

# Training Logical Mouses

A logical approach to agent design

R. Basili (Univ. Roma, Tor Vergata), a.a. 2023-24

# Overview

---

- Progettiamo un agente logico in Prolog
- Progettiamo la sua base di conoscenza, con l'intento di ottimizzarne la capacità di raggiungere l'obiettivo
- Osserviamo il comportamento autonomo dell'agente
- Ci porremo il problema: *può la conoscenza dell'agente essere appresa e non codificata manualmente?*
- Vedremo come collezionare esempi (positivi) del comportamento dell'agente (i.e. scelte della azione)
- Introdurremo un'ambiente per acquisire modelli di classificazione mediante Machine Learning: Weka
- Applicando Weka al caso del nostro agente, acquisiremo un modello decisionale (regole) attraverso l'apprendimento di una funzione di classificazione mediante Decision trees.
- Vedremo le analogie tra la base di conoscenza (Prolog) progettata top-down e il modello di azioni (albero delle decisioni) indotto dai dati dal DTL.

# Mouse And Cheese

---

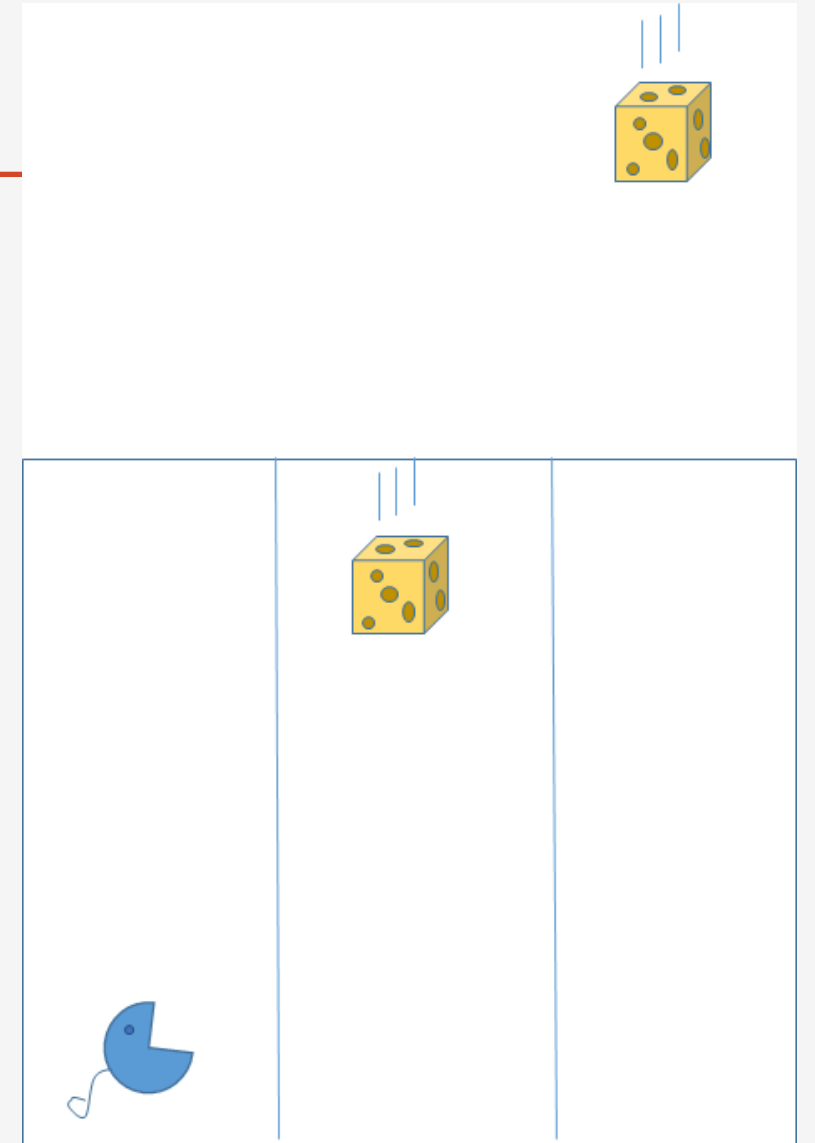
Nel mondo proposto, è necessario animare un piccolo topolino a raccogliere i pezzi di formaggio in caduta dall'alto in un ambiente 3x3 (vedi figura a lato).

Si progetti in Prolog il mondo del topolino (ambiente, sensori e mondi possibili) in modo da poter programmare le azioni del topolino che massimizzano l'utilità del goal:

*mangiare il numero massimo di pezzetti di formaggio.*

Definire la strategia di scelta delle azioni separatamente dalla applicazione delle azioni, in modo da favorire lo sviluppo di strategie diverse.

Si usi il predicato `random/1` per generare a caso le posizioni del topolino e del formaggio di volta in volta.



# Describing the state

---

The world corresponds to a 3x3 Map, where coordinates X (row) and Y (column) define an individual position.

There are two main objects in a map:

