Guião Teórico-Prático

MRSI - Sistemas Inteligentes II - 2024/2025

©Luís Seabra Lopes

Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

I Knowledge representation

1. The slide "Description logic examples (IV): Let's decode these axioms"contains several axioms. Translate them to first-order logic. Make sure you understand the meaning of each axiom.
2. Suppose a "beach resort" is a resort near a beach that the resort guests can use. The beach has to be near the sea or a lake, where swimming is permitted. A resort must have places to sleep and places to eat. Write a definition of beach resort in OWL.
3. A luxury hotel has multiple rooms to rent, each of which is comfortable and has a view. The hotel must also have more than one restaurant. There must be menu items for vegetarians and for meat eaters to eat in the restaurants. Define a luxury hotel in OWL, based on this description. Make reasonable assumptions where the specification is ambiguous.

II Prolog-based exercices

1. Practice Prolog by implementing the following list processing procedures:
   (a) Check if an element occurs in a list. (equivalent to pre-defined member/2)
   (b) List concatenation: given two lists, produce a new list containing all elements from the first list followed by all elements of the second list. (equivalent to pre-defined append/3)
   (c) List inverstion: given a list, produced a new list with the same elements in inverted sequence. (equivalent to reverse/2)
   (d) Counting: given an element and a list, count the number of occurrences of the element in the list. First implement without a cut. Then reimplement with a cut.
   (e) Minimum of a list of numbers: given a list of numbers, find the smallest element. It fails if the list is empty.
   (f) Minimum of a list: similar to the previous exercice, but elements can be of any type: in addition to a list, it receives the name of predicate to be used for comparison between elements.

(g) Merge: given two sorted lists of numbers, return a new sorted list containing all elements from the input lists.

(h) Set of all subsets: given a set of elements (in the form of a list), produce a list of lists representing all subsets of the input set.

(i) Given two lists, produce of list of the pairs formed by elements in equal positions in the input lists. The predicate fails if the input lists have different lengths.

If you need more practice, note: Many exercices from the Python guide (AI course) can be reimplemented in Prolog.

2. Consider the problem of controlling the behavior of a lift. Through perception, the lift can evaluate the following conditions:

   - `passengerFor(X)` - a passenger inside the lift is going to the current floor, or to a floor above, or to a floor below. ($X \in \{here, above, below\}$).
   - `callFor(X)` - a passenger outside the lift called the lift to the current floor, or to a floor above, or to a floor below. ($X \in \{here, above, below\}$).
   - `door(State)` - the door of the lift is in one of three states: closed; open and waiting for passengers to enter/leave the lift; open, but is time to close again. ($State \in \{open, timetoclose, closed\}$)

   The lift can perform the following actions:

   - `up` - go up one floor
   - `down` - go down one floor
   - `open` - open the door
   - `close` - close the door
   - `wait` - wait for people to enter / leave.

   (a) Develop a set of condition-action rules to control the lift. To code the rules, use the following operators:

   ```
   :- op(800,fx,when). % unary operator
   :- op(700,xfx,do). % non associative operator
   :- op(300,xfy,or). % righ-associative operator
   :- op(200,xfy,and).
   :- op(150,fx,not).
   ```

   The general format of each rule is the following:

   ```
   when Condition1 LogicalOperator Condition2 .... do Action
   ```

   (b) Develop a procedure `choose_action(+State,-Action)` that, given a list of conditions representing the current state information, chooses an applicable action.

3. Consider the following set of facts and rules. They identify possible causes of leaks in kitchens and toilets.

```
:- op(800,fx,if). % unary operator
:- op(700,xfx,then). % non associative operator
:- op(300,xfy,or). % righ-associative operator
:- op(200,xfy,and). % lower precedence == higher priority

fact(hall_wet).
fact(toilet_dry).
fact(window_closed).

if hall_wet and kitchen_dry then leak_in_toilet.
if hall_wet and toilet_dry then kitchen_problem.
if window_closed or no_rain then no_water_entered.
if kitchen_problem and no_water_entered then leak_in_kitchen.
```

The previous implementation of forward chaining uses the Prolog database to keep track of what has been inferred so far. Reimplement forward chaining in such a way that the Prolog database is not used, and no prints are needed. For that purpose, you should implement a procedure `forward(-L)` that produces a list `L` with all predicates that can be inferred from the known facts and rules.

Result:

```
?- forward(L).
L = [ leak_in_kitchen, no_water_entered, kitchen_problem,
      window_closed, toilet_dry, hall_wet].
```

4. Consider the following depth-bounded backward chaining procedure. It uses the same operator definitions as before, and it is largely similar to the backward chaining procedure given in the slides, However, there is a depth limitation: `bprove(G,D)` is true if `G` can be proved with a proof tree of depth less than or equal to `D`.

```
bprove(F,D) :- fact(F).
bprove((A and B),D) :- bprove(A,D), bprove(B,D).
bprove((A or B),D) :- bprove(A,D); bprove(B,D).
bprove(G,D) :- if C then G, D > 0, D1 is D-1, bprove(C,D1).
```

Modify this procedure (also in the attached "chaining.pl"program) such that the bound is on number of facts that appear in the proof. Why might this be better or worse than using the depth of the tree?

