CSE6390E PROJECT REPORT

HALUK MADENCIOGLU

April 20, 2010

This report describes the design and implementation of a system of programs to deal with parsing natural language, building semantic structures and making first order logic inferences.

TABLE OF CONTENTS

1. INTRODUCTION

Understanding natural language is a vast topic which includes many different approaches. All of these approaches have some common building blocks related to concepts inherited from the underlying linguistic theory. Nevertheless, the subsystems dealing with parsing, grammar rules, syntax checking and semantic interpretation can be highly dependent on the dynamics of the methodology used.

Parsing and syntax checking are fundamental parts of the understanding process since without them being successful, the soundness of any attempt to analyze natural language would be debatable. The semantic interpretation is a very crucial part in this process, as the extraction of meaning may establish a solid basis for building further AI applications. Making inferences based on extracted meaning needs another capability in the system – a logic programming framework

The logic programming based approach to natural language understanding using Prolog has been a topic of interest for quite some time. The inherent capabilities of a logic programming language for dealing with first order logic make this approach quite powerful when making inferences as well as understanding the language is targeted.

1.1 About Logic Programs and Prolog

The scope of logic programming explained here refers to monotonic reasoning and definite logic programs.

Monotonic reasoning establishes a logic system based on the assumption (or rather axiom) that adding knowledge to a knowledge base (set of truth sentences) does not reduce the set of inferences that can be made from that knowledge base. This may be a well established axiom if considered in a world of static truth sentences, but it can not capture interaction between entities. As an example, one can say “some vase has a cylindrical shape and I turned on the light” then the vase is still cylindrical. But if instead the sentence ended as “I dropped the vase” then a piece of information is gone with the broken vase.

Another problem with basic knowledge representation using first order logic is observed in defining the effects of change. It seems that unless everything is not stated one by one for what changes and what does not in case of a single change in knowledge base, the completeness of that knowledge base is not satisfied. As an example, consider the sentence “my pencil is 12 centimeters long”. The system must be told that this truth does not change when “I turn the light on”. There must be a set of axioms in the system what is called “frame axioms” to explicitly state all these facts of static existence, including “my pencil’s length does not change when I turn on the light”. This issue is named as “the framing problem”.

Definite logic programs are logic programs with no negative literals (this is, in a way related to the above framing problem of not being able to infer negative result by default). In a definite logic program all sentences have the form $A \leftarrow B_1,B_2,B_3\ldots\ldots B_k$. When a set of goals is given as $G_1,G_2,G_3\ldots\ldots G_k$, the system searches for a way to satisfy it, that is, it tries to find a constructive proof for the *existence of a set of variables and predicates* which satisfy the set of goals.

The rule called SLD (Linear resolution with Selection function for Definite clauses) is how the system selects a goal from the set of given goals. Once the rule is selected, a continuous application of unification till refutation (empty sentence) constructs the proof and binds the variables as the proof is built up. The result is a “Yes” or “No” together with a possible set of variable assignments.

These issues above are important to emphasize as they partly limit what can be done in a simple first order logic programming system with no further extensions.

Prolog, the logic programming language, has some nice features to be used in NLP. It has built-in list processing, and mostly is able to parse natural language using DCG structures (definite clause grammar). Furthermore, the way structures are built using the same Prolog syntax as for the predicates. This makes it a perfect choice for semantic extraction.

1.2 The Project

The main target of this project was to build a natural language processing system which:

1. is a Prolog/unification based project,
2. has a DCG parser
3. parses English sentences for grammatical correctness and extracts meaning
4. checks whether the syntax is correct also looking for verb tenses and noun multiplicities
5. has an inference engine that makes inferences to respond to input / questions asked
6. transforms English sentences in first order logic format into Prolog clauses and then uses SLD resolution built in Prolog to evaluate the first order logic sentence
7. learns a grammatically correct sentence input as a new fact - or responds if there is an inconsistency with the knowledge base

The work presented here seems to have achieved all results except dealing with inconsistencies promised in 7 above. That issue in particular is related to what is presented earlier as the framing problem. While working on the project I recognized that handling that problem would need a further refined approach using situation calculus, which in turn would make the project impossible to complete in limited time. But that end is still open, i.e. incorporating a situation calculus based program into this system will enable it to respond to queries in the case when knowledge is attempted to be altered.