**Cheese Position:** predicate `cheese(X,Y)`

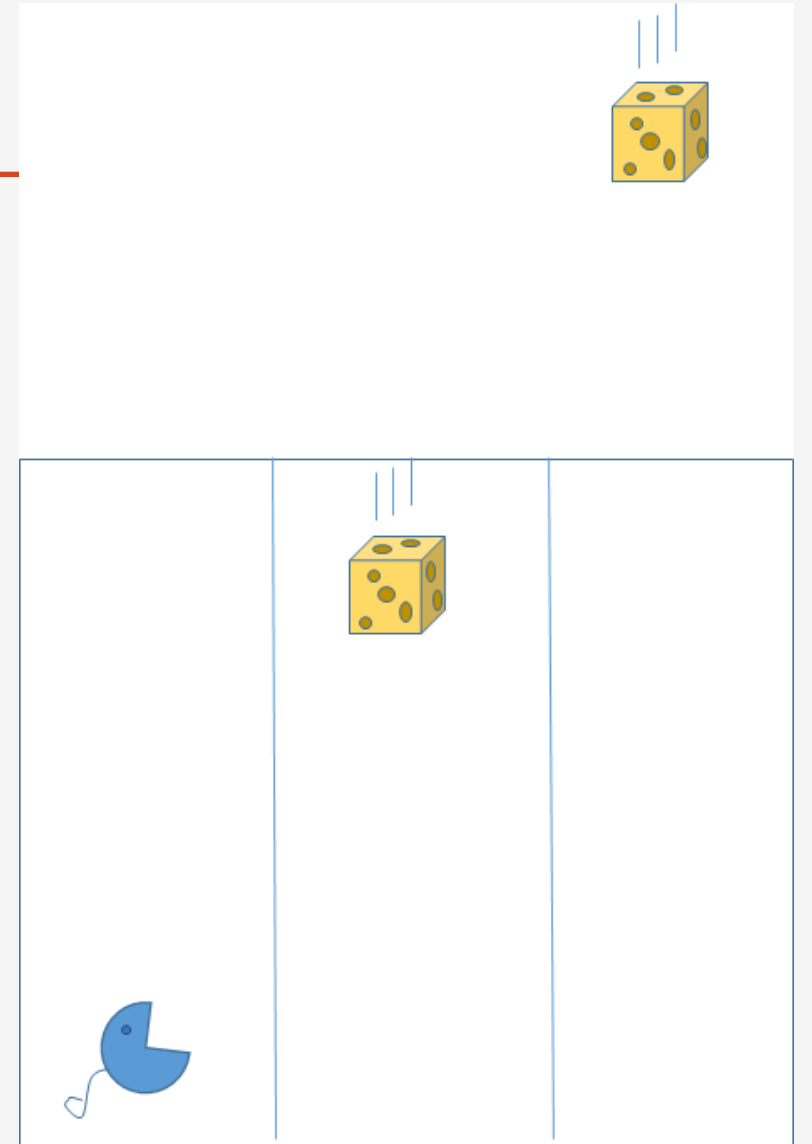
**Mouse Position:** predicate `mouse(X,Y)`

The State is thus defined as a pair (Prolog list)

`[cheese(XC,YC), mouse(XM, YM)]`

At every time stamp, the State list (i.e. the pair) represents the current state of the environment.

*"There is only one cheese at a time"* is implied by the fact that there is only ONE `cheese(X,Y)` predicate in the State declaration.



# Goal & Utility

**Goal:** The mouse stops when **there is no further cheese** that is falling down

As  $[cheese(XC, YC), mouse(XM, YM)]$

describes the State of the World, any value  $YC$  outside the  $[1,3]$  range is a valid indicator of the fact that there is no bit of cheese falling.

Default: When  $YC == 0$ , assume the Goal has been satisfied and the mouse stop acting.

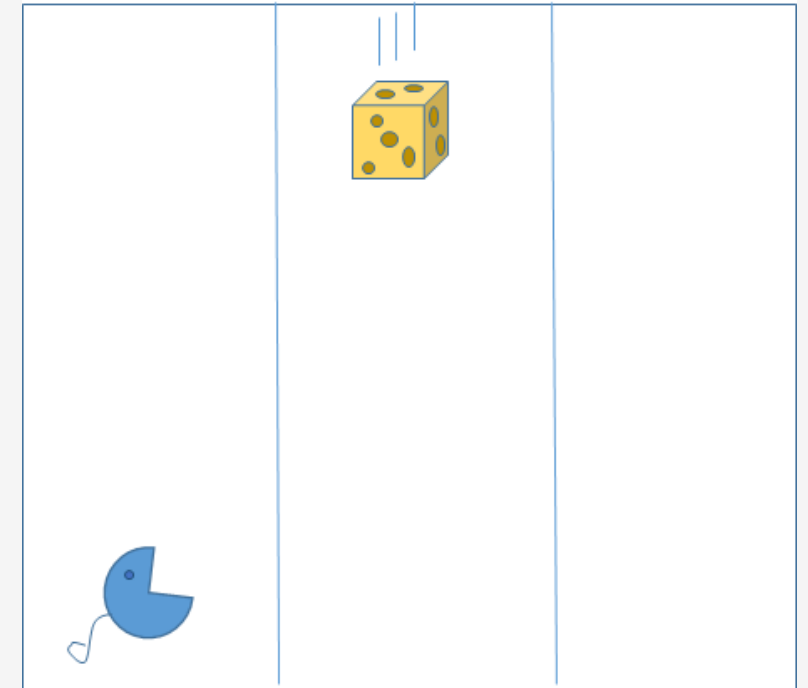
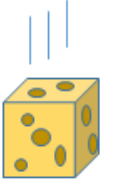
Moreover, as **the mouse does not jump**, at each time stamp  $XM == 1$ .

Eating is possible when the mouse is under the falling column of the cheese, and at the time the cheese touches the ground he is able to reach the same cell.

