Jonathan Murray 9008081532

Logic programming ID2213 project

A logic-based chatbot prototype

1. Structure of the code

The code is divided into the following modules:

- **main.pl**
    - the longest file, containing
        - the main-loop
        - input/output
        - storing, reading and reasoning about knowledge
- **grammar.pl**
    - Defines the structure of sentences that the program can accept and a tiny dictionary with valid words.
    - Grammatical correctness is of little interest. The task of the program is to try to decide the intent of the user; not to act as a proof reader.
    - The parsing rules are written as DCG's.
    - Input is a list of words, and output is a syntax tree. Example below:
        - In: [peter, has, a, cat]
        - Out:

```
s(
        np(
                nnp(peter)
        )
        vp(
                verb(has)
                np(
                        dt(a)
                        nn(cat)
                )
        )
)
```

- **semantcs.pl**
    - Defines how semantic meaning is extracted from a syntax tree.
    - There is a rule handling each subtree from the grammar-module.
    - Input is a syntax tree, and output is a list of prolog-facts, a flag telling if the sentence was a question or not, and a list of variables that the question is concerned with. Example below:

        - In:

```
s(
	np(
		wh(who)
	)
	vp(
		verb(has)
		np(
			dt(a)
			nn(cat)
		)
	)
)
```

        - Out:
            - [

```
	action(ActionId, possess),
	actor(ActionId, Who),
	receiver(ActionId, CatId),
	be(CatId, cat)
]
```

            - true
            - [Who]
        - In the example above, the only atoms of the output are “possess” and “cat”. The rest of the arguments are unbound variables that will be matched against stored knowledge. However only the variable “Who” is interesting to the user, which is indicated by the third output argument.
- **mylists.pl**
    - a few general list-related predicates, that I couldn’t find in SWI-Prolog’s standard library
- **write_predicates.pl**
    - a few general write-related predicates, that I couldn’t find in SWI-Prolog’s standard library

2. Knowledge format

The following are the dynamic facts that are used by the system to represent knowledge given (and queried) by the user:

- possess/2
    - possess(X, Y): the entity represented by X has the entity represented by Y in its possession, for instance "peter has a dog"
- action/2, actor/2, receiver/2
    - These three define actions and the participants of the actions.
    - action(X, meet): X represents an action defined by the verb "meet"
    - actor(X, peter): peter is the active participant of the action X
    - receiver(X, lisa): lisa is the passive/receiving participant of the action X
    - The three facts above say that "peter met lisa", or "peter has met lisa" etc.
- be/2
    - be(X, Y): the entity represented by X is Y, for instance "peter is human"

- location/2
    - location(X, Y): the entity (or action) represented by X has location Y, for instance "peter is in the park"
- called/2
    - called(X, Y): the entity represented by X is called Y, for instance "peter's dog is called johny"
- property/2
    - property(X, Y): the entity represented by X has property Y, for instance "peter is cute"

A somewhat longer example of facts generated by a sentence follows below:

Sentence:
"peter met lisa s dog in the park." (the s in lisa's must be separated due to the simple parser and constraints in prolog)

Knowledge embedded in the sentence:
1. There is an entity called peter.
2. There is another entity called lisa.
3. lisa has an entity that is a dog. (represented by 4 facts)
4. peter met lisa's dog. (represented by 3 facts)
5. two possible interpretations of the mentioned location:
    - the meeting took place in the park
    - lisa's dog was in the park (clearly not the appropriate choice in this example)

Generated facts:
called(id_peter, peter). (1)