The other issue is related to the scope of the project. Here my ambition was to obtain a system which understands first order logic given as natural language sentence. Thus the project concentrates more on meaning extraction and handling first order logic sentences than dealing with syntactic difficulties of the language. Rougly, a limited fragment of the language is used. In that scope some difficult issues are skipped such as:

- verb phrases with both a dative and accusative object
- relative clauses
- “wh – words” other than who
- interrogative sentences other than starting with is / are / did/ do / does
- adjective phrases including a noun, adjective specifiers, degree words, verb specifiers (adverbs),
- phrases involving numbers

The set of examples provided will explain the scope of the system best. Please examine the sample output for a better understanding of the system’s capabilities.

There are four categories of programs in this project. The program in the first category named parser.pl deals with reading English sentences into Prolog lists of tokens. This is a simple list processing program where the I/O subsystem is used to read input from standard input and lists are built. Later the list built here can be passed to others which need a list input.

The second category of programs is to deal with parsing, syntax checking and conversion of natural language queries into first order logic. The program fol.pl parses a Prolog list of tokens and uses DCG formalism to load the parts of speech and verb / noun phrases into Prolog structures. The execution of this program results in a 'No' answer or a 'Yes' and the semantic representation of the given English sentence. Here, the specifiers for first order logic are used to build combined sentences with logical connectives. These are "a / the / every /and/ or." The determiner 'a' corresponds also to the existential quantifier. 'Every' is used for universal quantification, 'and' and 'or' have their natural meaning. So you may not only validate the grammar of a given sentence for syntax but also build first order logic phrases as you parse. All valid results are saved as Prolog predicates into a database called 'world'. Nevertheless, the quantified sentences are not saved. They are just converted to first order logic queries. Also questions with is/are/does/did/do/who are not saved, but the user gets a valid answer for those simple questions. That means only declarative sentences are saved into the database. Later, inference engine will make use of that database to answer first order logic queries. Remember that there is no inference making in this program.

The program fol.pl uses the lexicon database program lexicon.pl for checking parts of speech. That lexicon is a long list of Prolog ground predicates. This is where correct syntax including tense and multiplicity is stored.

The programs of third category are designed to answer queries of first order logic, given an input query in the correct format. For example the correct format for universal quantification is every(_G469, man(_G469)=>husband(_G469)). That should have been already produced by the fol.pl program above issuing a command such as: fol(S,[every, man, is, a, husband]). The program query-dbase.pl does this inference. It accepts all valid forms of first order logic queries and checks the database 'world' depending on the meaning of the query, quantifiers, connectives, etc. and answers it. The other program, world.pl is the initial Prolog database representing the world as mentioned above.

2. ALGORITHMS, METHODS AND DATA STRUCTURES

General Information:

To give a complexity measure for a Prolog code in terms of the operational semantics procedural programs is quite difficult. But I can say that the complexity increases combinatorically as new facts are added to the database, since each new fact increases the possibilities of goal satisfaction. As new combinations are possible and Prolog has to search through all, no exact prediction can be made. Nevertheless, Prolog has its own ways to deal with complexity. The 'cuts' were used whenever there was an infinite iteration or when pruning the search tree would not affect the search results. Another technique of reordering the goals was used to avoid creation of useless sub-goals before a match is hit.

Program parser.pl:

This program parses a sentence word by word and each word letter by letter reducing input read from standard input and skipping space. It uses forward recursion with the lists so the complexity of the program is O(non white space characters in input).