The **EATing** action is possible thus **ONLY WHEN** for some  $YC=YM > 0$ ,

$$XC == XM == 1.$$

**Utility:** The mouse has to maximize the bit of cheese he is able to eat when they are on the ground (i.e.  $XC=1$ ).

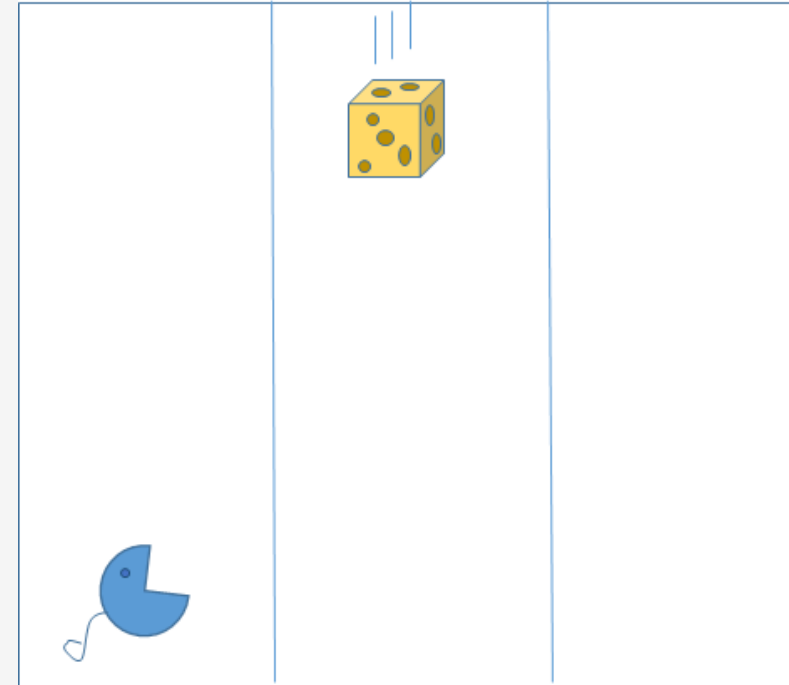


# The Mouse as an agent: requirements

---

## The following notions have to be specified:

- **Initialization** of the world (Use random/1)
- Synchronization of the world changes (i.e. the stepwise falling process of the cheese) with the mouse observations and actions
- The actions is selected by the mouse at each step in the attempt to maximize the utility
- **Goal-satisfaction check** at each step



# The Mouse as an agent: requirements



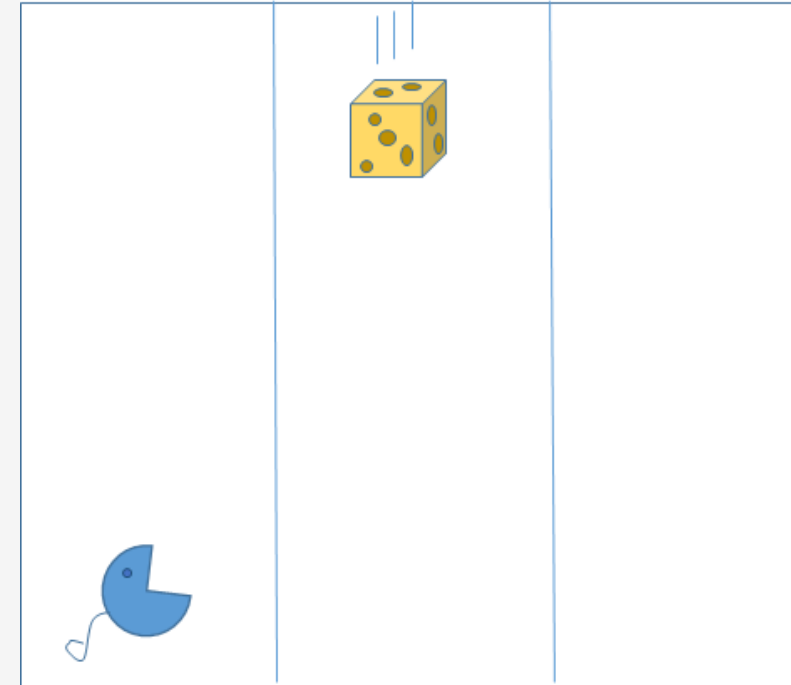
## Main problem instance cycle:

- Synchronize each requirements above in a step-by-step process. In Prolog:

```
run_problem_instance :-  
    init_the_world( YC, YM),  
    solve([cheese(3,YC), mouse(1,YM)], 0, Feed),  
    write("\n Task carried out: !!"),nl, write("The mouse has eaten "),  
    write(Feed), write(" cheese bits!!\n"), nl.
```

where:

```
init_the_world(CheeseNextColumn, MouseNextColumn) :-  
    random_pos_the_cheese(CheeseNextColumn),  
    random_pose_the_mouse(MouseNextColumn).
```



# Solving the task

## Two major actions:

- Check if the goal has been reached (no more cheese)
- OBSERVE/DECIDE/ACT cycle

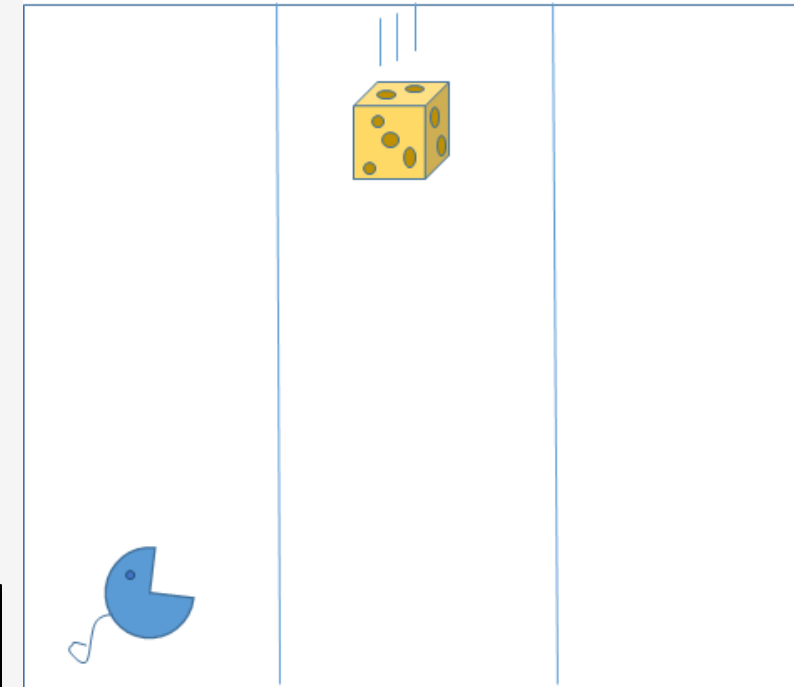
## Main solve cycle:

- In Prolog:

```
solve([cheese(XC,YC), mouse(_,_)], CurrFeed, CurrFeed) :-  
    check_goal_satisfaction(cheese(XC,YC)),  
    !.  
solve([cheese(XC,YC), mouse(XM, YM)], CurrFeed, AllFeeds) :-  
    decide_rule([cheese(XC,YC), mouse(XM, YM)], CurrFeed, Rule),  
    apply_action(Rule, [cheese(XC,YC), mouse(XM, YM)], CurrFeed, NewState, NewFeed),  
    env_step(NewState, NextState),  
    !,  
    solve(NextState, NewFeed, AllFeeds).
```

where:

```
env_step([cheese(1, _), mouse(XM, YM)], [cheese(3, NewYC), mouse(XM, YM)]) :-  
    /* generate a new starting position at column NewYC */  
    random_pos_the_cheese(NewYC).  
env_step([cheese(XC, YC), mouse(XM, YM)], [cheese(NewXC, YC), mouse(XM, YM)]) :- /* falling down ... */  
    XC > 1, NewXC is XC-1.
```





