Tutorium to Introduction to AI, 8th week - Nicolas Höning

Nicolas Höning

July 13, 2006

nl interface -> Prolog

- the task
- the toy task
- our toy task - using a DCG grammar

the homework task

- You are in principle expected to write a system that is driven by (a subset of) natural language.
- It accepts new information:
  process([the,mall,is,near,the,school.],[]).
  process([is,the,school,near,the,mall,?],[]).
  process([where,is,the,school?],A).

our "toy" task

To get you started, I thought we might develop a little toy natural language interface system with these properties:

- the database stores instances of an Object-Value predicate "xy(Object,Value)"
- there is only one statement-type: [Object,'is',Value,'.']
- and only one one question-type ['is',Object,Value,'?'] (with answers 'yes' or 'no')

our "toy" task

the database stores instances of an Object-Value predicate "xy(Object,Value)"

- ok, let's just imagine that data structure as a predicate like this (we could use any):
  xy(O,V).
- we can get this into the database "live", because Prolog is capable to add to its functions while working on its functions. We could say:
- assert(xy(O,V)).

our "toy" task

- *there is only one statement-type: [Object,'is',Value,'.']*
- ok, I would say that means we accept something like this: s([the, Object, is, Value, .])
- *and only one one question-type ['is',Object,Value,'?'] (with anwsers 'yes' or 'no')*
- I would say that means we accept something like this: s([is, the, Object, Value, ?])

our "toy" task

Now we have it all together. The first version is as short as this:

- s([the, Object, is, Value, .]) :- assert(xy(Object,Value)).
  s([is, the, Object, Value, ?]) :- xy(Object,Value).

our "toy" task

Here a few queries I posed and the responses:

-
  ```
  ?- s([the,dog,is,blue,.]).
  Yes
  ```

-
  ```
  ?- s([the,cat,is,red,.]).
  Yes
  ```

-
  ```
  ?- s([is,the,cat,blue,?]).
  No
  ```

-
  ```
  ?- s([is,the,cat,red,?]).
  Yes
  ```

-
  ```
  ?- listing(xy).
  :- dynamic xy/2.
  xy(dog, blue).
  xy(cat, red).
  ```

our "toy" task - using a DCG grammar

- Of course, that was really simple. We want to use more types of statements and questions.
- As we know a good way for formulating such stuff, DCGs, how can we use them?
- short answer... something like this:

```
s2(X) :-
  statement(X, []).
statement –> object, [is], value, [.].
statement –> [there], [is], [a], value, object, [.].
```

- in our program we call a DCG grammar rule with two extra arguments:
  the list to be checked - and an empty list (that's the difference list notation)

our "toy" task - using a DCG grammar

- But that only works for yes/no - answers. We need to check a sentence and process some ingridients of it (here: Object and Value).
- We can add parameters to the grammar rules such that we can ask for it like this:
-
```
s2(X) :-
  statement(Object, _, Value, _, X, []),
  assert(xy(Object,Value)).
s2(X) :-
  question(_, Object, Value, _ , X, []),
  xy(Object,Value).
```

our "toy" task - using a DCG grammar

- How does that work? There are a lot parameters...
- here are the rules (in this example I said object can be "a cat", "the dog" etc. just to get a little more complex...):
-
```
statement(O,[is],V,[.])
   -> object(O), [is], value(V), [.].
question([is],O,V,[?])
   -> [is], object(O), value(V), [?].
det -> [a].
det -> [the].
object(O) -> det, atomic(O).
atomic(X) -> [X].
value(X) -> atomic(X).
```

our ”toy” task - using a DCG grammar

- We can ask for the Prolog representation of such a rule:
-
```
?- listing(statement).
statement(A, [is], B, ['.'], C, D) :-
object(A, C, E),
'C'(E, is, F),
value(B, F, G),
'C'(G, '.', D).
Yes
```
- that means: C is put into the first (left) rule, object, as start. And in the end, we should get D. C is the sentence list that we put into the rule and D is the empty list.
  Ask for listing(object) and listing(det) to see more of the involved rules.

our "toy" task - using a DCG grammar

- OK, what can we do now?
- We can use all the linguistic representation mechanisms DCG gives us
- And we can use them in a normal Prolog program, which, of course, can be smarter than what I did here...

your job

- You can extend this in several ways:
- You could work on the linguistic capabilities of the system (question/statement - types)
- You could also think about the answers the system gives. What could make them useful in terms of finding something. Imagine you're standing at the church and ask the system how to find the school. You'd expect some kind of roadmap.

the end

- questions?