog's procedural meaning.

## Negation

- Prolog allows a predicate `\+/1` which models a specific type of negation.
  - In short it means:
    - `\+ f` is true in program `P` iff `f` cannot be derived from `P` and the program terminates.
  - Compare the difference:
    - `-p` in logic: proposition `p` is false.
    - `\p` in prolog: there is not enough information to derive `p`.
  - You can also write `not(...)` for `\+`.
- Definition:
  - A search tree for a program `P` for a goal `G` is called finitely failed if there is:
    - No successful branch in the tree.
    - Also no infinite branch.
- Negation as finite failure.
  - The operator `\+` is called negation with finite failure (`naff`).
  - `\+ G` can be derived from `P` iff the search tree of `P` for `G` is finitely failed.
  - Example:

```
daughterOf(alice,bob).
daughterOf(jessica,mary).
sonOf(paul,mary).
onlyDaughter1(X):- \+ sonOf( _, Z), daughterOf(X,Z).
onlyDaughter2(X):- daughterOf(X,Z), \+ sonOf( _, Z).
```



```
?- onlyDaughter1(A).
?- onlyDaughter2(A).
```

- Only the last rule will lead to a correct answer and will lead to negation with finite failure in the correct way.
  - It fails at the search.
- What is asked in the first rule is first if there is no `sonOf` relation in the KB.
  - Starting with such a negation with no variables instantiated is very risky.

## Built-in arithmetic functors

- Examples of important ones:
  - `+/2` addition
  - `-/2` subtraction
  - `-/1` negative number
  - `*/1` negative number
  - `//2` (floating point) division
  - `///2` (integer) division
  - `mod/2` remainder after division
  - `max/2` maximum of 2 numbers.
- These things are actually predicates in prolog.
- Terms are not evaluated.

`?- 4 + 3 = 7.`

`false.`

- We have operators that force evaluation.
  - `:=` (left and right evaluation of terms)

`?- 4 + 3 := 5 + 2.`

`true.`

- It will first calculate the 2 terms, and then compare.

- `=\=` (inequality)

`?- 4 + 3 =\= 5 + 2.`

`false.`

- Comparison operators
  - Force left and right evaluation.
    - `</2` (strictly larger)
      - `4 + 2 < 5 + 3.`
        - Is true
  - Other comparison operators:
    - `=<`
    - `>`
    - `>=`

### Evaluating terms with variables

`?- 4 + 3 := X + 2.`

`ERROR: :=/2: Arguments are not sufficiently instantiated`

- Same with inequality `=\=` because X is not instantiated yet.
- All these equations types will not work because terms are not instantiated yet.
- Prolog does all these things with the “is” operator.

- `t1 is t2`
  - `t1` equals the evaluation of `t2`
  - It asks if `t1` is equal to `t2` to the right.

`?- 4 is 2 + 2.`

`true.`

- What is done to the right is done first.
  - It will conclude a number and then checks if right and left are the same.

- But:

`?- 2 + 2 is 2 + 2.`

`false.`

- The `is` operator will do operation on the right side.
- In this specific case `2 + 2` is not the same as `4`.

- Also:

`?- X is 2 + 2.`

`X=4.`

- Will lead to an instantiation of X!

- But still this does not work:

?- 4 is X + 2.

\* ERROR: is/2: Arguments are not sufficiently instantiated

- 
- When X does not have a value yet, you can't do math.
- What will work:

?- X = 2, 4 is X + 2.  
true.

- You start instantiating X.
- Right-hand side of is/2 must be fully instantiated when evaluated.
- ?- 2 + 2 = 2 + 2. will return true.
  - It does not do any calculations, but 2 + 2 is exactly the same sequence as 2 + 2.
- ?- X = 2 + 2 will result in X being 2 + 2, so not 4.
- X := 2 + 2 will not work, no instantiation yet performed on X.
- 1 is 7 mod 2.
  - Is modulo
- Power of:
  - 1024 is 2\*\*10
- Compute maximum of 4 and 89:
  - X is max(4,89).
- Example to double some X and add 3:
  - addThreeAndDouble(X,Y):- Y is 2\*(X+3)

## Week 2, college 2

### Lists

- Lists are important data structure in Prolog.
- They can store (compound) terms.
- Examples of lists:
  - [michelangelo, donatello, raphael, leonardo]
  - [michelangelo, ninja(donatello), X, 2, michelangelo]
  - []
  - [michelangelo, [donatello, raphael], [splinter, sensei(splinter)]]
  - [[ ], mutant(z), [2, [b,c]], [ ], Z, [2, [b,c]]]
  - As you see lists within lists can also be.

### []/2 operator

- Internally, lists are defined recursively using a functor []/2 which allows to concatenate terms:
- How to use:
  - [] : the empty list
  - [](a, []) : corresponds to [a].
  - [](a, [](b, [])) : corresponds to [a,b].
  - [](a, [](b, [](c, []))) : corresponds to [a,b,c].

- Is the internal structure of prolog, which you probably won't be bothered with.

### Definition

- $[t_1, \dots, t_k]$  denotes a list of  $k$  elements.
- With head  $t_1$
- $[\text{Head} \mid \text{Tail}]$  is a list with head  $\text{Head}$  and tail  $\text{Tail}$ .
- Note that:
  - The head of a list is a term.
  - The tail of a list is a list itself.
- Empty list:
  - $[]$  is the empty list.
  - Empty list has neither tail nor head.
- The head itself can be multiple terms.
  - Or multiple lists.
- To not put the tail in a list, is not valid.
- The lists may be down in a different way, but they correspond to normal lists as you know in for example Python.
- Examples:

1.  $[1 \mid [2, 3, 4]]$   
=  $[1, 2, 3, 4]$ .
2.  $[1, 2, 3 \mid []]$   
=  $[1, 2, 3]$ .
3.  $[1 \mid 2, 3, 4]$   
→ not a list
4.  $[] \mid []]$   
=  $[]]$
5.  $[[1, 2], [3, 4] \mid [5, 6, 7]]$   
=  $[[1, 2], [3, 4], 5, 6, 7]$

○

### Checking membership

- A program that checks whether a term is a member of a list.
- $\text{member}(X, L)$  should succeed if  $X$  is contained in  $L$ .
- Idea of a recursive definition:
  - $X$  is the head of  $L$ , or  $X$  is a member of the tail of  $L$ .
- Recursive member definition:
  - $\text{member}(X, [X \mid \text{Tail}])$ .
  - $\text{member}(X, [Y \mid \text{Tail}]) \text{ :- } \text{member}(X, \text{Tail})$ .
  - First one corresponds to when  $X$  is the head.
  - $Y$  is not really used in the second case.
  - You can replace the  $\text{Tail}$  variable in the upper rule and the  $Y$  in the lower rule for  $\_$ , because they are not used and become anonymous ones.
- The predicate  $\text{member}/2$  is built-in in Prolog.
- What does  $?\text{-member}(a, X)$  return?
  - It will return an instantiation of  $X$  with  $a$  at the head.
  - You can so start a list, with  $a$  as it's only member.

### List programming in Prolog



- Recurring down a list is a very important technique when programming with Prolog.
- In a lot of base clauses is the empty list.
  - We want to go through the list until there is nothing left.
- Example:
  - Define a predicate `aEqualB/2` which takes 2 lists as input and succeeds iff the lists have same lengths; the former consists only of a's; and the latter only of b's
 

```
aEqualB([], []).
```
  - ```
aEqualB([a|L1],[b|L2]) :- aEqualB(L1,L2).
```

    - As you see the base clause is empty, because we want to end up with 2 empty lists then they are equal.
  - In the left of the down rule, we say for each recursion there should be an a and b, and L1,L2 to the right are the tails which will be proceeded with.
- Example, appending lists
  - We define a predicate `append/3` with `append(X,Y,Z)` is true if list X combined with list Y is list Z.
    - The first list empty: then Z equals the second list.
    - Else, the first element of first list, must be the first element of Z, and
    - Then the append predicate is applied on the remainder of the lists recursion.
  - How it looks like:
 

```
append([], X, X).
```

    - ```
append([X|Tail1], Y, [X|Tail2]) :- append(Tail1, Y, Tail2).
```
    - In the base clause we say that the second and the third list should be the same if we see that X is an empty list.
      - If not, we say that the head of the first list X is the same as the head of the output list.
      - Y is not used.
      - We make sure that everything that is included in the first list variable is included in the third list variable.
  - For example:
    - ```
?- append([a, 2], [b, 3], [a, 2, b, b]).
```