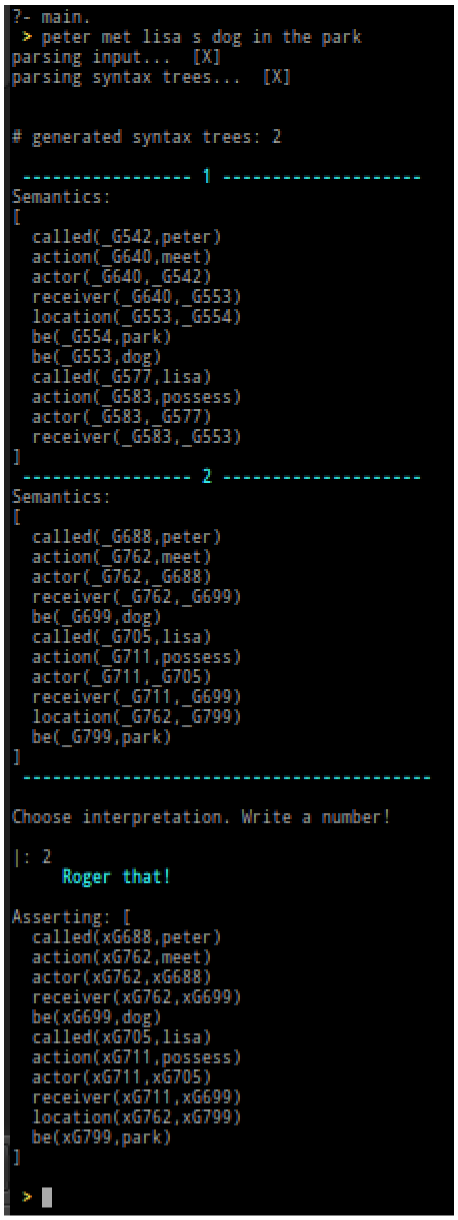
```
called(id_lisa, lisa).                    (2)
action(id_possess, possess).              (3)
actor(id_possess, id_lisa).               (3)
receiver(id_possess, id_dog).             (3)
be(id_dog, dog).                          (3)
action(id_met, meet).                     (4)
actor(id_met, id_peter).                  (4)
receiver(id_met, id_dog).                 (4)
location(id_met, id_park).                (5)
be(id_park, park).                        (5)
```

This rather lengthy format for storing the knowledge makes it possible to retrieve information such as:

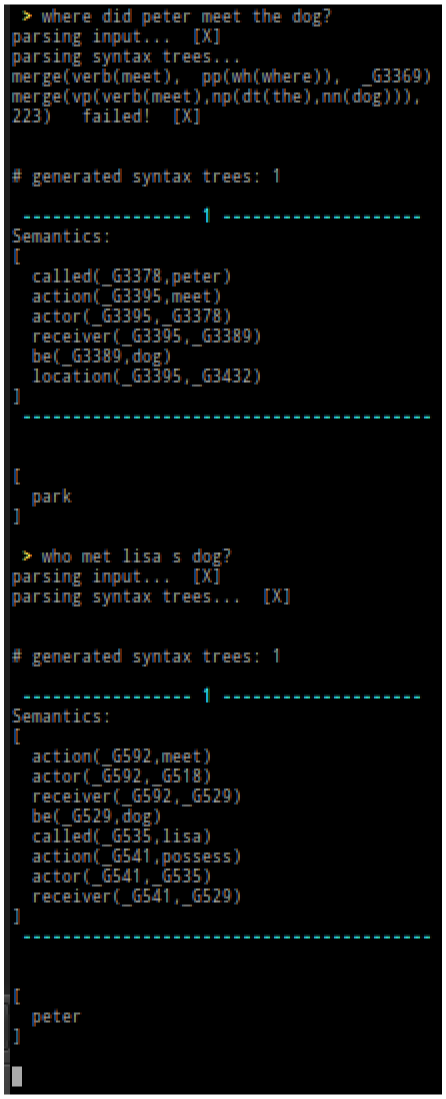
- who’s dog did peter meet in the park?
- where did peter meet a dog?
- who did peter meet?