# Decision Rules for acting

---

```
decide_rule( [cheese(1, Y), mouse(1,Y)], _, eat) :-  
    !.  
decide_rule( [cheese(_anyXC, Y), mouse(_anyXM, Y)], _, stand) :-  
    \+(_anyXC == _anyXM),  
    !.  
decide_rule( [cheese(_, NewYC), mouse(_, YM)], _, moveright) :-  
    NewYC > YM,  
    !.  
decide_rule( [cheese(_, NewYC), mouse(_, YM)], _, moveleft) :-  
    NewYC < YM,  
    !.  
decide_rule( _anyState, _anyFeed, _, stand).
```

# Run the program

---

The program is accessible in file: Mouse.pl

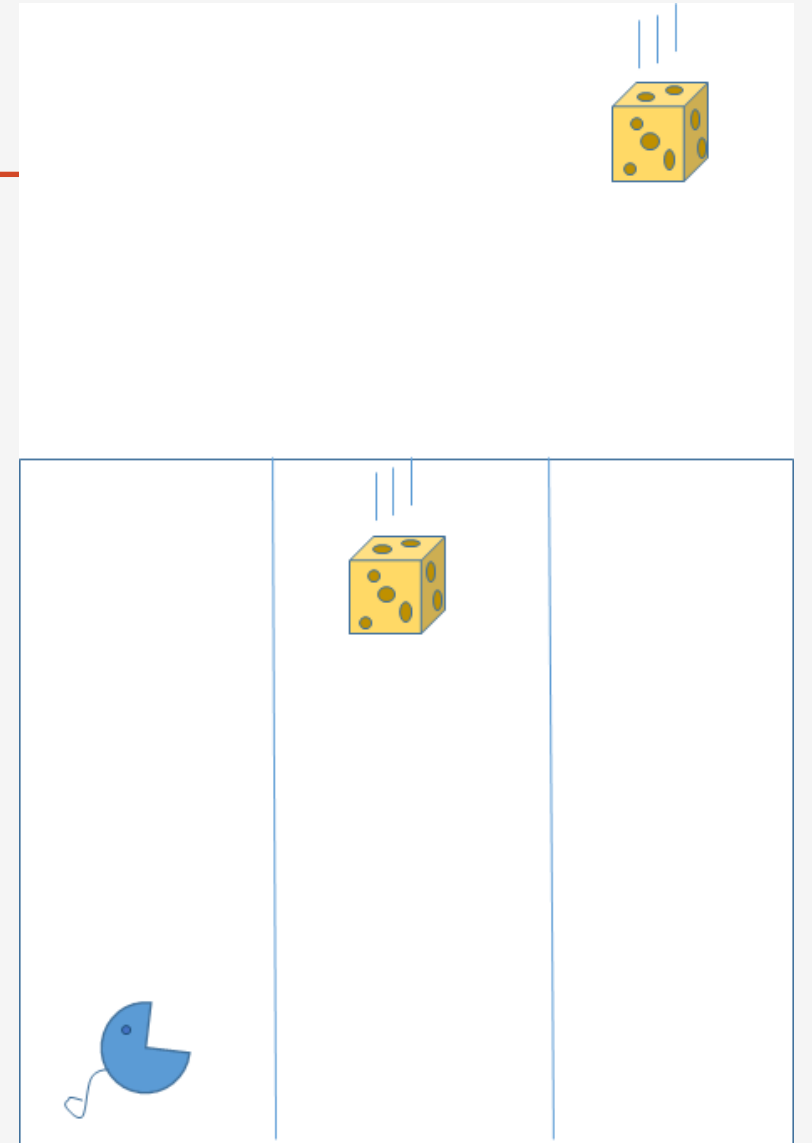
Use the predicate

*?- run\_problem\_instance.*

to run the agent.

The program includes:

- Messages to inspect the Mouse behaviour
- The (hand coded) decision rules that are optimal
- Logging of examples (see Machine Learning approach next slides)



# Training (Logical) Mouses

Adopting Machine Learning to design logical decisions

R. Basili (Univ. Roma Tor Vergata), a.a. 2019-20

# How can we train the Mouse to the proper decisions?

---

Let's for a while imagine that we got no decision rule, and that the system is blind, i.e. forced to decide randomly. e.g.

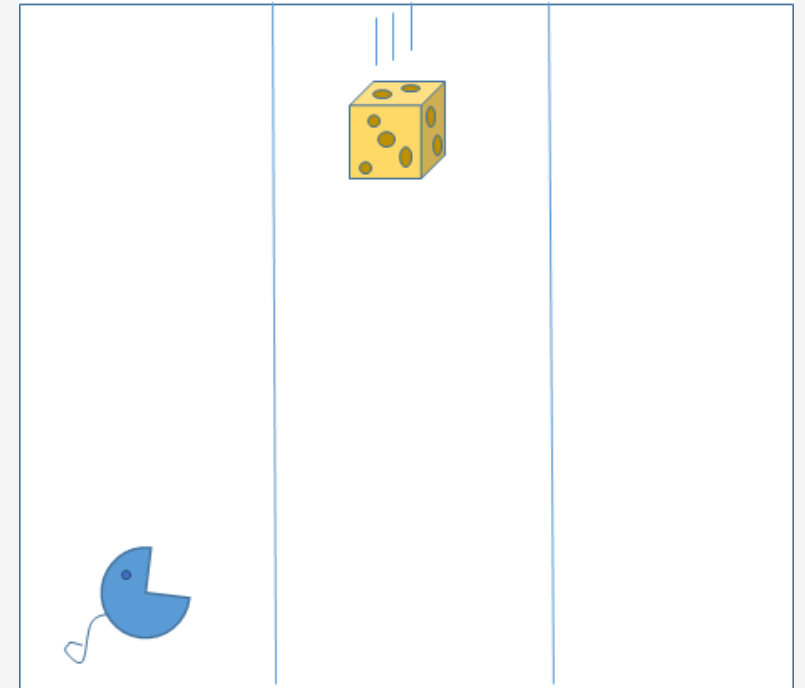
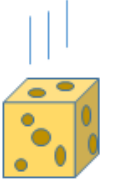
```
/* RANDOM, i.e. Non intelligent CHOICE */  
decide_rule(_anyState, _AnyFeed, Action) :-  
    N is random(4)+1,  
    select_ith(N, [eat,left,moveleft,moveright], Action).
```

Can we do better by Machine Learning the rules?