      - Will return false.
      - It must have given a,2,b,3 to be true in the third list.
    - ```
?- append([a, b], [c, d], X).
```

      - Output is X = [a,b,c,d]
    - ```
?- append(X, Y, [a, b, c, d]).
```

      - Prolog will give several outputs which can be possible for X and Y.
    - See also the trace example in slides.
- Example (define member by append)
  - Define `member/2` by means of `append/3`
  - There is a single clause solution.
 

```
member(X,L) :- append( _, [X| _ ], L).
```

- You can leave many variables open.
  - What it will say that any member of X is a member of list L in case we see that the head of X is in one of these 2 lists, and together lead to the append of L.
- Example (last element of a list)
  - Define predicate last/2 which is true if we have the last element of the list.
  - For example last(3,[1,2,3]).
  - ```
last(X, [X]).
```
  - ```
last(X, [H|T]) :- last(X, T).
```

    - Base clause is when we are left with one element in the list, with our target value.
    - In the recursive part we go on with our element and the tail, until we reach the base clause.
- Example all same
  - Write a predicate allSame(L), where L is a list, which is true if all elements in L are identical.
  - ```
allSame([]).
```

```
allSame([X]).
```
  - ```
allSame([H1| [H2|T]]) :- H1=H2, allSame([H2|T]).
```
  - In the beneath clause we want to make sure that the length is 2 or more.
  - We specify a list and a list with a head and a tail.
    - Only the tail might be the empty list.
  - We specify that the head and the second head are the same.
    - And then we call upon the function.
  - Can be simplified:
 

```
allSame([]).
```

```
allSame([_]).
```

    - ```
allSame([X, X | T]) :- allSame([X | T]).
```
- Example length of a list
  - Define a predicate length/2 which computes the length of a list.
  - ```
% base case
```

```
length([], 0).
```

```
% recursive clause
```
  - ```
length([H|Tail], N) :- length(Tail, P), N is P + 1.
```
  - Our base clause is again the empty clause and a value 0.
    - The empty list has a value 0.
  - The result will be that it recurses, and then it has got all value of p and then add it to N.
    - We saw that left recursion was risky,
    - But now it is functional.
- Count vowels in a list:

```
?- nr_vowel([],X).  
X = 0
```

```
?- nr_vowel([a,r,e,d,i],X).  
X = 3
```

```
?- nr_vowel([m,r],X).  
X = 0
```

```
?- nr_vowel([s,e,e,d],X).  
X = 2
```

- In here we use the member predicate.

**% auxiliary predicate**

■ **vowel(X):- member(X,[a,e,i,o,u]).**

- To specify what makes for a vowel.
- We say X should be part of vowel.
- Base case:

**% base case**

● **nr\_vowel([],0).**

- Here there are 0 vowels.

■ **Recursive clauses:**

- Will iterate through the list until it reaches the base clause.
- But it will get a vowel or non vowel.
- We say if X is a vowel, then go ahead with the recursive clause but add 1 to the total.

**nr\_vowel([X|T],N) :-**

○ **vowel(X), nr\_vowel(T,N1), N is N1+1.**

- The other thing is when there is no vowel, so we use negation and then we continue with recursion, but do not add 1.

**nr\_vowel([X|T],N):-**

○ **not(vowel(X)), nr\_vowel(T,N).**

- Example (sum numbers in a list)
  - Define predicate sum/2 which sums all numbers in a list.  
**sum([], 0).**  
**sum([H|Tail], Sum) :-**
    - **sum(Tail, Tailsum), Sum is H + Tailsum.**
    - The number that we add each time is the number that we find in the head what is taken from the tailsum.
    - We do not do right recursion, because tailsum does not has been instantiated.
      - Therefore we do left recursion.

### Concluding remarks

- [Head | Tail] is a list with head Head and tail Tail.
- The head of a list is a term.
- The tail of a list is a list itself.
- Empty list ([]) has neither tail nor head.

- Recurring down a list is a very important technique when programming with prolog.
- Base clause often gives the case for the empty list or list of length 1.
- Left recursion may be done when using calculations in combination with decreasing lists.

### Variable types

- Anonymous variable
  - Denoted by `_`
  - It is different than giving some random variable.
  - An anonymous variable can take every form.
- Instantiation patterns
  - `+X`: variable X is input variable and must be instantiated when predicate is queried.
  - `?Y`: variable Y does not have to be instantiated.

```
% addThreeAndDouble(+X, ?Y).
addThreeAndDouble(X,Y):- Y is 2*(X+3).
```

- Append instantiation pattern.
  - None of the variables of the built-in predicate `append/3` needs to be instantiated.

```
Instantiation pattern:
% append(?List1, ?List2, ?List3).
```

## Week 3, college 1

### Recap on append/3

- `append(L1,L2,L3)` is true if list L3 is the result of concatenating the lists L1 and L2 together.
  - `append([],L,L).`
  - `append([H|L1],L2,[H|L3]) :- append(L1,L2,L3).`
- Recursive definition
  - Base clause:
    - Appending the empty list to any list produces that same list.
  - The recursive step says that when concatenating a non-empty list `[H|T]` with a list L, the result is a list with head H and the result of concatenating T and L.

### Using append/3 to break up a list

```
append([], L, L).
append([H|L1], L2, [H|L3]) :- append(L1, L2, L3).
```

- In this example you don't specify the first 2 variables you actually specify the 3rd variable.
- Output:

```
?- append(X,Y, [a,b,c,d]).
X=[ ]           Y=[a,b,c,d];
X=[a]           Y=[b,c,d];
X=[a,b]         Y=[c,d];
X=[a,b,c]       Y=[d];
X=[a,b,c,d]     Y=[ ];
no
```

- 
- You tell prolog to give the 2 lists that together specify the 3rd list.
  - Which are a few that can make that list.

### Using append, example sublists

- These basic predicates can be used as building blocks for more complex predicates.
- A predicate sublist(S,L) which succeeds if S is a sublist of L.
  - We want a sublist, not a cloned list.

```
sublist(S,L) :- append(L1,L2,L), append(S, L3, L2).
```

This can be rewritten as:

```
sublist(S,L) :- append(_ ,L2,L), append(S, _ ,L2).
```

- - You have some unused variables hence the \_.

### List deletion and insertion

- del(X,L,D) succeeds if D is L with one X being removed.
  1. If X is head of L=[X|Tail] then D can be Tail.
  2. Else, consider the lists' tails:

```
del(X,[X|Tail],Tail).
```

- ```
del(X,[Y|Tail1],[Y|Tail2]) :- del(X,Tail1,Tail2).
```

- We can easily define a predicate insert/3:
  - ```
insert(X, L, Ins) :- del(X, Ins, L).
```
  - Inserting an instance of something in a list is the reverse of deleting it.
  - We use delete but we swap the output and input.
  - The list we want to add our target into, will be the output of delete.
  - The query will give several options.

```
Ins = [d, a, b, c] ;
Ins = [a, d, b, c] ;
Ins = [a, b, d, c] ;
Ins = [a, b, c, d] ;
false.
```

### Yet another elegant definition of member/2:

- ```
member(X,L) :- del(X, L, _).
```
- X is a member of a list L, if it is possible to delete X from L.
- We do not need output from delete so we leave it empty.

### Permutation

- Often, problems can be solved by considering permutations of a list:
  1. The empty list is its permutation.
  2. [X|L]: permute L and insert X somewhere.
    - We want to keep permuting until we reach the empty list.

```
permute([],[]).
permute([X|Tail],P) :-
    permute(Tail,L), insert(X,L,P).
```

- - Note that this approach is not very efficient.
    - It will do several attempts that might be duplicates.
    - And you might get duplicates back many times.
  - We use left recursion.
    - To the left is only happening if we come up to the empty list and start going up the tree again.

### Accumulators, definition

- Accumulators are variables that hold intermediate results.
- Increases efficiency.

### acclen/3, length predicate using accumulators

- The predicate acclen/3 has 3 arguments
  - The list whose length we want to find.
  - An accumulator keeping track of the intermediate values for the length.
  - The length of the list, an integer.
- The accumulator of acclen/3
  - Initial value of the accumulator is 0.
    - We need to initialize the value.
  - Add 1 to accumulator each time we can recursively take the head of a list.
  - When we reach the empty list the accumulator contains the length of the list.