3. An example user-session of the program

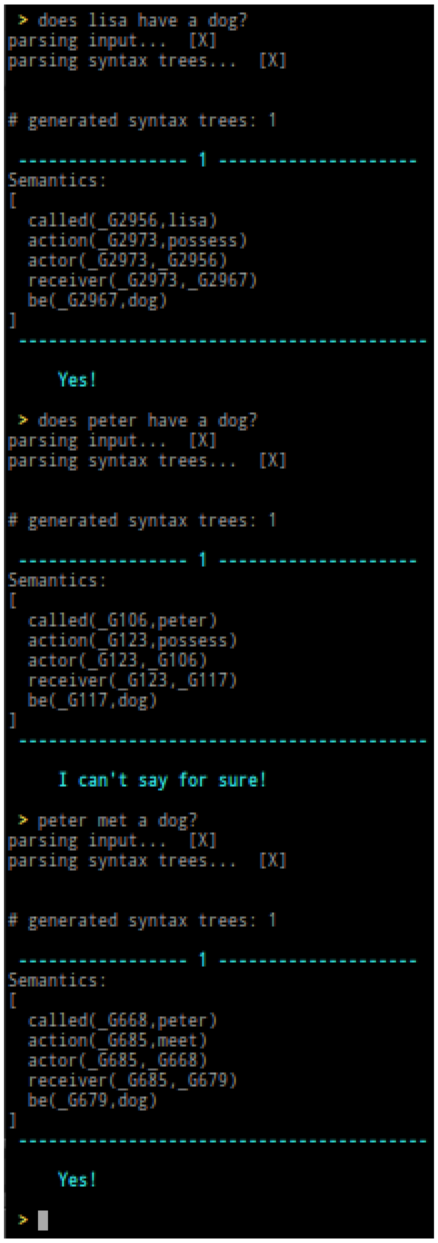
In the first three screenshots, only the semantic facts are printed, not the syntax trees.



```
?- main.
 > peter met lisa s dog in the park
parsing input...  [X]
parsing syntax trees...  [X]


# generated syntax trees: 2

 ----------------- 1 --------------------
Semantics:
[
  called(_G542,peter)
  action(_G640,meet)
  actor(_G640,_G542)
  receiver(_G640,_G553)
  location(_G553,_G554)
  be(_G554,park)
  be(_G553,dog)
  called(_G577,lisa)
  action(_G583,possess)
  actor(_G583,_G577)
  receiver(_G583,_G553)
]
 ----------------- 2 --------------------
Semantics:
[
  called(_G688,peter)
  action(_G762,meet)
  actor(_G762,_G688)
  receiver(_G762,_G699)
  be(_G699,dog)
  called(_G705,lisa)
  action(_G711,possess)
  actor(_G711,_G705)
  receiver(_G711,_G699)
  location(_G762,_G799)
  be(_G799,park)
]
 ----------------------------------------

Choose interpretation. Write a number!

|: 2
     Roger that!

Asserting: [
  called(xG688,peter)
  action(xG762,meet)
  actor(xG762,xG688)
  receiver(xG762,xG699)
  be(xG699,dog)
  called(xG705,lisa)
  action(xG711,possess)
  actor(xG711,xG705)
  receiver(xG711,xG699)
  location(xG762,xG799)
  be(xG799,park)
]

 >
```