Where to start?

IDEA: Let's start running the *Mouse.pl* program and collect its behavior (i.e. actions taken when the mouse observe States) and try **to learn the Prolog code for decision rules**.

What we need: collecting examples and a learning algorithm



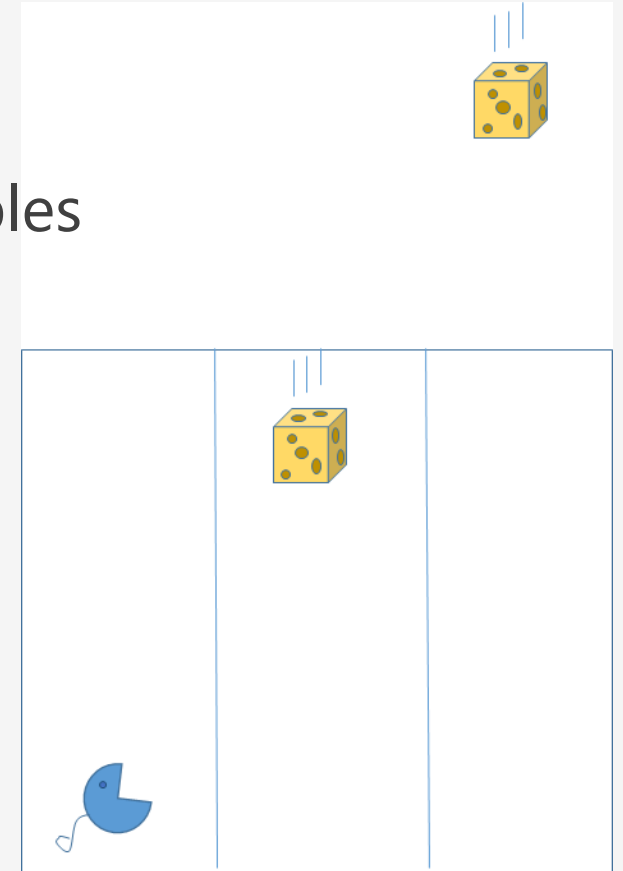
# Training the Mouse: collecting examples

---

Few lines of Prolog code (see file *Mouse\_to\_Learn.pl*) allow us to store the examples of State-Action choices.

After having decided to act the Prolog agent can store the choice (through dynamic predicates) by memorizing as examples every chosen State-Action pair.

```
storerules([cheese(XC,YC), mouse(XM, YM)], CurrFeed, Rule) :-  
    example(N, _, _, _), /* <== a dynamic predicate */  
    NewN is N+1,  
    asserta(example(NewN, [cheese(XC,YC), mouse(XM, YM)], CurrFeed, Rule)),  
    !.  
storerules([cheese(XC,YC), mouse(XM, YM)], CurrFeed, Rule) :-  
    asserta(example(1, [cheese(XC,YC), mouse(XM, YM)], CurrFeed, Rule)),  
    !.
```



# Training the Mouse: collecting examples

Can we make the *example/4* predicate facts available in a machine learning platform, such as Weka?

We need to store every example in terms of **features** (i.e. the observable properties of each recorded step) and associate to them **the chosen action** (as the target classification choice).

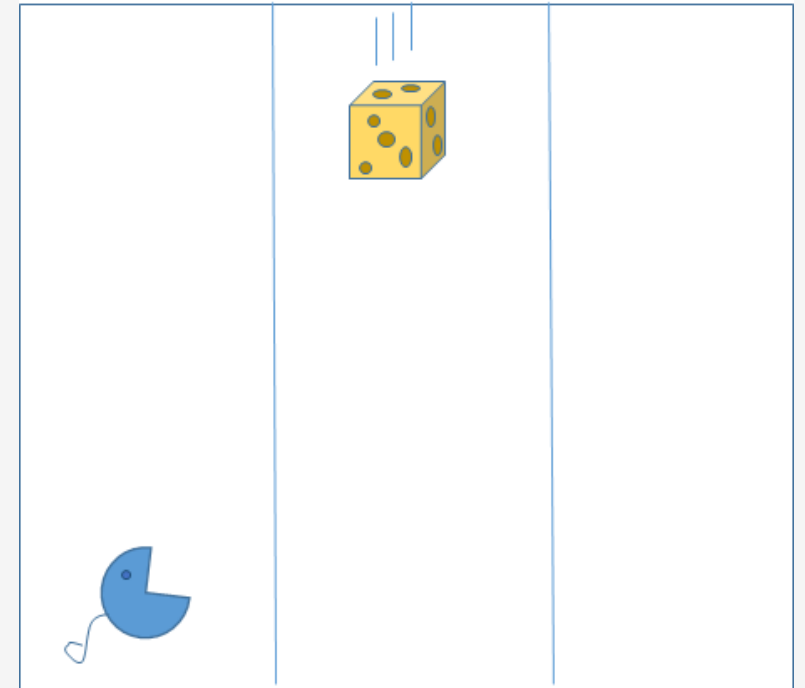
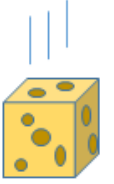
**Features** are **states of the world**, i.e. positions of cheese bits and mice.

The **action** is the **target decision**

The learning is devoted to induce a classification function from **states** to **actions**.