Programs world.pl and lexicon.pl:

Program world.pl stores an initial set of facts. This is the initial knowledge base and lexicon.pl is the grammar checking base. These programs are database-like programs holding variants of a set of predicates, so the complexity related to these programs may be considered as the complexity when a call is made to those predicates from outside. Since Prolog reads every line from top to bottom until it finds a match, the complexity of that call will be linear in number of lines in the database.

Program fol.pl

One of the key ideas used here is to make use of the built-in DCG structure to make difference lists. If everything was made through using append calls, the program could take enormous time to finish due to the additional goals generated by Prolog which would be wasted as the search proceeds in the SLD search tree. So DCG code s(S) --> np(NP),vp(VP) (an oversimplified fragment) is actually s(S,X) :- np(S,Y),vp(Y,X) in Prolog. If needed, additional Prolog commands may be added in DCG comma separated list on the right of arrow, but they must be in curly braces. That means a call to the Prolog system itself rather than searching to satisfy a difference list partitioning of the input. Once translated, when accessed, these difference lists show a linear complexity in the length of the sentence in terms of the number of words.

The most innovative key idea used here is the lambda calculus implementation over Prolog. In lambda calculus λx.f(x) defines a function and an argument 'x'. In Prolog, properties, or predicates are expressed as property(X). Both have the power to extract the meaning of a phrase if the semantic structure is thought as a variable or a constant having a property. For the lambda calculus, later when the built function is required to be applied on an argument, it is shown as λx.f(x) (argument). This is called *Beta reduction*. The equivalent to this is unification in Prolog where the expression X, property(X), binds X to have the true value for the predicate 'property'. If there is more than one variable in lambda calculus, then a partial reduction is done. For example λy.λx.f(x, y) when reduced by an argument1 becomes λx.f(x, argument1). So if symbolic reduction can be handled in Prolog, then that means the difficult to implement lambda calculus can be implemented and while the sentence is parsed, and the **partially initialized structures** may be kept in list of Prolog goals still to be reduced. What this means is that we can both parse the sentence and **incrementally build the semantic structures** to hold meaning. This idea is used extensively in all predicates. Generally the determiner, the quantifier, the verb or the adjective is the root of the semantic structure. When abstracted from that notation, all program sentences are well known Prolog sentences. So search complexity is as one that would occur in such a program, bearing in mind the precautions applied as explained above. One issue specific to grammar handling is left recursion. The best solution to handle that issue is to create intermediate goals when an infinite iteration occurs due to left recursion. But again, these are extreme cases of infinite iterations. A feature could have been added to the program to guess a general case for most of the tenses or plurals. This rule could first try to see if it is in the irregular verbs or nouns list, if not, s/es for present singular and d/ed for past could be derived automatically. Similar for the plural of the nouns applies. Then the loaded lexicon could contain just the base word and hence search complexity could be reduced. But that idea appeared by the time I was writing this report.

This program was the most difficult of all because I had to decide on the extracted form of semantic content and on how to limit the scope of queries. A sample sentence like ‘a husband is a man.’ gives ‘some(_G437, husband(_G437)&man(_G437))’ , but also the query ‘is a husband a man.’ gives the same result. On the other hand, ‘Larry is good’ gives good(‘Larry’) but the question ‘who is good’ results in ‘good(_G416)’. Remember that only non-quantified and at the same time simple declarative sentences (no connectives) are stored in the ‘world’ database when the program is executed.

Prolog offers some predicates such that one can build predicates from within the program and call them. The predicate **=..** is used to construct temporary predicates stored in variables and they are used to build more complex structures. When finished, only the complete structure is stored and those kept in variables are gone. An excellent example to predicate construction is building noun/verb/adjective phrases from data in lexicon before beta reduction is applied on those new predicates.

The overall complexity of dealing with exceptional cases and the necessity to achieve completeness of the system within the bounds of the language fragment used forced me to deal with rules piece by piece sometimes. That is why you may notice a rule is repeated sometimes to deal with exceptional cases.