```
acclen([],Acc,Length):- Length = Acc.
acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

*Simplify to*

```
acclen([],Acc,Acc).
acclen([_|L],OldAcc,Length):-
    NewAcc is OldAcc + 1,
    acclen(L,NewAcc,Length).
```

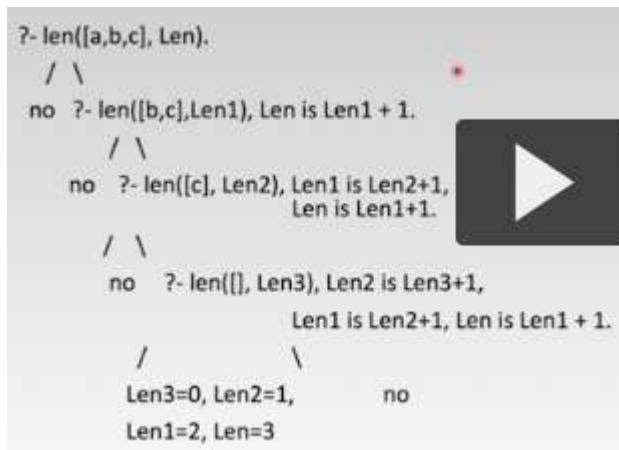
- - ?- acclen([a,b,c],0,Len).
  - If the 0 value in the call is not given it will not work properly.

### Recursion and stacking

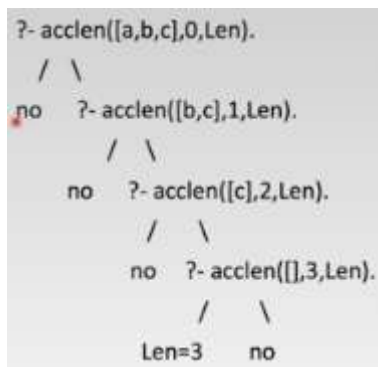
- Recursive procedures call themselves to work towards a solution to a problem.
- In simple implementations this causes the stack to grow and grow as the nesting gets deeper and deeper.
- When it reaches the solution, it returns through all of the stack frames.
- This waste is a common complaint about recursive programming in general.

## Tail recursion

- Definition:
  - A function call is said to be tail recursive if there is nothing to do after the function returns except return its value.
  - Since the current recursive instance is done executing at that point, saving its stack frame is a waste.
    - Specifically creating a new stack frame on top of the current, finished, frame is a waste.
  - This way you can write recursive definitions without worrying about space inefficiency (from this cause) during execution.
  - Tail recursion is then as efficient as normal iteration.
- Why is acclen/3 better than len/2
  - In tail recursive predicates the results are fully calculated once we reach the base clause.
  - In recursive predicates that are not tail recursive, there are still queries on the stack when we reach the base clause.
- Search tree for len/2:



- Search tree for acclen/3



- Adding a wrapper
  - A wrapper is given with an arity of 2, and we can call upon this accumulator predicate:

■ `length(List,Length):- acclen(List,0,Length).`

## Comparing numbers

- We are going to define a predicate that takes 2 arguments, and is true when:

- The first argument is a list of integers.
- The second argument is the highest integer in the list.
- Basic idea
  - We will use an accumulator
  - The accumulator keeps track of the highest value encountered so far.
  - If we find a higher value, the accumulator will be updated.

```
accMax([H|T],A,Max):- H > A, accMax(T,H,Max).
accMax([H|T],A,Max):- H <= A, accMax(T,A,Max).
accMax([],A,A).
```

- Either the accumulator value is lower than the head of the list, or it is higher or the same.
 

```
?- accMax([1,0,5,4],0,Max).
Max=5
```
- We cannot add a wrapper to start the accumulator.
  - Our answer would then not be correct, because you could have values below 0.
  - The best way to define the wrapper to start of with the head of the input list.
    - `max([H|T],Max):- accMax(T,H,Max).`
    - This also gives the highest negative value.

### Append/3 and efficiency

- Append/3 as a source of inefficiency:
  - Concatenating a list is not done in one simple action.
  - But by traversing down one of the lists.
- The order makes a difference in how efficient the program works.
- Suppose we want a list that has all the elements of [a,b,c,d,e,f,g,h,i] and [j,k,l].
  - What is more efficient way of getting this?
 

```
?- append([a,b,c,d,e,f,g,h,i],[j,k,l],R).
?- append([j,k,l],[a,b,c,d,e,f,g,h,i],R).
```

    - It is more efficient to put the longer list in the second variable.
      - In the second variable there is no further processing done other than unifying.
        - The recursion focuses on the first argument, not really touching the second argument.
        - That means it is best to call it with the largest list as first argument.
      - In the first variable we have to iterate through the list.
        - You don't always know what the shortest list is, and you can only do this when you don't care about the order of the elements in the concatenated list.

### Reversing a list

- We will define a predicate that changes a list [a,b,c,d,e] into a list [e,d,c,b,a].
- This would be a useful tool to have, as Prolog only gives easy access to the front of the list.



- And sometimes you want to start at the tail.
- Naïve reverse
  - Recursive definition
    - If we reverse the empty list, we obtain the empty list.
    - If we reverse the list  $[H|T]$ , we end up with the list obtained by reversing  $T$  and concatenating it with  $[H]$ .
  - To see that this definition is correct, consider the list  $[a,b,c,d]$ :
    - If we reverse the tail of this list we get  $[d,c,b]$ .
    - Concatenating this with  $[a]$  yields  $[d,c,b,a]$ .
  - Naïve reverse:
 

```
naiveReverse([],[]).
naiveReverse([H|T],R):-
    naiveReverse(T,RT), append(RT,[H],R).
```

    - This definition is correct, but it does an awful lot of work.
    - It spends a lot of time carrying out appends.
    - There is a better way.
- Reverse using an accumulator
  - The better way is using an accumulator
  - The accumulator will be a list, and when we start reversing it will be empty.
  - We simply take the head of the list that we want to reverse and add it to the head of the accumulator list.
  - We continue this until we reach the empty list.
    - At this point the accumulator will contain the reversed list.
  - In Prolog:
 

```
accReverse([],L,L).
accReverse([H|T],Acc,Rev):- accReverse(T,[H|Acc],Rev).

reverse(L1,L2):- accReverse(L1,[],L2).
```

    - And has the process:
 

|                   |                          |
|-------------------|--------------------------|
| List: $[a,b,c,d]$ | Accumulator: $[]$        |
| List: $[b,c,d]$   | Accumulator: $[a]$       |
| List: $[c,d]$     | Accumulator: $[b,a]$     |
| List: $[d]$       | Accumulator: $[c,b,a]$   |
| List: $[]$        | Accumulator: $[d,c,b,a]$ |
- Remember that accumulators are tail recursive.

## Week 4, video's working of goal

- mas2g stands for multi agent system to goal.
- Then we have mod2g which stands for module to goal.
  - The main logic of the agent can be found.
- Prolog is the knowledge the agent uses.
  - Represents the knowledge representation.
- You always need to use for an agent at least an empty knowledge file.
- There is also an option for a debug run.
  - Then you get a debug perspective.

- There you also can pause it by clicking on the agent and clicking the pause button.
- Then the code is green which is currently viewed and executed.
- At the top right you can see the agents' states.
  - Are in order of which they are added.
- Also percepts can be seen by the environment.
- With step-into you go to the next line of code which will be executed.
- You can also query the current state of the agent.

## Week 4, College 1

- See recap Intelligent agents from starting lecture.
- An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors.

### Cognitive agents and environments

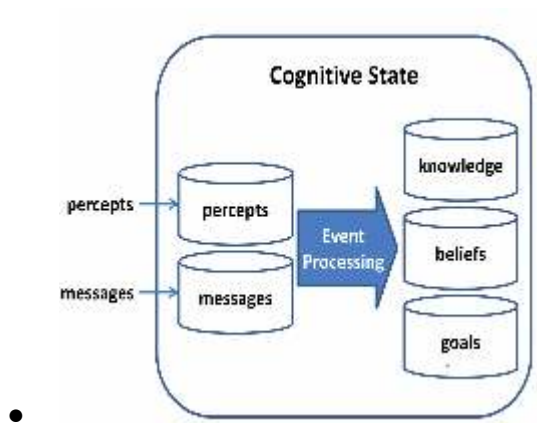
- For cognitive agents we take the view the environment offers controllable entities which are connected to cognitive agents.
- Possible view:
  - Cognitive agent is the mind.
  - Controllable entity is the body.
- ...

### Cognitive agent programming

- A cognitive agent language is a programming language with events, beliefs, goals, actions, plans,...as first class citizens.