In order to write on file all the meaningful attributes of the examples:

```
?- tell('Esempi2.data'),  
    example(_, [cheese(XC,YC), mouse(XM, YM)], CurrFeed, Rule),  
    write(XC),write(','),write(YC), write(','),write(XM),write(','),write(YM),write(','),  
    write(CurrFeed), write(','),write(Rule),nl,  
    fail.  
?- told.
```



# Training the Mouse: collecting examples

```
?- tell('Esempi2.data'),
    example(_, [cheese(XC,YC), mouse(XM, YM)], CurrFeed, Rule),
    write(XC),write(','),write(YC), write(','),write(XM),write(','),write(YM),write(','),
    write(CurrFeed), write(','),write(Rule),nl,
    fail.
?- told.
```

This is the outcome in an ARFF file (Mouse2.arff) ... from the header:

```
@RELATION Mouse
```

```
@ATTRIBUTE cheesePosX      INTEGER
@ATTRIBUTE cheesePosY      INTEGER
@ATTRIBUTE mousePosX        INTEGER
@ATTRIBUTE mousePosY        INTEGER
@ATTRIBUTE eatenBits        INTEGER
@ATTRIBUTE class            {eat,movelf,moveleft,moveright,stand}
```

```
@DATA
```

```
1,2,1,2,13,eat
```

```
...
```

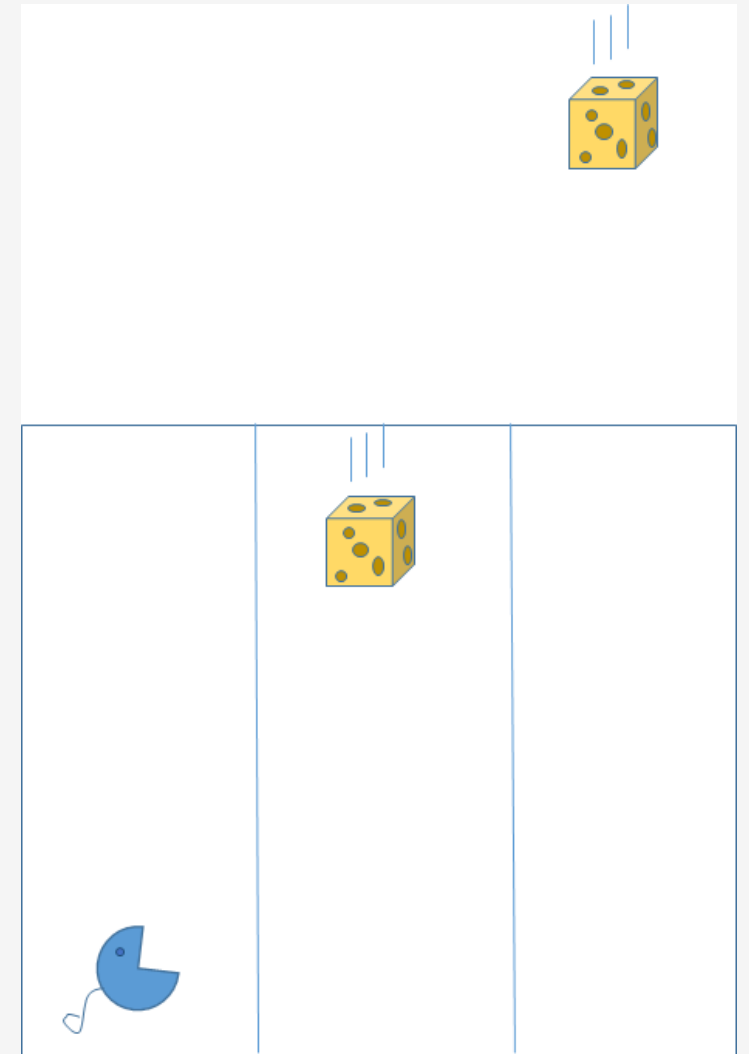
```
1,2,1,2,3,eat
```

```
2,2,1,2,3,stand
```

```
3,2,1,1,3,moveright
```

```
2,1,1,2,2,moveleft
```

```
...
```