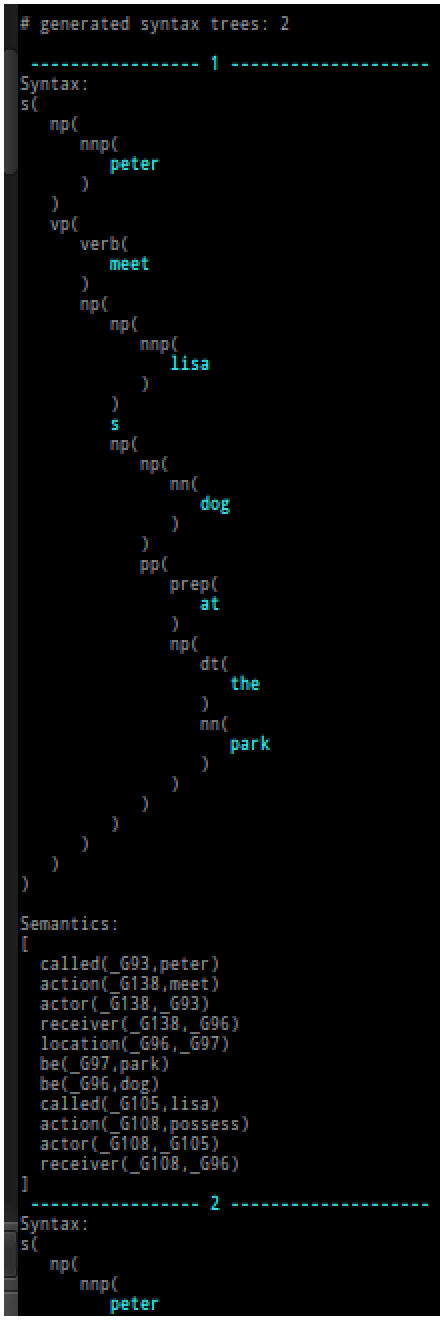
```
 > where did peter meet the dog?
parsing input...  [X]
parsing syntax trees...
merge(verb(meet),  pp(wh(where)),  _G3369)
merge(vp(verb(meet),np(dt(the),nn(dog))),
223)   failed!  [X]


# generated syntax trees: 1

 ----------------- 1 --------------------
Semantics:
[
  called(_G3378,peter)
  action(_G3395,meet)
  actor(_G3395,_G3378)
  receiver(_G3395,_G3389)
  be(_G3389,dog)
  location(_G3395,_G3432)
]
 ----------------------------------------


[
  park
]

 > who met lisa s dog?
parsing input...  [X]
parsing syntax trees...  [X]


# generated syntax trees: 1

 ----------------- 1 --------------------
Semantics:
[
  action(_G592,meet)
  actor(_G592,_G518)
  receiver(_G592,_G529)
  be(_G529,dog)
  called(_G535,lisa)
  action(_G541,possess)
  actor(_G541,_G535)
  receiver(_G541,_G529)
]
 ----------------------------------------


[
  peter
]

```



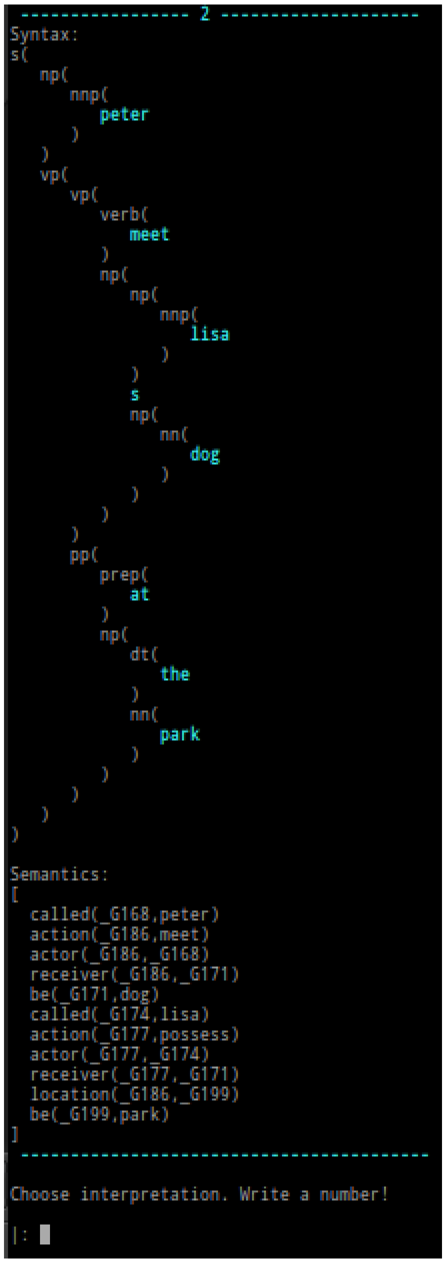
```
 > does lisa have a dog?
parsing input...  [X]
parsing syntax trees...  [X]


# generated syntax trees: 1

 ----------------- 1 --------------------
Semantics:
[
  called(_G2956,lisa)
  action(_G2973,possess)
  actor(_G2973,_G2956)
  receiver(_G2973,_G2967)
  be(_G2967,dog)
]
 ----------------------------------------

     Yes!

 > does peter have a dog?
parsing input...  [X]
parsing syntax trees...  [X]


# generated syntax trees: 1

 ----------------- 1 --------------------
Semantics:
[
  called(_G106,peter)
  action(_G123,possess)
  actor(_G123,_G106)
  receiver(_G123,_G117)
  be(_G117,dog)
]
 ----------------------------------------

     I can't say for sure!

 > peter met a dog?
parsing input...  [X]
parsing syntax trees...  [X]


# generated syntax trees: 1

 ----------------- 1 --------------------
Semantics:
[
  called(_G668,peter)
  action(_G685,meet)
  actor(_G685,_G668)
  receiver(_G685,_G679)
  be(_G679,dog)
]
 ----------------------------------------

     Yes!

 >
```