### Cognitive state

- The internal state of a cognitive agent is called a cognitive state.
- Typically it includes:
  - Event component
    - Percepts
    - Messages
  - Informational component:
    - Knowledge (static)
      - Prolog
    - Beliefs (dynamic)
  - Motivational component (what the agent wants to achieve):
    - Goals.



### Key language elements

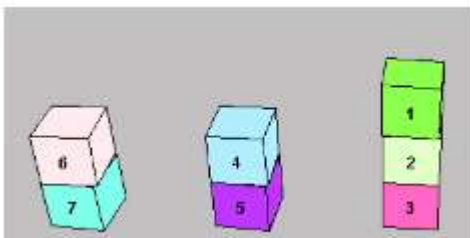
- Events received from environment (and agents).
- Beliefs and goals to represent environment.
- Actions to update beliefs, adopt...
- Cognitive states used for:
  - Representing and reasoning about the environment.
  - Deriving...

### The blocks world

- Objective: move blocks in initial state such that result is goal state.
- Positioning of blocks on table is not relevant.
- A block can be moved only if there is not other block on top of it (clear(X)).

### Cognitive state

- To represent and reason about environment.
  - What agent believes in the case.
  - What the agent wants the environment to be like.
- Need a language to represent the environment.
- Prolog is the knowledge representation language we will use.
- Using the  $on(X,Y)$  predicate we can represent the initial state.



- We translate them to beliefs.

#### beliefs:

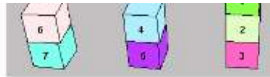
```
on(b1,b2).  
on(b2,b3).  
on(b3,table).  
on(b4,b5).  
on(b5,table).  
on(b6,b7).  
on(b7,table).
```

- Initial belief base of agent
  - Blocks have names to be able to uniquely identify them.

- We have the following knowledge to represent the blocks world:

Basic predicate:

`on(X,Y).`



Defined predicates:

```
block(X) :- on(X, _).  
clear(X) :- block(X), not(on(_,X)).  
clear(table).  
tower([X]) :- on(X,table).  
tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
```

- tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
- The last tells us that there is room to put block on table.
- In the last we recursively define towers.
- We also say :- dynamic on/2
  - So that we can change it.
- None of these clauses do change.
  - They are in the knowledge base and don't change.
  - So what we have defined is knowledge.

### Dynamic predicates

- Recall from a previous lecture the definition of a predicate:
  - The set of facts which use the predicate, plus
  - The set of all rules the head which matches with the predicate.
- A knowledge base can use predicates which are not defined, for example they occur as facts only in the belief base as on/2 below.
- We use the rule :- dyna...

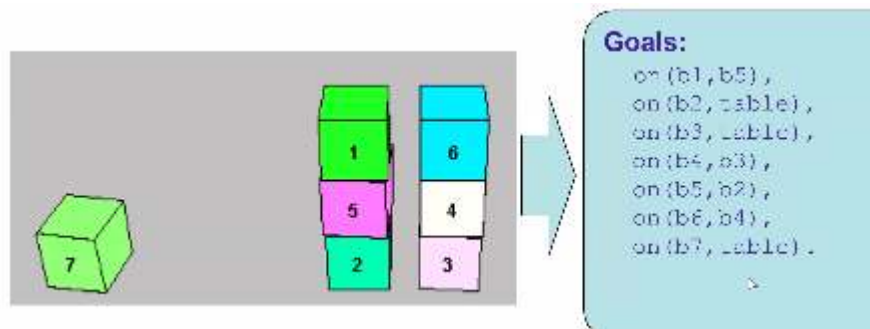
### Background knowledge

- Domain knowledge:
  - Knowledge about the domain.
    - For example block(X) :- on(X,\_).
- Conceptual knowledge:
  - Knowledge consists of additional knowledge about relevant predicates and also definitions such as for example:
    - Tower([X]) :- on(X, table).
    - ... rest of recursive thing as said previous.
  - It is more independent of the environment than domain knowledge.

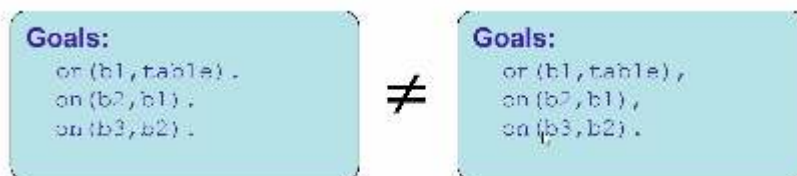
- ...

### Ey KGOAL feature: declarative goals

- What is a declarative goal?
- Agents may have multiple goals that specify what the agent wants to achieve at some moment in the near or distant future.
- Declarative goals specify a state of the environment that the agent wants to establish, they do not specify actions or procedures how to achieve such states.
  - Example: I want to be rich in 10 years.
  - Does not say how to get rich.
- Using the on(X,Y) predicate we can represent the goal state.



- 
- We separate these facts by komma's.
  - It is a conjunction.
- One or many goals
  - In goals section, using the comma- or dot operator makes a difference.



- Left goal base has 3 goals, right has a single goal.
  - If the goals on the left one are achieved, the right one is not.
  - The reason is that the goal base on the left does not require block b3 to be on b2, b2 to be on b1 and b1 to be on the table at the same time.

- The knowledge, beliefs and goal sections together provide a specification of the cognitive state of a GOAL agent.

#### Knowledge:

```
:- dynamic on/2.  
block(X) :- on(X, _).  
clear(X) : block(X), not(on(_,X)).  
clear(table).  
lower([X]) :- on(X,table).  
tower([X,Y|I]) :- on(X,Y), tower([Y|I]).
```

#### Beliefs:

```
on(b1,b2). on(b2,b3). on(b3,table). on(b4,b5).  
on(b5,table). on(b6,b7). on(b7,table).
```

#### Goals:

```
on(b1,b5), on(b2,table), on(b3,table), on(b4,b3),  
on(b5,b2), on(b6,b4), on(b7,table).
```

### Creating cognitive states

- Content of each module depends on the knowledge representation language, in our case prolog.
- Thus knowledge beliefs and goals are simple prolog file...

KnowledgeFile.pl

#### A valid Prolog program

Note: all predicates must be **defined/declared**.

BeliefsFile.pl

#### A valid Prolog program, often just facts.

GoalsFile.pl

#### A query (conjunction) of positive facts.

- The goals are not a valid prolog program.
- Note:
  - Negation facts are not allowed in a prolog program (so cannot be part...

### Why a separate knowledge base

- Concepts defined in the knowledge base can be used in combination with both the belief and goal base.

#### Example

- If agent **believes** `on(b5,table)`, `on(b4,b5)`, then infer:  
agent **believes** `tower([b4,b5])`.
- If agent **wants** `on(b1,table)`, `on(b2,b1)`, then infer:  
agent **wants** `tower([b2,b1])`.
- Knowledge base avoids duplicating clauses in belief and...

### Using the belief and goal base

- Selecting actions using beliefs and goals.
- How to encode reactive and proactive behaviour?
- Basic idea:
  - If the agent believes B then do action A (reactively).
  - If the agent believes B and has goal G, then do action A (proactivity).

### Inspecting the belief and goal base

- Operator `bel(s)` to inspect the belief base.
- Operator `goal(s)` to inspect the goal base.
  - Where `s` is a prolog query (conjunction of literal).
- Not `bel(s)` and not `goal(s)` is true if something is not believed or not a goal.

#### Examples:

- `- bel(clear(a), not(on(b1,b3))).`
- `- goal(lower([b1,b2])).`
- Note that `bel` belongs to the goal language where `s` is a query in prolog.
- `Bel(s)` succeeds if `s` follows from the belief base in combination with the knowledge base.
  - Condition `s` is evaluated as a Prolog query.

#### Example:

- `- bel(clear(b1), not(on(b1,b3))) succeeds`
  - See in the slide.
- `Goal(s)` succeeds if `s` follows from one of the goals in the goal base in combination with the knowledge base.

#### Example:

- `- goal(clear(b1)) succeeds.`
- `- but not goal(clear(b1), clear(b3)).`
  - See also the info on slide.

### Negation and beliefs

- `not(bel(on(b1,b3))) == bel(not(on(b1,b3)))`
  - It is the same.
- `not(goal(s)) == goal(not(s))?`
  - Not the same.
  - `goal(not(on(b2,table)))`, but not `not(goal(on(b2,table)))`.