Program query-dbase.pl:

This program again makes extensive use of structure building predicates and when necessary uses those predicates. The core of the program is based on symbolic manipulation. Quantifiers and connectives are parsed and interpreted, then validity is checked by calling components as directed by the semantics of the given query (every / some / or / and). The method used for existential quantification is enforcing a double negation. This means “if the predicate formed from the query does not fail”. The universal quantification does almost the same. But before that the program forms a domain of true predicates. The universally quantified sentence is true if every predicate in the domain with the same name as in quantification is true. The parsing of query into sub-queries is linear operation in terms of Prolog terms and connectives/quantifiers. A simple query may not take much time to answer if the world database is not too big. On the other hand a universally quantified sentence requires an exhaustive search so this is the most demanding query of all.

3. EXAMPLES OF OUTPUT

The examples in this section are of utmost importance to understand the limits of the system. When there is a negative reply because of a grammatical mistake that will be emphasized. Other failures must serve as indications of cases where the system breaks down. The sample input sentences are chosen to give a complete set of examples.
In the sample executions below, first the command issued is shown and then the response. The first line is the actual command and the line which starts with |: is where Prolog is waiting for an input.

The first part below is related to the program fol.pl. Here you will see how the meaning is obtained and stored with the outcome of a first order logic sentence.

Please remember that you have to end every sentence with a full stop even if it is a question and must not use any other punctuation anywhere else.

___SEMANTIC EXTRACTION AND TRANSLATION INTO FIRST ORDER LOGIC ____

1- *Declarative 'is' sentence with regular nouns:*

```
parse(X),fol(S,X).
|: a husband is a man.


X = [a, husband, is, a, man]
S = some(_G431, husband(_G431)&man(_G431))

Yes

?- parse(X),fol(S,X).
|: the weather is cold.


X = [the, weather, is, cold]
S = the(_G446, weather(_G446), cold(_G446))

Yes


?- parse(X),fol(S,X).
|: the sandwiches are good.


X = [the, sandwiches, are, good]
S = the(_G458, sandwiches(_G458), good(_G458))

Yes
```

Observe the use of adjective.

BUT NOT:

```
?- parse(X),fol(S,X).
|: weather is cold.


No      (missing determiner)
```

OR NOT:

```
?- parse(X),fol(S,X).
|: the brother is a good man.

No
```

The reason for this failure is that this sentence must assert that the brother is a man and the brother is good. That complex expression could not be handled by the system. It is a missing feature.

2- *Declarative 'is' sentence with regular plural nouns:*

```
?- parse(X),fol(S,X).
|: husbands are men.

No
```

The reason this is not handled was the difficulty in storing the semantic value extracted from this sentence. It is ambiguous. Are all husbands men or some are, some aren't?

3- *Declarative 'is' sentence with proper nouns:*

```
?- parse(X),fol(S,X).
|: Diane is a waiter.


X = ['Diane', is, a, waiter]
S = waiter('Diane')

Yes
```

Notice that Diane is quoted. That is you write a proper noun with a capital letter and parser program turns it into a quoted atom.

BUT (wrong syntax):

```
?- parse(X),fol(S,X).
|: Diane is a waiters.


No
```

4- *Declarative sentence with intransitive verb and regular nouns:*

```
?- parse(X),fol(S,X).
|: a man laughed.
```

```
X = [a, man, laughed]
S = some(_G422, man(_G422)&laughed(_G422))

Yes
```

BUT (wrong syntax):

```
?- parse(X),fol(S,X).
|: a man laugh.

No
```

5- *Declarative sentence with transitive verb and regular nouns as subject and object:*

```
?- parse(X),fol(S,X).
|: a woman wants a car.

X = [a, woman, wants, a, car]
S = some(_G440, woman(_G440)&some(_G478, car(_G478)&wants(_G440, _G478)))

Yes
```

BUT (wrong syntax):

```
?- parse(X),fol(S,X).
|: a girls wants a car.

No
```

Similarly (missing article),

```
16 ?- parse(X),fol(S,X).
|: a woman wants car.

No
```

