CS 421 – Natural Language Processing – Spring 2019
Term Project (Part 2)

1 General Information

Deadline: Sunday 5/5, 11:59pm
Worth: 220 points (+ 50 extra credit **only for undergraduates**)
Implementation Language: any of Java or Python. Please consult with the instructor and especially with the TA if you want to use any other common language.

In the second part of the project, you will derive a semantic representation from the syntactic trees the parser you used in part 1 returns to you. To provide a semantic interpretation for the input means to transform the input into a suitable representation, and then build a small interpreter that uses the semantic representation to compute answers from the database. To give you a taste of a realistic application, we are using SQL, a database query language, rather than first order logic, like the book does. As you will see, a SQL query can be built by using similar semantic attachments to the ones used for logic.

IMPORTANT NOTE: Simplification. To make the assignment doable, we will significantly simplify the queries from Part 1. Part 1 was meant to give you a taste of how a broader range of queries would be parsed; Part 2 a taste of how complicated deriving a semantic representation is. Please see additional file **project-part2-questions.pdf** for the specific queries you are asked to answer.

Additionally:

1. **For undergraduates:** you will limit yourselves to the Movie domain, unless you want to do extra-credit, see next.
2. **For graduates:** you **must** be able to answer queries in the other two domains. Graduate students must do more work for a 400-level class, as per University requirements.
3. **Undergraduate extra credit (50 points):** same as for graduate students.

2 SQL

SQL is a popular language used to submit queries to a relational database system. SQL is a *declarative* language, i.e., we tell the database engine *what* we want to get, without specifying any detail on *how* the system should actually perform the retrieval.

A separate file, **sqlite.pdf**, in this same folder provides some references to the tool side of SQL. Here, we introduce how we can use the SQL language to obtain information from a database.

The main data structure of a relational database is called *table*. Each row in a table (*record*) stores some information. Such information is divided into or more labeled *fields*. For example, in a hypothetical DB, we could have (we will use examples from our own DBs later in this document):

Person

id	first_name	last_name
1	John	Smith
2	Mary	Green
3	Susan	Brown
4	William	Smith

Phone_numbers

person	number
1	312 123 4567
2	773 321 7654
3	312 111 2233
4	312 555 0000

The process of answering a query consists of the retrieval of a set of records matching the required conditions. For example, to find the people whose last name is Smith, we can write the query:

```
SELECT *
FROM people
WHERE last_name = 'Smith';
```

The database engine will return two records ('*' means, return every field in the selected records):

id	first_name	last_name
1	John	Smith
4	William	Smith

If the information we are looking for is scattered in more than one table, we can *join* tables by setting up appropriate equality conditions on related fields. For example, to find the phone number of Susan Brown, we need to join the tables *people* and *phone_numbers* on the fields *id* of *people* and *person* of *phone_numbers*:

```
SELECT number
FROM people P
INNER JOIN phone_numbers PN ON P.id = PN.person
WHERE first_name = "Susan"
  AND last_name = "Brown";
```

The result is:

number
773 321 7654

In case of name conflicts, that can occur while joining tables having columns with the same name, we should make explicit reference to the specific columns, using the dotted notation (table.column). For example, the previous query could also be written as:

```
SELECT PN.number
FROM people P
INNER JOIN phone_numbers PN ON P.id = PN.person
WHERE P.first_name = "Susan"
  AND P.last_name = "Brown";
```

For further details on the SQL language, consult a book or one of the many online resources available, e.g. http://sqlzoo.net/.

3 Representative SQL queries

In this project, we will build SQL queries using the idea of compositional semantics. Instead of building a semantic representation for a sentence using first order logic, we will represent the semantics in the form of a SQL query. This approach is possible because of the declarative nature of the SQL language, that makes it somewhat similar to a logic representation. In practice, we will attach pieces of an appropriate SQL-semantic representation to the rules of the grammar that have been used to build the parse trees, such that the final SQL query can be built compositionally from these pieces. A possible implementation of this approach is illustrated in the following example.

Examples. Consider the sentence:

> Which Italian actress won the oscar in 1962?
> Answer: Sophia Loren

This sentence is a WH question. We want to retrieve the answer from the movie database, that consists of five tables; three are relevant here, Actor, Oscar, and Person (note: the numeric ids were copied from the DB. I make no guarantee that every single digit is accurate in the tables below)

Person

id	name	dob	pob
0000047	Sophia Loren	1934-09-20	Pozzuoli, Italy
0000138	Leonardo DiCaprio	1974-11-11	Hollywood, California, USA
0000473	Dianne Keaton	1946-01-05	Los Angeles, California, USA

Oscar

movie_id	person_id	type	year
0075686	0000473	best-actress	1978
0054749	0000047	best-actress	1962

Actor

actor_id	movie_id
0000047	0054749
0000138	0120338
0000138	0407887
0000473	0082979
0000473	0075686

The answer to the former question can be retrieved using this query:

```
Select P.name FROM Person P
INNER JOIN Oscar O ON P.id = person_id
INNER JOIN Actor A ON A.actor_id = person_id
WHERE P.pob LIKE "%Italy%" AND
    O.year = "1962";
```

This query returns *Sophia Loren*. This query can easily be run e.g. within SQLite Manager, by cutting and pasting it at the the "Enter SQLite commands" prompt. Note that technically this query is not fully specified,

because we did not specify that we are asking about the Oscar for best actress, implicit in the *actress* word in the query. Ie, if an Italian actor had also won an Oscar in 1962, or another Italian actress had won the oscar for best supporting actress, their names would also be returned.
A question like *Did Bigelow win an oscar for best director?* would result in the following SQL query

```
Select Count(*) FROM Oscar O
INNER JOIN  Person P ON person_id = P.id
WHERE P.name LIKE "%Bigelow%" AND
    O.type="BEST-DIRECTOR"
```

This query would return 1, hence, "yes". If we asked the same question but about Sophia Loren (i.e., `P.name LIKE "%Loren%"`), the query would return 0, hence, "no".

