CSCI 334 Principles of PL : Homework 8

Professor Dan Barowy

Due: November 8, 2018 at 10:00pm

ID: 3013644, 3023651

Q1. Project Proposal

"KCICK: the KCICK Consulting Interview CracKer"

(a) Introduction

Management consulting is a dream job for every Williams student. Well, at least that's what our *non-CS* friends say.

One of the most notorious aspects of consulting interviews are the Fermi problems. These are problems that require a fast and rough estimation of quantities that may be hard to measure physically. One example of a Fermi problem is "How many tennis balls could fit in an Olympic sized pool?" You might answer this by first considering how many tennis balls could fit in a dresser, then estimating how many dressers might fit in a lane, then multiplying that by the number of lanes in a pool.

But what if you are asked one of these questions and you have *absolutely no clue*? Is there any way to prepare for these questions, ace them, and fulfill your dreams of becoming a consultant?

Don't worry, KCICK is here for you.

KCICK will be a query language that finds and retrieves data from a database. The database will provide a means of storing important information and facts and performing various calculations before displaying the final result. In short, our project will be to design a user-friendly language that can be used to solve and answer Fermi problems.

(b) Design Principles

Our goal is to keep KCICK simple and direct (for our econ friends). To this end, the design of our language will mimic that of the metric system. The foundation of our language will be composed of "base units", which are our standard units of measurements that can be used to measure every other type of object in our language. The language will be used much like a "natural language", where the user should be able to supply input, typed out like a regular question, and the language will be able to recognize certain keywords and "vocab" and retrieve the measurements for these objects in the database.

(c) Examples

#1) How many **tennis balls** can fit in an **Olympic sized pool**?

- Parse and recognize key phrases "How many," and "fit in"
- Also recognize primitives "tennis ball" and "Olympic sized pool"
- Query relevant information from database

- Keyword ”fit” tells our language to query data regarding sizes
- Olympic pools = 10 lanes = 100 dressers = 50 tennis balls
- Answer: There are approximately 5000 tennis balls in an Olympic sized pool.

#2) How many **Big Macs** are sold in **New York City** every **day**?

- Parse and recognize key phrases ”How many,” and ”sold in”
- Also recognize primitives ”Big Macs” and ”New York City” and ”day”
- Query relevant information from database
- Keyword ”sold” tells our language to query data related to commerce/eceonomic statistics
- New York City = 50 McDonald’s = 300 Big Macs sold a day
- Answer: There are approximately 15000 Big Macs sold in New York City per day.

#3) How many **chalkboards** are there in **Williams College**?

- Parse and recognize ”How many,” and ”there in”
- Query relevant information from database
- Keyword ”there” tells our language to query data regarding known locations/buildings/establishments
- Williams College = 2 libraries and 50 classrooms = 30 chalkboards per library and 1 chalkboard per classroom
- Answer: There are approximately 110 chalkboards in Williams College.

(d) Language Concepts

KCICK is essentially a *smarter* version of Google search–its goal is to answer estimation questions that Google may not find the answer on the web. Since Fermi questions are almost always asked in regular forms, users only need to

- understand what they want to ask (e.g. how many basketballs can fit in a Boeing-747?)
- make sure they use certain keywords (syntax), like
  - ”How many”, ”How often”, etc.
  - ”sold in”, ”there in”, ”fit in”, ”
- include search terms that are objects defined in our language (e.g. basketball, pool, plane).

The key idea is to set a quantity you want to estimate with certain constraints. Assuming that the user follows the grammatical and vocabulary rules of KCICK syntax, the parser should be able to identify the keywords and objects, combine these together via calculations, and output an answer.

(e) Syntax

Following is an informal syntax for KCICK.

First we have the overarching non-terminal “question”, which will basically be the entire user input:

```
<question>::= <header> <object> <category> <object>
```

Now let’s look at the individual components. First, the “header” is the first indicator of what type of answer we are trying to estimate:

```
<header> ::= HowMany
           | HowOften
           | etc
```

HowMany is a type of header that consists of the string, ”How many”. This tells us that the user wants to answer a question regarding quantities, as opposed to frequencies in the case when the “header” is “How often.”

Next, is “object”:

```
<object>::= Main
          | Compare
```

Main and Compare are the two types of objects in KCICK. The Main object will be a string that represents the object of interest, and the Compare object will be the object being compared against.

```
Main ::= "(some string)"
Compare ::= "(some string)"
```

The third component is “category” which determines the type of the quantity we want to estimate :

```
<category> ::= FitIn
             | SoldIn
             | etc
```

FitIn and SoldIn are two types of categories. This syntax will determine the “tag” of the objects we will query. For example, the “category” of “sold in” will tell KCICK where to look in the database for the relevant information.
The syntax very much replicates that of standard English. For example, the Fermi question

”How many oranges can fit in a truck?”

will be broken down into

```
<header> = "How many"
<object> = "oranges"
<category = "fit in"
<compare> = "truck"
```

(f) Semantics

The *primitives* of our language are data, headers, objects, categories, and constraints. We will have a primitive that represents the smallest unit of measure. For example, if our primitive is the object, ”ping-pong ball”, everything else in our language will be measured in terms of quantity of ping-pong balls. Since

ping-pong balls are very small and not always the best way to measure certain objects given their shape, we might include additional primitives such as "Twix candy bars" or "Principles of Programming Languages textbooks". The data will simply be floats that represent the quantity, size, or frequency (depending on the tag) of the object. And finally, the question
The actions in this language will require fetching data and performing calculations.
Let's look at example #2) How many Big Macs are sold in New York City every day?