6- *Declarative sentence with transitive verb and proper nouns as subject and object:*

```
?- parse(X),fol(S,X).
|: Jim loves Cathy.
X = ['Jim', loves, 'Cathy']
S = loves('Jim', 'Cathy')

Yes
```

7- *Declarative sentence with transitive verb and proper noun as subject and regular noun as object:*

```
?- parse(X),fol(S,X).
|: Tom closed the door.

X = ['Tom', closed, the, door]
S = the(_G442, door(_G442), closed('Tom', _G442))

Yes

?- parse(X),fol(S,X).
|: women want car.

No    (determiners missing)


?- parse(X),fol(S,X).
|: women want the car.

No (determiner of subject missing)


?- parse(X),fol(S,X).
|: the women want the car.

X = [the, women, want, the, car]
S = the(_G452, women(_G452), the(_G488, car(_G488), want(_G452, _G488)))

Yes

?- parse(X),fol(S,X).
|: the women want the cars.

X = [the, women, want, the, cars]
S = the(_G455, women(_G455), the(_G491, cars(_G491), want(_G455, _G491)))

Yes
```

Observe how the semantic structure has the determiner as the root.

8- ***Declarative sentence with transitive verb and regular noun as subject and proper noun as object:***

```
?- parse(X),fol(S,X).
|: a man kissed Sue.


X = [a, man, kissed, 'Sue']
S = some(_G431, man(_G431)&kissed(_G431, 'Sue'))

Yes
```

OR similarly:

```
?- parse(X),fol(S,X).
|: the  sisters love Sue.

X = [the, sisters, love, 'Sue']
S = the(_G443, sisters(_G443), love(_G443, 'Sue'))

Yes
```

BUT NOT (missing determiner):

```
?- parse(X),fol(S,X).
|:   sisters love Sue.

No
```

9- Yes/No question with subject regular noun and object regular noun:

```
?- parse(X),fol(S,X).
|: is a wife a woman.

X = [is, a, wife, a, woman]
S = some(_G441, wife(_G441)&woman(_G441))

Yes


?- parse(X),fol(S,X).
|: are the men the waiters.

X = [are, the, men, the, waiters]
S = the(_G459, men(_G459), waiters(_G459))

Yes
```

BUT NOT:

```
?- parse(X),fol(S,X).
|: is the wives a woman.

No
```

OR NOT:

```
?- parse(X),fol(S,X).
|: are the men waiters.

No   (missing determiner)
```

10- ***Yes/No question with subject proper noun and object regular noun:***

```
?- parse(X),fol(S,X).
|: is John  a brother.

X = [is, 'John', a, brother]
S = brother('John')

Yes
```

BUT NOT:

```
?- parse(X),fol(S,X).
|: is John  Bill.

No
```

11- ***Do-word question with proper noun and intransitive verb*****:**

```
?- parse(X),fol(S,X).
|: did Larry sleep.

X = [did, 'Larry', sleep]
S = sleep('Larry')

Yes
```

BUT NOT:

```
?- parse(X),fol(S,X).
|: did Larry sleeps.

No
```

OR NOT:

```
?- parse(X),fol(S,X).
|: does Larry slept.

No
```

12- ***Do-word question with regular noun and intransitive verb*****:**

```
?- parse(X),fol(S,X).
|: does a boy sleep.

X = [does, a, boy, sleep]
S = some(_G438, boy(_G438)&sleep(_G438))

Yes

?- parse(X),fol(S,X).
|: do the brothers run.
```

```
X = [do, the, brothers, run]
S = the(_G447, brothers(_G447), run(_G447))

Yes
```

BUT NOT

```
?- parse(X),fol(S,X).
|: do the brothers runs.

No
```

13- ***Do-word question with proper nouns as subject and object and transitive verb*****:**

```
?- parse(X),fol(S,X).
|: did Larry love Sue.


X = [did, 'Larry', love, 'Sue']
S = love('Larry', 'Sue')

Yes
```