5. Consider the Python implementation of a traveller agent, with its perceive-decide-act loop, in the attached module `"pyswip5.py"`.

The `decide(current,end)` method is a wrapper for a Prolog procedure, `decide(+CurrentSpot,+GoalSpot,-Plan)`, that will determine a plan to reach `GoalSpot` from `CurrentSpot`.

Implement in Prolog the `decide/3` procedure following an A*-like strategy. To compute/estimate travelling costs between spots, use the foreign procedure `current_cost(+Spot1,+Spot2,-Cost)`. Use this procedure both for adjacent and non-adjacent spots.

For testing, use the following topology (also in the attached "topology.pl"):

```
% file: topology.pl
connection(a,b).
connection(a,c).
connection(c,d).
```

```
connection(c,e).
connection(c,g).
connection(b,e).
connection(e,f).
connection(e,g).
connection(d,g).
connection(f,h).
connection(g,h).

connected(X,Y) :- connection(X,Y).
connected(X,Y) :- connection(Y,X).
```

Initialize and run as follows:

```
>>> from pyswip5 import TravellerAgent
>>> traveller = TravellerAgent( { 'a':(3,8), 'b':(1,6), 'c':(6,6),
                                  'd':(8,7), 'e':(4,5), 'f':(2,2),
                                  'g':(6,3), 'h':(3,1) }, "topology.pl" )
>>> traveller.mainLoop('a','h')
....
....
```

6. Consider the following semantic network coded in Prolog:

```
declaration(descartes, subtype(mamifero,vertebrado)).
declaration(darwin, mamar(mamifero,sim)).
declaration(descartes, altura(mamifero,1.2)).
declaration(darwin, subtype(homem,mamifero)).
declaration(darwin, gosta(homem,carne)).
declaration(descartes, altura(homem,1.75)).
declaration(damasio, gosta(filosofo,filosofia)).
declaration(descartes, member(socrates,homem)).
declaration(damasio, member(socrates,filosofo)).
declaration(descartes, professor(socrates,matematica)).
declaration(descartes, professor(socrates,filosofia)).
declaration(descartes, peso(socrates,80)).
declaration(descartes, member(platao,homem)).
declaration(descartes, professor(platao,filosofia)).
declaration(descartes, member(aristoteles,homem)).
```

   (a) Implement a procedure `query(+PredName,+E1,-E2,-User)` [1] that, given the name of a relation (predicate) and an entity, finds the value of that relation for that entity, possibly using inheritance, as well as the user that declared it. [solved in slides]

   Examples:

```
?- query(professor,socrates,Subject,WhoTold).
Subject = matematica,
WhoTold = descartes ;
Subject = filosofia,
```

[1] By tradition, in arguments of Prolog procedures or predicates, '+' indicates input argument and '-' indicates output argument. It is very often the case that an argument can be both input and output in different calls of the same procedure. This is sometimes signaled as '?'. Note that '+', '-' and '?' are only used in documentation, not in actual code. See some examples here: https://stackoverflow.com/questions/16002546/input-output-parameters-in-prolog-definition.

```
WhoTold = descartes ;
false.

?- query(gosta,socrates,X,WhoTold).
X = carne,
WhoTold = darwin ;
X = filosofia,
WhoTold = damasio ;
false.

?- query(altura,socrates,Height,WhoTold).
Height = 1.75,
WhoTold = descartes ;
Height = 1.2,
WhoTold = descartes ;
false.

?- query(mamar,socrates,X,WhoTold).
X = sim,
WhoTold = darwin ;
false.
```

(b) Implement a `query_cancel/4` [2] procedure, similar to the previous one, but with inheritance cancelling.

Example:

```
?- query_cancel(altura,socrates,Height,WhoTold).
Height = 1.75,
WhoTold = descartes ;
false.
```

(c) Implement a `query_path(+PredName,+E1,-E2,-Path)` procedure, similar to the previous ones, with inheritance cancelling, and determining the path from `E1` to the entity from which the relation is inherited.

Example:

```
?- query_path(mamar,socrates,X,Path).
X = sim,
Path = [socrates, homem, mamifero] ;
false.

?- query_path(gosta,socrates,X,Path).
X = carne,
Path = [socrates, homem] ;
X = filosofia,
Path = [socrates, filosofo] ;
false.
```

7. The STRIPS-based planner developed in Prolog and reproduced in the theoretical slides uses breadth-first search without avoiding repeated states. Improve the procedure

[2] We can also refer to Prolog procedures in the form ≪predicate name≫/≪arity≫, where arity is the number of arguments.

`generateTransition(+ID,+State,-ChildID)` to ensure that no nodes are created for states that already exist in other nodes.