### Combining beliefs and goals

- Consider the following beliefs and goals:

```
Beliefs:
on(b1,b2). on(b2,b3). on(b3,table).

Goals:
on(b1,b2). on(b2,table).
```

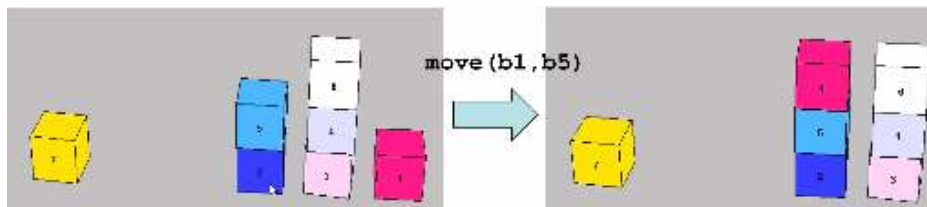
- 
- We have both `bel(on(b1,b2))` as well as `goal(on(b1,b2))`.
- Why have something as a goal that has already been achieved?
  - It is not the complete goal, it is only part.
  - The second part is not realized yet.
- `a-goal(s) = goal(s), not(bel(s))`



- Agent only has an achievement goal if it does not believe the goal has been re...
  - Is not yet achieved.
- goal-a(s) = goal(s), bel(s)
  - Goal achieved ...
- We need to know when block x is misplaced.
  - A block is misplaced if it is not in the position we want it to be.
  - Solution:
    - Goal(tower([X|T])), not(bel(tower([X|T]))).
    - So we have this goal, but we do not yet believe it exists.
    - But this means that saying that a block is misplaced is saying that you have an achievement goal:
      - a-goal(tower([X|T])).
- Block x can be moved into position
  - A block can be moved into position if it can be stacked on top of the tower that we want it to be on.
  - Solution:
    - a-goal(tower([X,Y|T])), bel(tower([Y|T]), clear(X), clear(Y)).
    - We say that X is still misplaced.
    - a-goal(tower([X,Y|T])), bel(tower([Y|T]))
      - Note: second solution if action move has precondition clear(X), clear(Y)
    - If a block X can be moved into position, we also say that a constructive move can be made (with block X).

### Actions specifications

- Actions change the environment.



- To ensure adequate beliefs after performing an action the belief base needs to be update.
- Add effects to belief base: insert on(b1,b5) after move(b1,b5).
- Delete old beliefs: delete on(b1,table) after move(b1,b5).
- If a goal has been completely achieved, the goal is removed from the goal...
- If a goal has been completely achieved (believed to be), the goal is removed from the goal base.
  - IT is not rational to have a goal you believe to be achieved.
  - Default update implements a blind commitment strategy; that is goals only removed when believed to be achieved.

### Action specification

- Actions in GOAL have preconditions and postconditions.
- Executing an action in GOAL means:
  - Preconditions are conditions that need to be true:

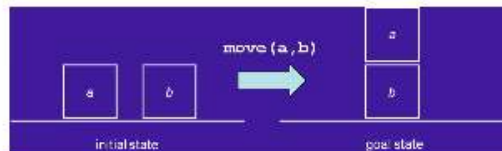


- Check preconditions on the belief base.
- Postconditions (effects) are add/delete lists(STRIPS):
  - Add positive literals in the postcondition.
  - Delete negative literals in the postcondition.

```
define id(parameters) with
  pre { query }
  post { update }
```

- - Precondition is checking if we believe the fact.
- An action is said to be enabled if the precondition holds.
- All...
- Example:

```
define move(X,Y) with
  pre { clear(X), clear(Y), on(X,Z), not( on(X,Y) ) }
  post { not( on(X,Z) ), on(X,Y) }
```



- - Tells us that we can only move a block when it is clear and we can only put the block on something when it is clear.
- The move name should match with something that makes the environment change.
- First facts are removed, and then they are added.

### Durative and instantaneous actions

- External actions not only update the agent's belief base but are also sent to the environment.
- In the environment the action is forwarded to the entity.
- The entity may need time to execute an action:
  - Durative action:
    - Need time which cannot be neglected.
  - Instantaneous action:
    - Are performed immediately.
- Durative actions may fail or cancel each other.
- Postconditions are executed immediately.
- Rule of thumb: use post-condition true with durative action.
- Lesson: be careful with using post-conditions in combination with durative.

### Action specification files

- Implicitly we considered action specifications together...
- See slides for this and built-in internal actions.
- Goal provides a number of built-in actions:

- `insert(<conjunction>)`  
*example:* `insert(on(X,Y), not(on(X,Z)))`  
*meaning:* **add** `on(X,Y)` and **delete** `on(X,Z)`
- `delete(<conjunction>)`  
*example:* `delete(on(X,Y), not(on(X,Z)))`  
*meaning:* **delete** `on(X,Y)` and **add** `on(X,Z)`
- `adopt(<conjunction of positive literals>)`  
*precondition:* agent does not believe query and does not have the goal.  
*example:* `adopt(on(a,b), on(b,c))`  
*meaning:* add **new** goal (if not already **implied** by a goal nor believed)
- `drop(<conjunction>)`  
*example:* `drop(on(b,a), not(on(c,table)))`
  - *meaning:* **remove all goals that imply <conjunction>** from the goal base
  - Belief base affected, sometimes also goal base affected.
- Further built-in actions
  - `print(term)`: prints term term to the standar console.
  - `log(parameter)`: writes logging information to file.
    - Possible options:
      - `log(bb)`: content of belief base is written to file.
      - `log(gb)`: content...

**Note**  
All actions need to be fully instantiated when they are executed.

### Multiple actions in rule

- Using the + operator multiple actions can be combined and performed in a single rule.
  - `<action1> + <action2> + ...`
- Executed in same reasoning cycle.
- combined actions are executed in order (left to right).
- ...
- Insert and delete actions always executes.

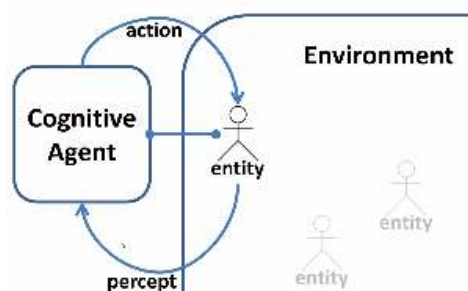
### Decision rules

- Agent oriented programming
  - How do humans choose and/or explain actions?
  - Examples:
    - I believe it rains, so i will have an umbrella with me.
  - Use intuitive common sense concepts:
    - Beliefs + goals → actions.
- Decision rules
  - Decision rules are if-then conditions:
    - `if <cognitive-state-condition> then <action>.`  
*Example:* `if bel(on(X,Y)), not(Y=table!) then move(X,table).`
  - Evaluating a rule:

- Evaluate cognitive state condition: check if condition can be derived from current cognitive state.
  - Check precondition of action.
  - Select instantiation of variables for which cognitive state condition and precondition succeed.
  - Then perform instantiated action.
    - By default, left-most instantiation is selected.
    - Then perform instantiated action move.
- A decision rule is called applicable iff.
  - The cognitive state query is true and at least ...
- Rules with composed actions.
  - A rule can also contain composed actions:
    - ...
- Devising a strategy
  - Decision rules are used to define a strategy for action selection.

## Week 5, college 1

### MAS files: launching agents and environment



- Environment section:

```
use "blocksworld-1.2.0.jar" as environment with start="bwconfigEx1.txt".
```

- Specifies connector file, used for connecting the GOAI runtime and/or launching the environment.
- Specifies initialisation parameters, different for different environments.
- It is not the environment itself.
  - Just a connector file.

- Agent definitions:

```
use "blocksworld-1.2.0.jar" as environment with start="bwconfigEx1.txt".

define BwAgent as agent {
  use startin1000 as main.
  use bwEvents as event.
}
```

- Specifies agent name.
  - Use clauses which reference init, main, and/or event module; each agent definition must have at least one main or event module.
- Launch policy

```

use "Blockworld-1.2.2.jar" as environment with start="bwconfig2x1.txt".

define BWagent as agent {
  use "BlockBuilder" as main.
  use bwEvents as event.
}

launchpolicy {
  when * launch BWagent with name = stackBalancer.

```

- Launch rule(s): '(when <cond>) launch <name> (with...)'.
  - Condition '\*': rule applicable when arbitrary entity created.
  - Condition 'type=<id>': rule is applicable if entity is of type id.
  - Agent name <name> must have been defined.
  - Agent name can be changed by 'with name' phrase.
- Agent names:
  - 'with name=<name>' launches an agent with a given name.
  - 'with name = \*' uses entity name (given by environment).
  - Name of agent definition is used as agent name otherwise.
  - 'name = <name>' can also be used in condition rule.