14- ***Do-word question with proper nouns as subject and regular noun as object and transitive verb*****:**

```
?- parse(X),fol(S,X).
|: did Larry read a book.

X = [did, 'Larry', read, a, book]
S = some(_G455, book(_G455)&read('Larry', _G455))

Yes
```

15- ***Do-word question with regular nouns as subject and object and transitive verb*****:**

```
?- parse(X),fol(S,X).
|: did a girl love the hamburgers.

X = [did, a, girl, love, the, hamburgers]
S = some(_G483, girl(_G483)&love(_G483, _G518))

Yes

?- parse(X),fol(S,X).
|: did the girls love the hamburgers.
```

```
X = [did, the, girls, love, the, hamburgers]
S = the(_G492, girls(_G492), love(_G492, _G525))

Yes
```

BUT NOT:

```
?- parse(X),fol(S,X).
|: did the girls love hamburgers.

No
```

Observe that again there is a missing determiner for object so it is not clear *some hamburgers* or *all hamburgers* or *the hamburgers* are mentioned.

16- *Do-word question with regular noun as subject and proper noun as object and transitive verb*:

```
?- parse(X),fol(S,X).
|: did the girls love Tom.

X = [did, the, girls, love, 'Tom']
S = the(_G459, girls(_G459), love(_G459, 'Tom'))

Yes
```

17- *Who question is- verb, regular noun*

```
?- parse(X),fol(S,X).
|: who is a mother.

X = [who, is, a, mother]
S = mother(_G438)

Yes

?- parse(X),fol(S,X).
|: who are the husbands.

X = [who, are, the, husbands]
S = husbands(_G446)

Yes

?- parse(X),fol(S,X).
|: who are good.

X = [who, are, good]
S = good(_G422)

Yes
```

Observe how Prolog inserted a variable in the predicate.

18- ***Who – intransitive verb, regular noun***

```
?- parse(X),fol(S,X).
|: who slept.

X = [who, slept]
S = slept(_G413)

Yes
```

19- ***Who – transitive verb, regular nouns as subject and object***

```
?- parse(X),fol(S,X).
|: who wants a house.


X = [who, wants, a, house]
S = some(_G440, house(_G440)&wants(_G437, _G440))

Yes

?- parse(X),fol(S,X).
|: who ate the sandwiches.


X = [who, ate, the, sandwiches]
S = the(_G455, sandwiches(_G455), ate(_G452, _G455))

Yes
```

Observe how the determiner acts as the root of the semantic structure.

BUT NOT:

```
?- parse(X),fol(S,X).
|: who does eat a sandwich.


No
```

That type of question is not handled. It is a missing feature.

20- ***Who – transitive verb, regular noun as subject and proper noun as object***

```
?- parse(X),fol(S,X).
|: who knew Cathy.


X = [who, knew, 'Cathy']
```

```
S = knew(_G428, 'Cathy')

Yes
```

20- *Connectives and logical operators; various sentences*
This part is the most exiting experiment before the next. Here we get FOL sentences as bindings.

```
?- parse(X),fol(S,X).
|: Tom loves a woman and the weather is cold.


X = ['Tom', loves, a, woman, and, the, weather, is, cold]
S = some(_G514, woman(_G514)&loves('Tom', _G514))&the(_G577, weather(_G577),
cold(_G577))

Yes


?- parse(X),fol(S,X).
|: every boy loves a girl and a man eats a hamburger.


X = [every, boy, loves, a, girl, and, a, man, eats|...]
S = every(_G536, boy(_G536)=>some(_G574, girl(_G574)&loves(_G536,
_G574)))&some(_G637, man(_G637)&some(_G675, hamburger(_G675)&eats(_G637,
_G675)))

Yes


?- parse(X),fol(S,X).
|: every worker wanted a house or  a student read a book.


X = [every, worker, wanted, a, house, or, a, student, read|...]
S = every(_G533, worker(_G533)=>some(_G571, house(_G571)&wanted(_G533,
_G571)))\/some(_G634, student(_G634)&some(_G672, book(_G672)&read(_G634, _G672)))

Yes

?- parse(X),fol(S,X).
|: every man loves every woman.


X = [every, man, loves, every, woman]
S = every(_G458, man(_G458)=>every(_G496, woman(_G496)=>loves(_G458, _G496)))
```

WORKING WITH FIRST ORDER LOGIC

Now the examples will show how the system interprets logic syntax.