The screenshots below show the syntax trees of the input sentence, highlighting the two different interpretations of the sentence.

Jonathan Murray 9008081532



```
# generated syntax trees: 2

 ----------------- 1 --------------------
Syntax:
s(
   np(
      nnp(
         peter
      )
   )
   vp(
      verb(
         meet
      )
      np(
         np(
            nnp(
               lisa
            )
         )
         s
         np(
            np(
               nn(
                  dog
               )
            )
            pp(
               prep(
                  at
               )
               np(
                  dt(
                     the
                  )
                  nn(
                     park
                  )
               )
            )
         )
      )
   )
)

Semantics:
[
  called(_G93,peter)
  action(_G138,meet)
  actor(_G138,_G93)
  receiver(_G138,_G96)
  location(_G96,_G97)
  be(_G97,park)
  be(_G96,dog)
  called(_G105,lisa)
  action(_G108,possess)
  actor(_G108,_G105)
  receiver(_G108,_G96)
]
 ----------------- 2 --------------------
Syntax:
s(
   np(
      nnp(
         peter
```



```
 ----------------- 2 --------------------
Syntax:
s(
   np(
      nnp(
         peter
      )
   )
   vp(
      vp(
         verb(
            meet
         )
         np(
            np(
               nnp(
                  lisa
               )
            )
            s
            np(
               nn(
                  dog
               )
            )
         )
      )
      pp(
         prep(
            at
         )
         np(
            dt(
               the
            )
            nn(
               park
            )
         )
      )
   )
)

Semantics:
[
  called(_G168,peter)
  action(_G186,meet)
  actor(_G186,_G168)
  receiver(_G186,_G171)
  be(_G171,dog)
  called(_G174,lisa)
  action(_G177,possess)
  actor(_G177,_G174)
  receiver(_G177,_G171)
  location(_G186,_G199)
  be(_G199,park)
]
 -----------------------------------------

Choose interpretation. Write a number!

|: 
```

4. Discussion

Prolog is a language well fitted for this kind of task, and the program could easily be extended to support more advanced logic.

- Rules could be defined for handling actions that are known to be reflective. (If peter met lisa in the park, lisa also met peter in the park.)
- Global quantifiers: A sentence like “peter likes dogs” could generate more advanced rules such as:
    - action(x, like).
    - actor(x, peter_id).
    - receiver(x, Receiver):-
      be(Receiver, dog).
- Cause and consequence: A sentence like “peter cried because lisa sang” could generate a fact relating the two actions
    - cause(sing_action, cry_action).

  A query such as “why did peter cry”, would generate
    - cause(X, cry_action)

  and result in X unifying to sing_action.