- Counting constraints

```

launchpolicy {
  launch coffeeMaker order max = 4.
  launch coffeemaker with number = 3.

```

- Example of unconditional launch rules.
- 'number' specifies that 3 agents are created at once, all connected to the same entity if launch rule is conditional.
- The max parameter restricts the number of times a rule is executed (higher priority than number).
- Launch rules are evaluated in order.
- For complete picture of program see slide.

## The towerworld environment

1. Fully observable:
  - a. Agent can see everything relevant for decision making.
2. Deterministic:
  - a. Action effects are predictable.
  - b. Is the case for BW4T.
3. Dynamic:
  - a. Environment can change even if agent does nothing because user can interact via GUI.
4. Discrete:
  - a. Finite list of blocks;
  - b. nb: continuous gripper movement not relevant for decision making.
5. Can be both single as well as multi-agent (possible to connect multiple agent to gripper entity).
- We need to keep track of percepts for number 3.

## Need for sensing

- Agents need sensors to:

- Explore an environment when they have incomplete information.
  - Like BW4T
- Keep track of changes in the environment that are not caused by itself or unpredictable.

## **Sensing**

- Goal agents sense the environment through percepts...

## **Initially agent knows nothing**

- In most environments, an agent initially has no information about the state of the environment.
- Agent has an empty belief base.
- There is no need...
- Percepts are received by an agent in its percept base.
- Percepts are accessed by reserved keyword `percept(<query>)`.
- Not automatically inserted into belief base.
- Different from `bel` the `percept` operator does not use an agent's knowledge base.

## **Processing percepts**

- The percept base is refreshed.
  - i.e. emptied and filled again, every cycle of the agent.
- Agent has to decide what to do when it perceives something
  - I.e. receives a percept
- Use percepts...

## **Rules for processing percepts**

- Rules for event processing are called event rules.
- They are placed in the event module.
- Event module is executed every time that an agent receives (new) percepts.
- By default all rules in event module are processed.

## **Updating agent's cognitive state**

- One way to update beliefs with percepts:
  - First, delete everything agent believes.
  - Example: remove all `block` and `on` facts.
- Second, insert new information about current state provided from percepts into belief base.
  - Example: insert `block` and `on` facts for every `percept(block(...))` and `percept(on(...))`.
- Assumes that environment is fully observable with respect to `block` and `on` facts which is not the case in BW4T so do not use this.
  - Also not very efficient.

## **Percept update patter**

- A typical pattern for updating is:

- Rule 1**
- If the agent
- **perceives** block X is on top of block Y, and
  - does **not believe** that X is on top of Y
- Then **insert** on (X, Y) into the belief base.
- 
- Rule 2**
- If the agent
- **believes** that X is on top of Y, and
  - does **not perceive** block X is on top of block Y
- Then **remove** on (X, Y) from the belief base.

- 
- Assumes full observability with respect to on facts.

### Implementing pattern rule 1

- Event module:**
- ```
% assumes full observability.
if percept(on(X,Y)), not(bel(on(X,Y))) then insert(on(X,Y)).
```
- We want to apply this rule for all percepts instances that match it.
- We want to apply this rule for all percept instances that match it!*
- Content Percept Base**
- |                      |       |                     |
|----------------------|-------|---------------------|
| percept(on(a,table)) | ----> | insert(on(a,table)) |
| percept(on(b,table)) | ----> | insert(on(b,table)) |
| percept(on(c,table)) | ----> | insert(on(c,table)) |
| percept(on(d,table)) | ----> | insert(on(d,table)) |
| ...                  |       | ..                  |
- Event module:**
- ```
% assumes full observability.
forall bel(percept(on(X,Y)), not(on(X,Y))) do insert(on(X,Y)).
```
- 

### Implementing pattern rule 2

#### Rule 2

- If the agent
- **believes** that X is on top of Y, and
  - does **not perceive** block X is on top of block Y
- Then **remove** on (X, Y) from the belief base.

- Event module:**
- ```
% assumes full observability.
forall percept(on(X,Y)), bel(not(on(X,Y))) do insert(on(X,Y)).
forall bel(on(X,Y)), not(percept(on(X,Y))) do delete(on(X,Y)).
```
- - See slide for clear picture.
    - Also for example for event module in the blocks world.
    - Also for assignment.
  - Assumes full observability.
  - We want that all rules are applied
  - Note that none of these rules fire if nothing changed.

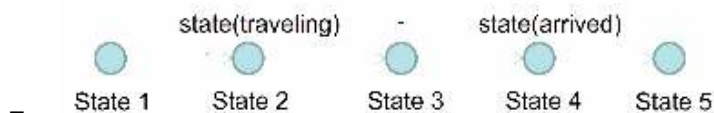
### Types of percepts

- Processing percepts

- Types of percepts:
  - ‘send always’
  - “send once”, for example place(<PlaceID>).
    - Typically used for a non changing environment.
  - “send on change”, for example at(<PlaceID>).
    - If information has been updated.
    - A robot might be at another place if it moves.
- How to handle these different type of percepts?
- Pattern for send once
  - “send once” percepts are sent only once when the agent first connects to an environment.
  - Use the init module to insert the beliefs you want to use in the agent’s belief base.

```
forall percept( place(x) : do insert( place(x) );
```

- “Send on change” percepts are sent once when a change occurs in the environment:
  - We have for example this timeline:



- In state 3 we assume the agent is still travelling.
- This happens when you use a goTo action.
- We don’t want to do anything on state 3, but on state 4 we want to act that we have arrived.

- Rule: remove old belief and insert new percept.

```
Event module:

forall bel( state(Old) ), percept( state(New) )
do delete( state(Old) ) + insert( state(New) );
```

- Put rule in event module.
- instead of forall in this rule could also have used if there, at most one state percept received.

- Combining information
  - In BW4T it is important to remember in which place a colored block can be found.
  - Idea: combine at location with ‘color’ percept using a new ‘block’ predicate.

```
Event module:

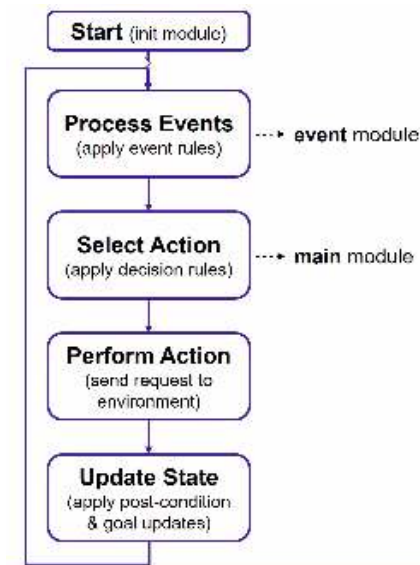
forall percept( color(BlockID, ColorID) ), percept( at(PlaceID) )
do insert( block(BlockID, ColorID, PlaceID) );

forall bel( at(PlaceID), block(BlockID, ColorID, PlaceID) ),
percept( not( color(BlockID, ColorID) ) )
do delete( block(BlockID, ColorID, PlaceID) );
```

- For clear picture see slide 33.
- You gain info of the color, place and id of the block.
- NB: make sure by earlier rule that “at” belief is correct.

## Agent execution cycle





- Start: init module:
  - Process “send once” percepts
  - Initialize cognitive state.
  - Executed once at the beginning.
- After the init module the agent starts a cycle.
- Process: event module
  - Process percepts
  - Process messages
  - Goal management
- Select: main module
  - Action selection (strategy).
- Action: request execution
- Update: cognitive state

### Other rule evaluation orders

```

if a-goal(tower([X,Y]), bel(tower([Y])) then move(X,Y).
if a-goal(tower([X-1])) then move(X, table);
  
```

- Default order:
  - Main module is linear (a rule starting from top\_.
  - init and event module is linearall (all rules starting from top).
- The order can be changed as well as the exit conditions.

### Summarizing

- 2 types of rules:
  - if <cond> then <action>.
    - is applied at most once (if multiple instances chooses randomly).
  - forall <cond> do <action>.
    - Is applied once for each instantiation of parameters that satisfy condition.
- Main module by default:
  - Checks rules in linear order.
  - Applies first applicable rule (also checks action precondition!).



- ...