Note that the `LIKE ``%string%"` operator is used to allow flexibility in matching proper names (e.g. in the movie database, full names are provided but we will use only last names in the queries).

4 SQL queries and compositional semantics

How can we compositionally build such a query? We will sketch one way of doing so, which is based on a much simplified grammar and lexicon.
IMPORTANT NOTE. While you'll use a similarly simplified grammar and lexicon to what is presented below, you're welcome to devise a different, principled way of assembling the SQL query. The way we propose is not necessarily the best. *Principled* here means a method that is compositional in nature, and not too *ad hoc*.

Let's use as an example *Did Allen direct MightyAphrodite?* One simple grammar we can use follows **(this grammar is for illustrative purposes only!!! your grammar is the grammar used by the parser you chose, please see NOTE at the end of this section)**:

S $\rightarrow$ Aux NP VP
VP $\rightarrow$ Verb NP
NP $\rightarrow$ ProperNoun
ProperNoun $\rightarrow$ Allen | MightyAphrodite
Verb $\rightarrow$ direct
Aux $\rightarrow$ did

Now, let us attach a function to each rule of the grammar, as shown on the next page. Each attachment can be implemented as a function that can do three main things:

1. Return a string that will be used by another function in the lambda reduction mechanism.
2. Call another function belonging to a child node of the parse tree.
3. Add elements to data structures that will be used to compose the final query.

S → Aux NP VP

```
{[SELECT] += "count(*)"
VP.sem(NP.sem)}
```

VP → Verb NP

```
{Verb.sem(NP.sem)}
```

NP → ProperNoun

```
{'ProperNoun.sem'}
```

ProperNoun → Allen

```
{Allen}
```

ProperNoun → MightyAphrodite

```
{Aphrodite}
```

Verb → direct

```
(lambda :y :x)
{ [FROM] += " FROM Person P "
  [FROM] += " INNER JOIN Director D ON P.id = D.director_id "
  [FROM] += " INNER JOIN Movie M ON D.movie_id = M.id "
  [WHERE] += " P.name LIKE '%:x%' "
  [WHERE] += " M.name LIKE '%:y%' "}
```

In the example above, we have three data structures: [SELECT], [FROM], and [WHERE]. In practice, they are sets of strings. At the last step of semantic processing, we can transform the three data structures into plain strings using the following rules – the indices below (1, 2 etc), simply refer to members of the corresponding sets of strings.

```
SelectString := [SELECT].1, [SELECT].2, ... , [SELECT].n
FromString := [FROM].1, [FROM].2, ... , [FROM].n
WhereString := [WHERE].1 and [WHERE].2 and ... and [WHERE].n
```

Finally, we can build the final, well formed SQL query:

```
select SelectString
from FromString
where WhereString;
```

Important Notes:

1. The semantic structure provided for the verb "direct" implies that its arguments are Proper Nouns, given the WHERE constraints that include proper nouns directly, via `P.name` and `M.name`. This

would not work for, say, *Did a French director direct MightyAphrodite?* It would probably be better to add WHERE specifications to the Proper Noun semantics itself, even if at that point we wouldn't know to which entity (movie or director) the proper noun refers to. It is something you will need to explore, and that will need to be dealt with in the last steps of semantic processing we just described.

2. You do not have a grammar written declaratively as the one just described. This is due to using existing parsers such as those provided by OpenNLP or NLTK. Basically, each subtree $Node1 \rightarrow Subnode_1\ Subnode_2\ ...\ Subnode_n$ in the parse trees you obtain can be taken as denoting one such rule.

5 Sentences to be covered

See additional file **project-part2-questions.pdf**. For the bulk of the project, we will limit ourselves to the Movie domain. For graduate students or for undergraduate extra credit, queries in the other two domains will be considered as well.

6 Input/Output of your program

Your program should parse the sentence with the parser of your choice, take the resulting syntactic tree, produce the SQL query, route it to the SQLite engine with corresponding API in the programming language your choose, and process it against the DBs we provide. Your program should output: the original sentence, the SQL query and the answer you obtain from the DB. The final answer, produced by SQLite, does not need to be in natural language, but it has to be unambiguous and understandable. For example,
`SELECT count(*)`
will return a number n, where n can be 0 or > 0: you should transform 0 into 'no' and any other number into 'yes'.

7 What to hand in and how

You'll hand in

- all the code you develop, in a runnable form that your TA will specify;
- at least a half page of comments on how you developed your semantic attachments, issues you had to face, etc. Note that comments in the code (which should of course be plentiful) are not what we're looking for here: rather, we'd like you to step back from your code and reflect on how it achieves the intended goal
- sample output from your program, i.e., the original natural language query, the SQL query your program generates, and the answer you get from the DB. The sample output will include six queries of your choice, three Yes/No questions, and three Wh-questions.