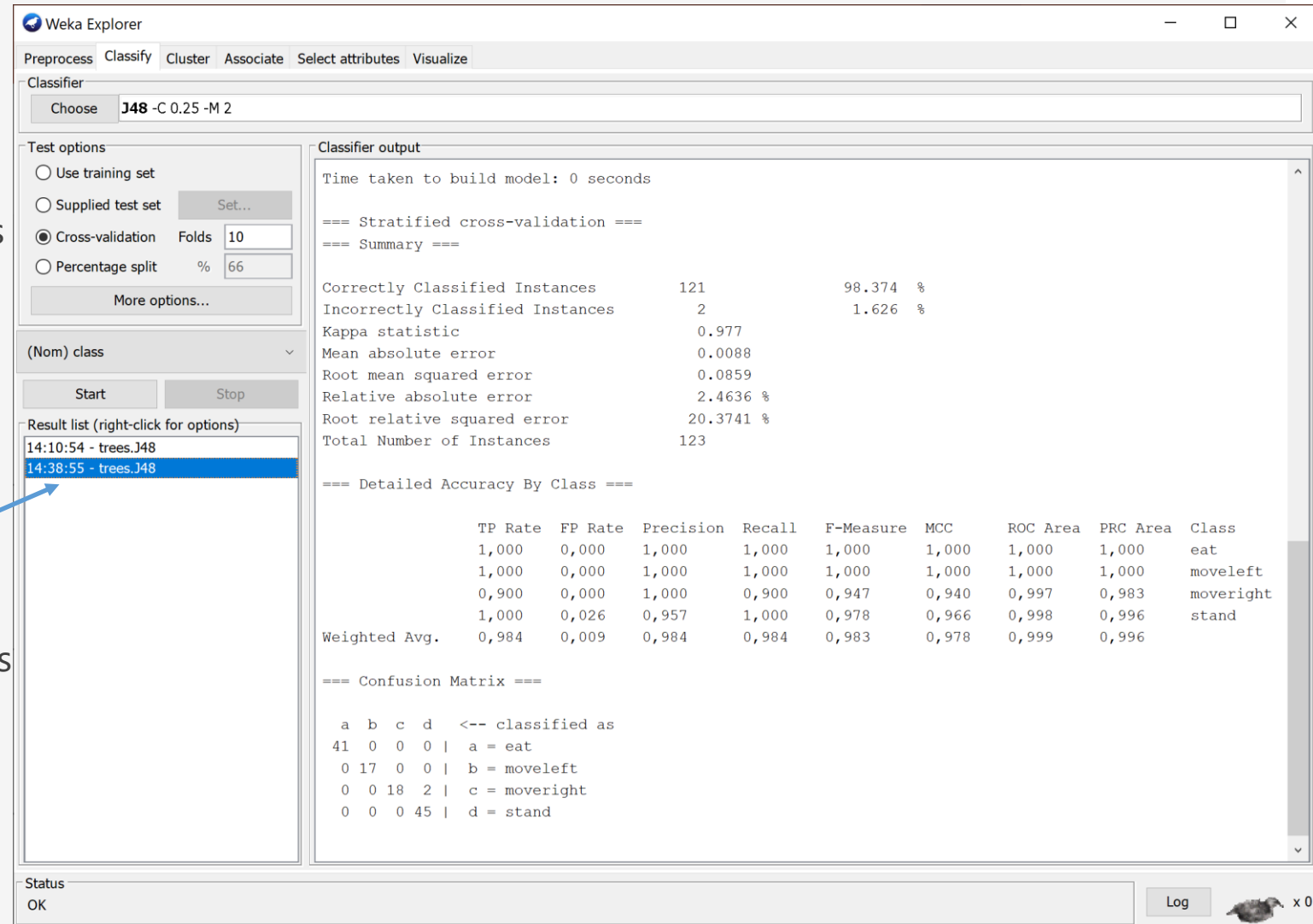
# Learning a decision tree

With Weka we can apply the J48 algorithm to induce the corresponding decision tree

The outcome is a model whose performance is 98.37% as shown on the left

In the 2 errors the system predicted action *stand* in the case it needed to *moveright* (i.e. a possible error in the training data)

In Weka, the model, i.e. the decision tree in this case, can be inspected by clicking (with the right side) on the Result list elements



The screenshot shows the Weka Explorer interface. The Classifier is set to J48 -C 0.25 -M 2. The Test options are set to Cross-validation with 10 Folds. The Classifier output shows a performance of 98.374% for correctly classified instances. The Result list shows two entries: 14:10:54 - trees.J48 and 14:38:55 - trees.J48. A blue arrow points to the second entry. The Status bar shows OK.

**Classifier output**

```
Time taken to build model: 0 seconds
=== Stratified cross-validation ===
=== Summary ===
Correctly Classified Instances      121      98.374 %
Incorrectly Classified Instances     2       1.626 %
Kappa statistic                    0.977
Mean absolute error                 0.0088
Root mean squared error             0.0859
Relative absolute error             2.4636 %
Root relative squared error        20.3741 %
Total Number of Instances          123

=== Detailed Accuracy By Class ===
```

|               | TP Rate | FP Rate | Precision | Recall | F-Measure | MCC   | ROC Area | PRC Area | Class     |
|---------------|---------|---------|-----------|--------|-----------|-------|----------|----------|-----------|
|               | 1,000   | 0,000   | 1,000     | 1,000  | 1,000     | 1,000 | 1,000    | 1,000    | eat       |
|               | 1,000   | 0,000   | 1,000     | 1,000  | 1,000     | 1,000 | 1,000    | 1,000    | moveleft  |
|               | 0,900   | 0,000   | 1,000     | 0,900  | 0,947     | 0,940 | 0,997    | 0,983    | moveright |
|               | 1,000   | 0,026   | 0,957     | 1,000  | 0,978     | 0,966 | 0,998    | 0,996    | stand     |
| Weighted Avg. | 0,984   | 0,009   | 0,984     | 0,984  | 0,983     | 0,978 | 0,999    | 0,996    |           |

```
=== Confusion Matrix ===
 a  b  c  d  <-- classified as
41  0  0  0 | a = eat
 0 17  0  0 | b = moveleft
 0  0 18  2 | c = moveright
 0  0  0 45 | d = stand
```



# The acquired decision tree

**The decision tree** acquired in the right hand side of the slide **is very close to the Prolog code for the *decide\_rule/4* predicate.**

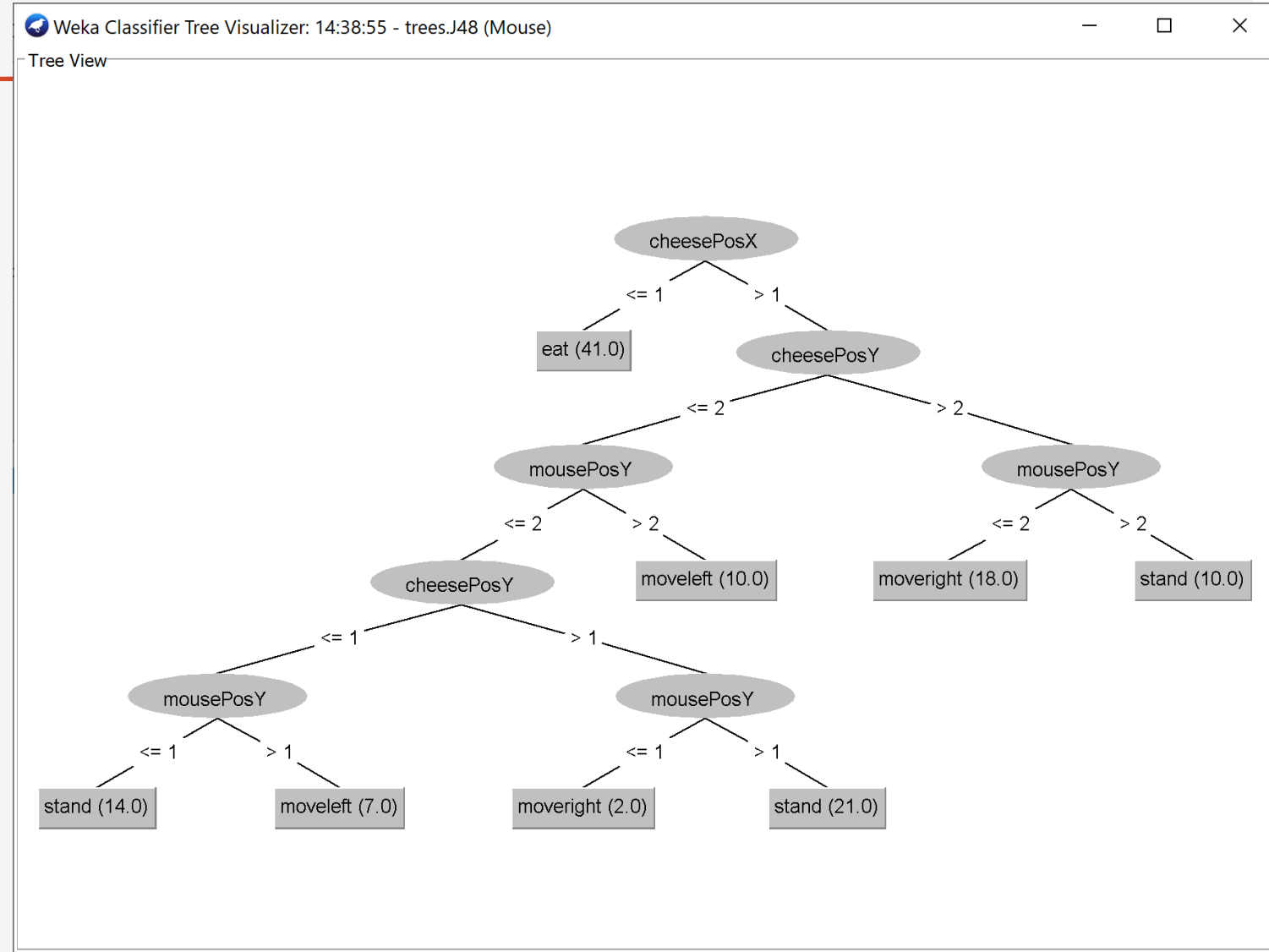
Surprised?