Test the two versions of the planner (with and without repeatitions) in the problem defined by the following facts:

```
initial_state( [ floor(a), floor(b), floor(d), on(c,b),
                 free(a), free(c), free(d), robot_free ] ).
goal( [ on(a,c),on(c,b),on(b,d) ] ).
```

8. Consider the following DCG grammar for a small subset of the English language:

```
sentence --> np, vp.

np --> properNoun.
np --> article, commonNoun.

vp --> verb, np.
vp --> verb, adjective.

commonNoun --> [breeze].
commonNoun --> [city].
commonNoun --> [man].
commonNoun --> [men].
commonNoun --> [cat].
commonNoun --> [cats].

properNoun --> [peter].
properNoun --> [aveiro].

verb --> [is].
verb --> [are].

adjective --> [smelly].
adjective --> [beautiful].

article --> [the].
article --> [a].
article --> [an].
```

This grammar recognizes sentences like: "*the breeze is smelly*", "*aveiro is beautiful*", "*aveiro is a city*", "*peter is a man*", "*the cats are beautiful*". However, it also recognizes many sentences that are not grammatically correct.

The following exercices are designed to guide the student in gradually extending and improving the grammar. Each exercice introduces some new features. The required grammar rules are given, but you need to develop suitable augmentations, i.e. extra arguments for agreement, verb subcategorization and synthatic tree.

For convenience, a Prolog program with all grammar rules and relevant example sentences (`"grammar.pl"`) is provided with this description.

The new features targeted in one exercice typically imply changes in the rest of the grammar. If you prefer to solve each exercice separately, you can consider developing separate grammars (in separete files) featuring only the rules that are relevant to cover the respective example sentences.

(a) Augment the grammar to ensure agreement in number (for example "*the cat is ...*"is correct, but "*peter are ...*"...] is not).

(b) Augment the grammar so that it works not only for recognition but also for syntactic analysis, that is, the grammar must produce a syntactic tree as result.

(c) Extend the grammar to recognize sentences like: "*peter is in aveiro*". The following additional rules are necessary:

```
vp --> verb, pp.

pp --> preposition, np.
```

```
preposition --> [in].
```

Augment them as before.

(d) Extend the grammar to recognize pronouns in sentences like these: "*he is in aveiro*"and "*she loves him*". The following additional rules are required.

```
np --> pronoun.

pronoun --> [i].  % subject
pronoun --> [you].
pronoun --> [he].
pronoun --> [she].
pronoun --> [we].
pronoun --> [they].

pronoun --> [me]. % object
pronoun --> [you].
pronoun --> [him].
pronoun --> [her].
pronoun --> [them].
pronoun --> [us].

verb --> [love].
verb --> [loves].
```

Augment them as before. Where appropriate, add an extra argument to distinguish pronouns in subject role and pronouns in object role. Note that sentences "*her is in aveiro*"and "*me loves him*"are not correct. Also, where appropriate, augment the grammar to ensure agreement in person. (e.g., "*he is in aveiro*"is correct, but "*he are in aveiro*"is not.)

(e) Extend the grammar to recognize sentences like: "*you give the gold to me*"and "*you give me the gold*". The following additional rules are required:

```
vp --> verb, np, np.
vp --> verb, np, pp.

verb --> [give].
verb --> [gives].

commonNoun --> [gold].

preposition --> [to].
```

Augment them as before. Additionally, augment the grammar to perform verb subcategorization, preventing sentences like *"i am the gold to you*", "*you give*", etc., to be recognized.

(f) Extend the grammar to recognize sentences like "*they smell the wumpus*", *"the wumpus smells awful*"and "*peter smells like a wumpus*". The following additional rules are required:

```
verb --> [smell].
verb --> [smells].

adjective --> [awful].

preposition --> [like].

commonNoun --> [wumpus].
```

Augment them as before.

(g) Extend the grammar to recognize sentences like "*the cat is here*"and "*peter is there*". The following additional rules are required:

```
vp --> verb, adverbe.

adverbe --> [here].
adverbe --> [there].
```

Augment them as before.

(h) Extend the grammar to recognize sentences like *"the beautiful cat died*". The following additional rules are required:

```
vp --> verb.

np --> article, adjective, commonNoun.
```

Augment them as before.

9. It is intended that the VG-10 robot responds to commands specified in natural language, such as the following:

   - "Go to the Oval Crater station"→ move(oval_crater)
   - "Go to Deep Crater"→ move(deep_crater)
   - "Load a barrel at the Lunar Valley station"→ load(lunar_valley,barrel)
   - "Load a food package at Oval Crater"→ load(oval_crater,food)

   Using Prolog's Definite Clause Grammar (DCG) formalism, specify a grammar that allows recognizing commands of this type, returning its semantics.

10. The semantic dialog code presented in the theoretical class (also in the attached 'dialog.pl' program) can process the sentences like the following:

    - "The professor is in the workshop."
    - "The bicycle is in the workshop."
    - "Where is the professor."
    - "What is in the workshop."

    Extend the grammar and adapt the program where needed, in order to process the following additional sentences:

    - "The name of the professor is einstein."
    - "What is the name of the professor."
    - "Where is einstein."
    - "The car of the professor is in the workshop."
    - "Where is the car of the professor."
    - "Where is the car of einstein."
    - "Where is the car."