## Durative actions

- Instantaneous versus durative
  - Instantaneous actions
    - Actions in the blocks world environment are instantaneous.
    - They do not take time.
      - Move a block action is of this type.
  - Durative actions
    - Actions in the tower world environment take time.
    - When a goal agent sends an action to such an environment, the action...
- Durative actions
  - Deciding on a next action typically gets more complicated when action take time.
  - Agents may decide to perform an action before finishing another.
  - Three cases:
    - Sending an action B while another action A is still ongoing overrides and terminates previous action A.
    - Action B that is sent while another action A is still ongoing is performed in parallel.
      - Both actions are performed.
      - For example some bot can shoot and run (think of games).
    - Action B that is sent while another action A is still ongoing is simply ignored.
      - Nothing new happens.
      - Less frequent.
- Durative actions and sensing
  - While durative actions are performed an agent may receive percepts.
  - Useful to monitor progress of action.
  - Tower world example:
    - Gripper may no longer be holding block because user removed it.
  - BW4T example:
    - Other robot is blocking entrance to room.
- Specifying durative actions
  - Tower world has 2 durative actions:
    - Pickup and putdown block both take time.
  - How should we specify these actions?
    - Delayed effect problem.
  - Solution:
    - Do not specify postcondition,
    - Make sure action effects are handled by event rules.
  - Tower world action specification:

```

pickup(X) {
    pre: clear(X), not(holding(X))
    post: true
}

putdown(X,Y) {
    pre: holding(X), clear(Y)
    post: true
}

nil {
    pre: true
    post: true
}

```

- 
- Postcondition may also be empty: post { }
- Better practice is to indicate that you have not forgotten to specify it by using post { true }.
- Multiple environment actions
  - Warning: it may not be useful to combine environment actions.
  - Environment actions may cancel each other out.

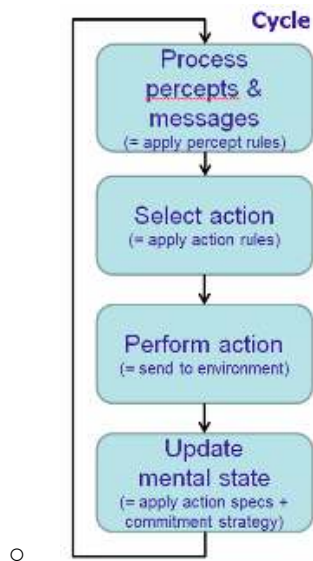
### The design of agent program

- First understand the environment
  - Actions, percepts
  - How to represent logic (knowledge), how to track changes (beliefs), and how to represent goals.
- Ontology
- Designing the predicates (labels)
- Declarative (what, not how)
- Iterative refinement.
  - Refine it over time.
- A goal which describes an action is not good.
  - killWumpus is bad
  - wumpusIsDead is good.
- Then design the strategy
  - Event rules, action specifications
  - Action selection strategy, decision rules.
- Code readability
  - Comments
    - Meaning of predicate
    - Explaining action specification
    - Purpose or role of module
    - Purpose of (groups of) rule(s).
  - Modules
    - Intuitive labels to group similar rules together.
    - For example for decisions, percepts, communication and goal management.
    - ...

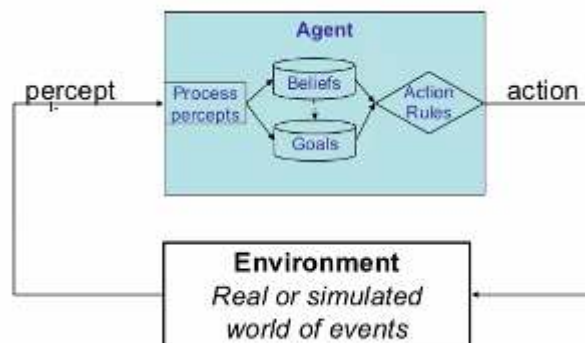
## Week 5, college 2 Guest lecture tools for developing cognitive agents.

### Cognitive agents (recap)

- Decision cycles.



- External environments
  - Like the environment of BW4T.
- We can have multiple agents.
  - Who are communicating with each other.
- Knowledge representation (cognitive state)
- Rule-based reasoning.



- - Goals might influence actions and other way around.
  - A wrong belief or goal might influence an action.
  - With multiple agents these cycles go on for many agents.

## Debugging

- Detecting, locating and correcting faults (bugs).
- Observable (mis)behaviour of system(reasoning)
- Program comprehension
- Significant effort is in debugging!

## Debugging techniques

- Logging
- Assertion
- Source-level debugging
  - Stepping through your program on what is going on.

- Reading the code
- Static checks
- Etc.

### Source-Level debugging for GOAL

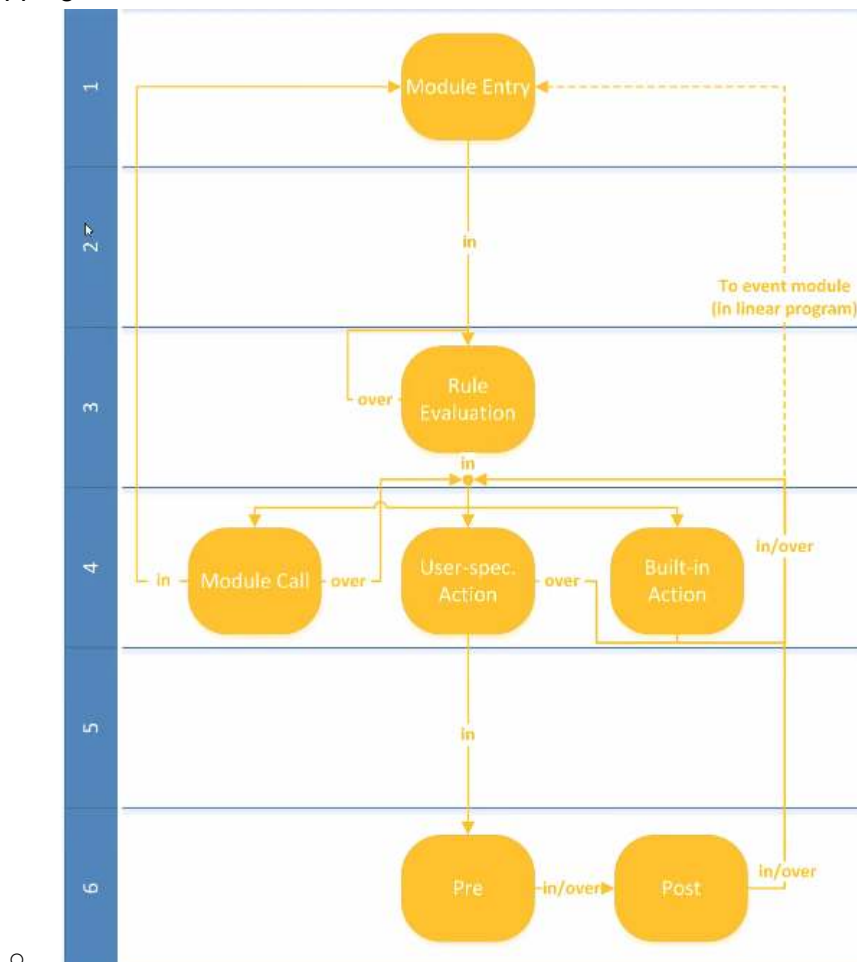
- Visualizing the execution
  - Highlight code that is to be executed.
  - Show relevant evaluations.

### Pre-defined breakpoints

- Code-based breakpoints:
  - Have a location in the code (based on syntax).
- Cycle-based breakpoints:
  - Do not (always) have a location in the code.
  - For example receiving percept, achieving a goal, etc.

### Debug environment

- Step into: go to next step
  - To the left of the arrows in debug environment.
- You can query things in the interactive console about beliefs and goals.
- Stepping flow:



- Step over steps over rules until an action is performed.
- Return means you go to a previous rule, so backtracking.

- Using the stepping flows allows you to go quickly to the place you want to be.

### User-defined breakpoints

- Regular (red): placed on 1st code-based bp.
  - The agent pauses there and from there you can go on stepping.
- The other one (yellow) is only applied when the condition rule applies.
- You can only see beliefs and Goals in that list if you pause the bot.
  - Would be going too fast if it is running.
  - You can also add expressions, which show always were something is for example.
- You can also set preferences for logging

### Additional tooling (don't learn)

- Automated testing
  - Complementary to debugging
    - More efficient
    - Less subjective
    - Facilitates repetition
  - Failure detection > fault localization
  - Test conditions
    - Properties of a module:
      - **always**  $sc$
      - **never**  $sc$
      - **eventually**  $sc$
      - $sc1$  **leadsto**  $sc2$
- Omniscient debugging
  - 'back-in-time' debugging allows exploring a (test)run by recording the execution.