The variables used to decide the action class by the tree are:

|                       |         |
|-----------------------|---------|
| @ATTRIBUTE cheesePosX | INTEGER |
| @ATTRIBUTE cheesePosY | INTEGER |
| @ATTRIBUTE mousePosX  | INTEGER |
| @ATTRIBUTE mousePosY  | INTEGER |

These do not include the ***eatenBits* attribute that is in fact irrelevant to the decision**: this has been rediscovered by the DT learning algorithm

As in the Prolog code, only the relative positions of the cheese and the mouse (i.e. mousePosX, mousePosY, cheesePosX, cheesePosY) are influential to the decisions.

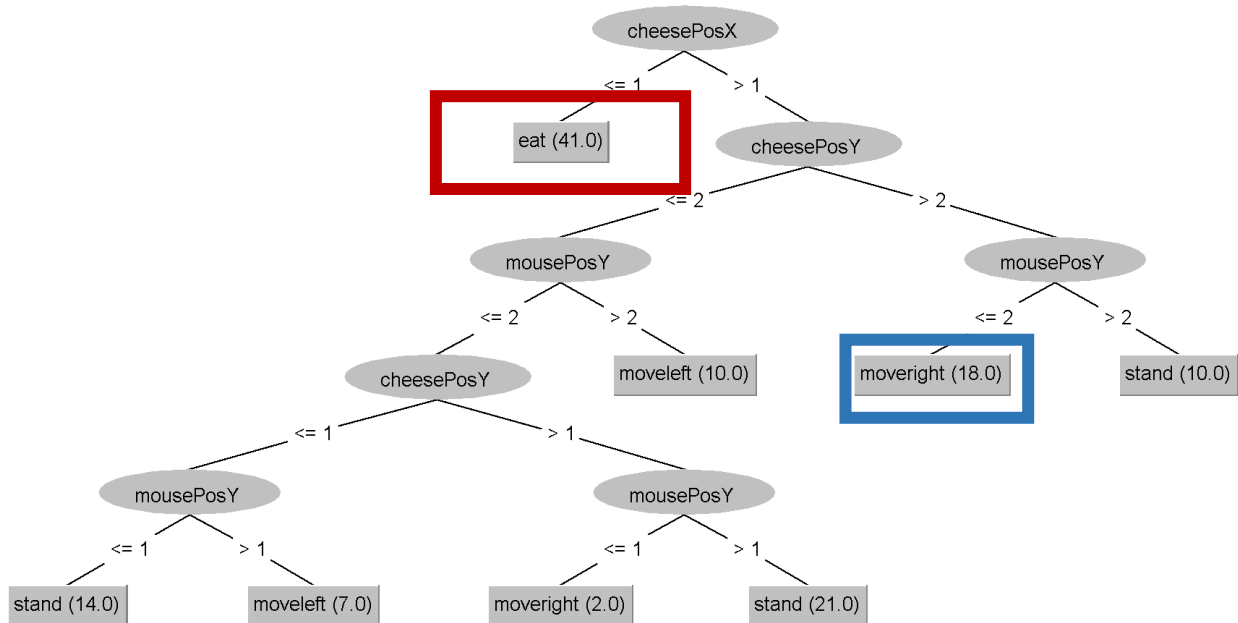


# Examples of Prolog rules induced by the DT algorithm

```
weka_decide_rule([cheese(CheesePosX, _), mouse(_, _)], _, eat) :-  
    CheesePosX = < 1,  
    !.
```

Weka Classifier Tree Visualizer: 14:38:55 - trees.J48 (Mouse)

Tree View



```
weka_decide_rule([cheese(CheesePosX, CheesePosY),  
    mouse(_, MousePosY)], _, moveright) :-  
    CheesePosX > 1,  
    CheesePosY > 2,  
    MousePosY = < 2,  
    !.
```

# The acquired decision tree

Obviously the learning machine has been exposed to the perfect behaviour of the hand-coded Prolog program and it has been easy to rebuild such a simple model.

In case *no such knowledge is available* the question is: **can we train the system? Where are the examples? We could use the estimated quality of individual moves (i.e. an empirical measure of their advantage), and then train the system to select the optimal move (e.g. a regression task)?** Randomly selected actions could be used to gather training data.

Modify the provided Prolog code to formulate a quality estimate function and to train the random agent mouse from scratch.