As said before, the "logic chain" includes

```
New York City -->  50 McDonald's --> 300 Big Macs
```

Fetching Data

This type of fetching could be done recursively, searching for relevant attributes until we arrive at our "destination" object.

Performing calculations

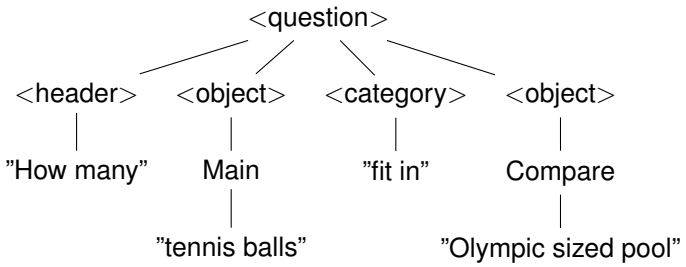
We will need to combine the data in a productive manner that helps us arrive at the answer. The calculations involved in this example would be multiplying 50 by 300, since each McDonald's sells 300 Big Macs and New York City has 50 McDonald's.

Representation

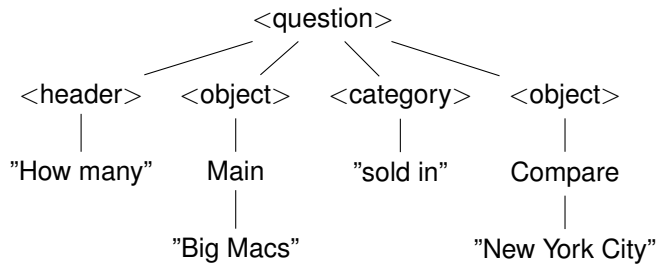
Our program is a database, which will represented by a table data structure. The rows will correspond to the name of the object, and the column will correspond to the tag that that object is associated with. So in the "fetching" action, the program would match the name of the object of interest (a string) with the row values, and match the tag of that object (a string) with the column values. This will allow the program to locate the "metric conversion" that we want and retrieve the relevant data, to be used in performing calculations.

Sample Abstract Syntax Trees

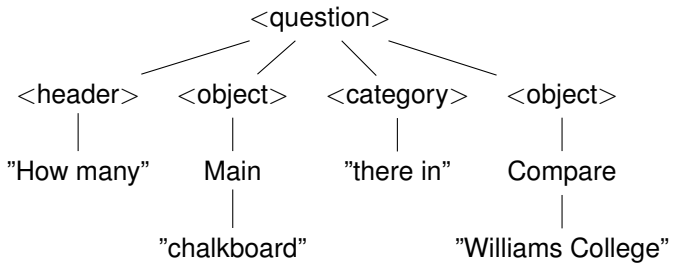
#1) How many **tennis balls** can fit in an **Olympic sized pool**?



#2) How many **Big Macs** are sold in **New York City**?



#3) How many **chalkboards** are there in **Williams College**?

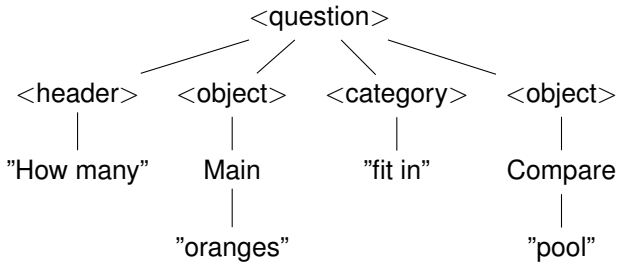


Evaluation (as of 11/15/18)

Evaluation begins when the user provides input in the form of a question, otherwise known as a "query". Then, the program will perform the two actions 1) fetching data and 2) performing calculations until the output is reached. The output will likely be a quantity of type float that is returned to the user as the answer to their Fermi question.

For now, we can only parse objects of one word length. Let's consider the question: "How many oranges can fit in a pool?"

Our parser returns AST:



First, we would examine the category, "fit in", which tells us where in our database to search. Our database will ultimately be a map of strings to maps (and these "interior" maps will be from strings to floats). So it will be of type

```
Map<string, Map <string, float>>
```

The category string will be the key, and lead us to another map with various objects as keys and floats as values. For this week's implementation, our database is just a map from strings to floats, since we only have one category. Then, we find the value in the map by using the Main object and Compare object as keys, divide their values, and return the answer as 12.5 oranges!