Safety

Liveness

## Week 6, college 1

### Modules

- A tool to structure programming.
  - Create some order.
- Modules play a key role in the execution cycle of the agent.
  - Think of the execution cycle discussed earlier.
- The modules are the files like init, event and main files.
- Init and event module by default:
  - Checks rules in linear order.
  - Applies all applicable rules.
- Main module by default:
  - Checks rules in linear order.
  - Applies first applicable rule.

### Rule and option ordering

- Code generates some options of rules that can be executed.
  - More than 1 rules can at one point be executed.
- order= ...specifies the order in which rules and options are selected:
  - linear: execute first applicable rule
  - linearall: execute all applicable rules in linear order.
  - random: select one of the options randomly.
  - linearrandom: select option of first applicable rule randomly.
  - randomall: like linear all but rules and option selection is done randomly.

### Using modules

- So far, we used modules as init, main or event module.
- Any module can be used as an action.
- Include module with a use clause.
- Refer to action by calling the module, e.g. clearblocks.
- A module is
  - entered
  - executed
  - exit-ed
- A module becomes active when it is entered.
- Only rules in an active module can generate options.
- Modules have no precondition, that is, a rule.
  - if true then action
    - is in general different from a rule
      - if true then module
- Modules can be nested recursively.

### Modules can have variables too

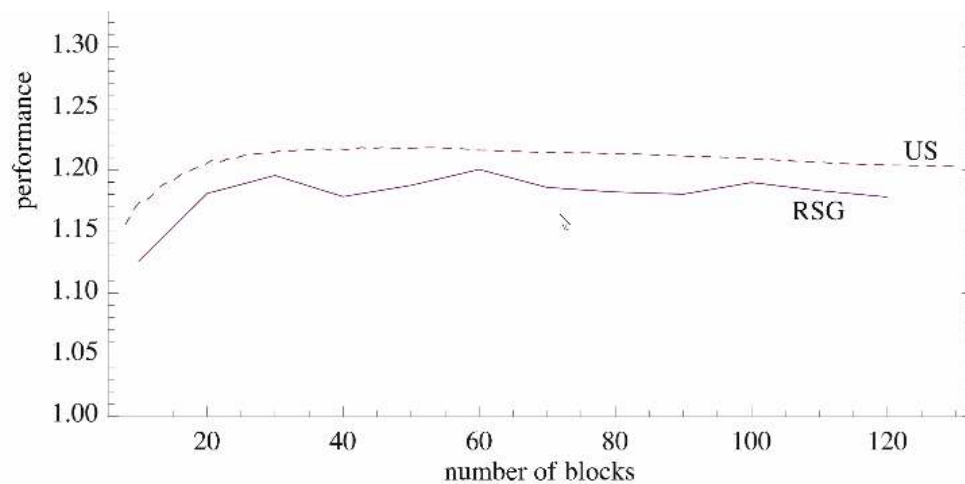
- Variables are instantiated throughout module.

```
module clearBlocks(X) {  
    if bel(on(X, Y), block(Y)) then move(X, table).  
}
```

- 
- Exiting a module: exit condition
  - exit=
  - Always: the module is executed once and then control is handed back to calling module (default)
  - never: module is never excited (use only at top-level)
  - noggoals: no goals to pursue in the module
  - noaction: exit if no more options are generated by the module.
- The built-in action exit-module exits the module immediately.
- In all these cases the agent execution cycle is still executing.

### Rule and option ordering

- US: simple unstacking strategy for the blocks problem (move to table, then stack).
- RSG: average performance random rule order.



- - These type of choices affect efficiency on program.

### Modules: decision rules

- When a module is entered:
  - Only decision rules...

### Modules

- Useful for structuring code.
- Effective for specific situation.
- Combination of conceptual knowledge, goals, and actions for a specific situation.
- Hiding information → readability.
- Supports reusability.

### Creating focus

- Focus
  - What if the agent has several goals?
  - They may not be achievable simultaneously.
  - How to decide which goal to pursue
  - Idea: focus on one goal at a time using modules.
  - The agent selects a goal and puts it in its attention set which is a new goal base associated with a module.
- You can get 2 goals which contradict each other and you end up with endless loop.
  - Like one goal putting a block on table and the other one putting it back.

### Modules: focus of attention

- Focus option of module creates new attention set ('local' goal base):
  - focus=select
- Cognitive state condition that trigger modules act like a filter.
- One of the (possibly multiple) goals that satisfies the condition is put in the attention set of the module.
- If multiple goals satisfy the condition, either one of them may be selected and put in the attention set.

### Modules and goal bases

- An attention set functions like a regular goal base.

- Cognitive state conditions about goals used within a module are evaluated on the active attention set.

### Focus option

- Set filter condition using
  - focus=none: no new attention set, global goal base used (default)
  - focus=new: new empty attention set is used
  - focus=select: new attention set with a selected goal
    - if <cognitive-query> then module
    - A goal of the active goal base is selected which satisfies the query <cognitive-query>.
    - Note only goal operators count.

### Selectors: 'this' and 'self'

- Modules: adopt and drop
  - Semantics of the built-in adopt and drop action within modules:
- Adopt action within module:
  - Adds goal to the current attention set.
  - Only local effect.
  - adopt(...) is the same as this.adopt(...)
- drop action within module:
  - Removes goal from all attention sets
  - Has global effect.

### Selectors this and self

- Adopt adds goals to the active attention set.
- By using the selector self the goal is added to the top-level goal base:
  - self.adopt(query)
- The selector can also be added to goal queries:
  - goal...

### Communication

- It is possible that agents cannot perceive changes in available resources; they have to communicate this.

### Send action and mailbox

- send action:
  - (agent).send(Msg) sends message Msg to Agent.
- Cognitive state primitive:
  - (Agent).sent(Msg) inspects agent's mailbox if agent Agent sent message Msg to this agent.
  - Note difference in send and sent above!
  - If there is .sent! we talk about goals, without it we talk about beliefs.

### Goal agent architecture

- The basis of communication in GOAL is a simple mailbox semantics.
- There can be a delay in receiving a message.



## Send action

- Suppose we have 2 agents: fridge, groceryplanner.
  - In the fridge believes that no milk is left, it should inform the groceryplanner.

```
if bel(amountLeft(milk, 0)) then (groceryplanner).send(amountLeft(milk, 0)).
```
  - If the groceryplanner is informed about it, it should adopt the goal of buying milk:

```
if (fridge).send(amountLeft(milk, 0)) then adopt(buy(milk)).
```
- We can use variables within a message:

```
if bel(amountLeft(P, N), N < 2)  
  then (groceryplanner).send(amountLeft(P, N)).
```
- We can also use variables as the Sender and Receivers of a message.

```
forall (X).send(fact) do (X).send(not(fact)).
```

  - Just an example
- All variables in a (send) action must be closed.

## Agent selector

- Prefix of a send action is called an agent selector, what if you don't know all the agent names in the system?
- There are different agent selectors which can be used:
  - An agent name: (agentname).send(...)
  - A variable (to be instantiated): (Person).send(...)
  - Multiple agent names or variables:  
(agent1,agent2,Agt),send(...)
  - A quantor:
    - self.send(...): message sent to agent itself
    - allother.send(...): message sent to all other agents
      - For this and the one after this, we use an if...then rule to avoid flooding other agents rather than forall.do.
    - all.send(...): message sent to all agents

## Errors in selectors

- IF an agent tries to send a message to an agent which does not exist (anymore), a warning will be generated.
- The agent will be terminated if not all agent variables are instantiated at the time of sending (same for any action).

## Channels

- It is also possible to message by subject through channels.
  - Action: subscribe(Name), where Name is a simple atom.
  - Also available: unsubscribe(Name)
  - Allows using (Name).send(...)
    - Sends the message to all agents subscribed to the channel name; the sender does not need to be subscribed to the channel itself.
    - Message will still be received as (Agent).send(...) by the subscribers.
      - Still agent name.

### Moods of messages

- Goal supports 3 message types, called moods:
  - Indicative, typically used to inform.
    - Operator: ':' : send:(amountLeft(milk,0))
  - Declarative, typically used to indicate a goal.
    - Operator ! : sent!(status(door, closed)).
  - Interrogative, typically used to ask a question.
    - Operator ? : sent?(...)
- The indicative mood operator send: is optional as it is the default way of sending messages.
- The mood operator must also be used with sent.