Additionally we must use satisfy(world,S) predicate form quer-dbase.pl here.

Initially this is the world:

```
world(woman('Cathy')).
world(woman('Sue')).
world(man('Jim')).
world(man('John')).
world(man('Tom')).
world(man('Larry')).
world(woman('Diane')).

world(loves('Jim','Cathy')).
world(loves('Tom','Diane')).
world(loves('Larry','Diane')).


world(wants('Bill',house)).
world(wants('Maggie',house)).
```

So by examining the situation, we can’t say that every man loves a woman.

```
?- parse(X),fol(S,X),satisfy(world,S).
|: every man loves a woman.


No
```

Let’s add the knowledge that John also loves a woman.

```
?- parse(X),fol(S,X).
|: John loves Sue.

X = ['John', loves, 'Sue']
S = loves('John', 'Sue')

Yes
```

Now:

```
?- parse(X),fol(S,X),satisfy(world,S).
|: every man loves a woman.


X = [every, man, loves, a, woman]
S = every(_G566, man(_G566)=>some(_G604, woman(_G604)&loves(_G566, _G604)))

Yes
```

As observed above, the added fact by the execution of :

```
?- parse(X),fol(S,X).
|: John loves Sue.
```

enabled us to get a positive result for the universally quantified sentence.

There is still an ambiguity in this query which the system can not handle. Is the referenced woman a unique woman or for every man there is a woman but not necessarily the same one?

We humans can not answer this either. But the way the system handles it is clear from the context. The system does not look for uniqueness, it is just any woman.

Similarly let's try:

```
?- parse(X),fol(S,X),satisfy(world,S).
|: every man loves every woman.


No
```

And add the missing facts: (one by one for them to be included in the database)

```
?- parse(X),fol(S,X).
|: John loves Diane.

X = ['John', loves, 'Diane']
S = loves('John', 'Diane')

Yes

?- parse(X),fol(S,X).
|: John loves Cathy.

X = ['John', loves, 'Cathy']
S = loves('John', 'Cathy')

Yes
?- parse(X),fol(S,X).
|: Larry loves Cathy.
X = ['Larry', loves, 'Cathy']
S = loves('Larry', 'Cathy')

Yes

?- parse(X),fol(S,X).
|: Larry loves Sue.

X = ['Larry', loves, 'Sue']
```

```
S = loves('Larry', 'Sue')

Yes

?- parse(X),fol(S,X).
|: Tom loves Sue.

X = ['Tom', loves, 'Sue']
S = loves('Tom', 'Sue')

Yes

?- parse(X),fol(S,X).
|: Tom loves Cathy.

X = ['Tom', loves, 'Cathy']
S = loves('Tom', 'Cathy')

Yes

?- parse(X),fol(S,X).
|: Jim loves Sue.


X = ['Jim', loves, 'Sue']
S = loves('Jim', 'Sue')

Yes
?- parse(X),fol(S,X).
|: Jim loves Diane.

X = ['Jim', loves, 'Diane']
S = loves('Jim', 'Diane')

Yes
```

Amazingly, now the query is successful:

```
?- parse(X),fol(S,X),satisfy(world,S).
|: every man loves every woman.


X = [every, man, loves, every, woman]
S = every(_G584, man(_G584)=>every(_G622, woman(_G622)=>loves(_G584, _G622)))

Yes
```

But look at the following example:

```
?- parse(X),fol(S,X),satisfy(world,S).
```

```
|: Bill wants a house and Maggie wants a car.


X = ['Bill', wants, a, house, and, 'Maggie', wants, a, car]
S = some(_G622, house(_G622)&wants('Bill', _G622))&some(_G687,
car(_G687)&wants('Maggie', _G687))

Yes
```

What happened above?

The only predicate in the world database about Maggie was : world(wants('Maggie',house)).

The problem is that while fol(S,X), executed, as soon as it saw a proper name as Maggie, it appended the fact it learned to the database.

To check:

```
?- world(some(_G687, car(_G687)&wants('Maggie', _G687))).

_G687 = _G512 ;
```
(so it was stored in the knowledge base)

No (No here appears because I pressed semicolon to look for other results. It is not a failure)

This can be considered as a good side effect if only FOL queries are important. But on the other hand it may be considered as a defect since the way the knowledge stored is different from that of the world database.

4. PROGRAM CODE

Please see the attached printouts as it was not very practical to include code here because of page orientation.

5. REFERENCES

Bratko, I. (2001). Prolog Programming for Artificial Intelligence (3rd ed.). Harlow, England: Pearson Education Limited.

Brewka, G., & Dix, J. (1997). Knowledge representation with logic programs.
Retrieved from http://ii.fmph.uniba.sk/~sefranek/cudzie/bd.pdf.

Clocksin, W. F., & Mellish, C. S. (2003). Programming in Prolog (5th ed.). Berlin, Germany: Springer- Verlag.

Cooper, R., Lewin, I., & Black, A. W. (1993). Prolog and natural language semantics.
Retrieved from http://www.ling.gu.se/~cooper/papers/comp-sem.pdf.

Lloyd, J. W. (1984). Foundations of Logic Programming. Berlin, Germany: Springer-Verlag.

Nilsson, U., & Maluszynski, J. (2000). Logic, Programming and Prolog (2nd ed.). Linkoping, Sweden: Linkoping University.

Nugues, P. M. (2006). An Introduction to Language Processing with Perl and Prolog: An Outline of Theories, Implementation, and Application with Special Consideration of English, French, and German. Gabbay, M., & Siekmann, J. (ed.). Heidelberg, Germany: Springer-Verlag.

O'Keefe, R.. A. (1990). The Craft of Prolog. Cambridge, MA: The MIT Press .

Pereira, F. C. N., & Shieber, S. M. (2002). Prolog and Natural-Language Analysis. Retrieved from http://www.mtome.com/Publications/PNLA/prolog-digital.pdf.

6. APPENDIX – Project/Program Systems Manual

The whole set of programs must be in the same directory. Copy all files to some directory. Change to that directory. On a Windows system changing to that directory is meaningless since it is GUI based, just start the program.

These programs were developed on SWI-Prolog. So please open up an SWI-Prolog session (from the directory where the files are copied to – if using Unix).

On a Unix system, type pl or swi-pl depending on your installation, if unsure issue command "apropos prolog" and follow the information.

Then type **consult('fol.pl').** at the Prolog interpreter prompt. This file will also load the other 4 files needed. Do not get surprised by the amount of warnings. They occur because Prolog thinks some variables are useless, but the underlying symbolic manipulation for lambda calculus needs them in place.

Steps of increasing complexity:

To see how the parser works alone, **type parser.** at the prompt. This will prompt you showing **|:** sign. Type your English sentence without using any commas or other punctuation symbols and always end the sentence with a full stop—even questions.

Then you will get the Prolog list filled with your words and anything that starts with capital letter will be single quoted.

To see the translation using the DCG grammar parser and FOL translator, type : **parse(X),fol(S,X).** Then the same prompt for input appears and when you type your sentences, if they are syntactically valid, a query of FOL will be formed and bound to the S variable. Follow section 3 examples for further details.

To see how the inference engine works, type: **parse(X),fol(S,X),satisfy(world,S).**
This will read your sentence, parse it, translate into FOL, and query the database world.pl. For further details, see the examples at the end of section 3.

If you need more facts, you can write them into world.pl. Follow the same format for predicates. Same applies to lexicon.pl. If you need to extend the lexicon, write the facts there.

Do not rename the two knowledge base files unless you modify fol.pl fie to consult the new files.