CSCI 334 Principles of PL : Homework 10

Professor Dan Barowy

Due: November 29, 2018 at 10:00pm **ID: 3013644, 3023651**

"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 standard units of measurements that can be used to measure every 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. We also want our program to be dynamic, so users can input data and expand the database at anytime.

(c) Examples

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

- run with ”dotnet run ⟨filename⟩”. The file name is an optional parameter that lets the user supply a database. Otherwise, the program will use the default database, ”data.txt”.
- Parse and recognize key phrases
- Query relevant information from database

- tennis balls: 10.0 cubic inches and Olympic sized pool: 150650325.21 cubic inches
- Answer: 15065032.521 tennis balls.

#2) How many **big dangerous huge cannons** can fit in an **Olympic sized pool**?

- ”dotnet run ⟨filename⟩”
- Parse and recognize key phrases
- Query relevant information from database
- big dangerous huge cannons: 80.0 cubic inches and Olympic sized pool: 150650325.21 cubic inches
- Answer: 1883219.065125 big dangerous huge cannons

#3) How many **hamburgers** are sold in **New York City** every day?

- ”dotnet run ⟨filename⟩”
- Parse and recognize key phrases
- Query relevant information from database
- hamburgers: 0.1 consumed per person every day and New York City: 8623000.0 people
- Answer: 862300 hamburgers

(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 “query”, which will basically be the entire user input:

```
<query>::= <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> ::= How many
```

‘How many’ 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)*(some string)"
```

Notice that Compare is a defined to as a tuple of strings, the latter to represent the unit of the comparing object. The second string is not strictly required, in that it can be an empty string; but it can be used as in the case ”How many hamburgers are sold in New York City every day?” where ”every day” is the second string.

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

```
<category> ::= fit in
             | sold in
```

Fit in and Sold in 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

Semantics of each language element

(1) Query

The abstract syntax of Query is `(Header, Object, Category, Object)`. The first Object is the Main object and the latter is a Compare object. This element represents the entire question given by the user, consisting of only the meaningful parts that are required for computing the answer. Inputting a valid query evaluates each `Header`, `Main`, `Category` and `Compare` and returns an appropriate answer of type float.

(2) Header

The abstract syntax of header is `Header of string`. This element, along with Category, gives information on what type of data the query is asking about.

(3) Main

The abstract syntax of Main is a `Main of string`. This element represents the main object of the query, for example ”tennis balls” or ”hamburgers”.

(4) Compare

The abstract syntax of Compare is a `Compare of string * string`. This element represents the object to be compared against, for example ”swimming pool” or ”New York City every week”. The second string is the unit of the item of the first string; the second part can be omitted, in which case the program will parse as an empty string.

(5) Category

The abstract syntax of Category is a `Category of string`. This element represents what type of information the Query is asking for. For example, the Category ”fit in” will signify that the question about comparing the volumes of two objects.

In short, Query is a meaningful deconstruction of a typical (but restricted) question in English.

Operations

The single operation that KCICK supports is to enter a question in the form of a Query; KCICK will return an answer in floats or will tell the user it cannot solve the problem.

Answering questions further requires two actions: fetching data and performing calculations.

Let’s look at example #2) How many Big Macs are sold in New York City every day?

```
Big Macs sold per capita (per day) --> 0.05

Population of New York City --> 8623000.0
```

Fetching Data

Category gives information on which Map in the database the evaluation should be based on. For example, a Category of ”sold in” directs the interpretor to reference the Map that contains information of per capita quantities of consumer goods and population statistics. Then the interpretor fetches the numeric data corresponding to the Main and Compare objects.

Performing calculations

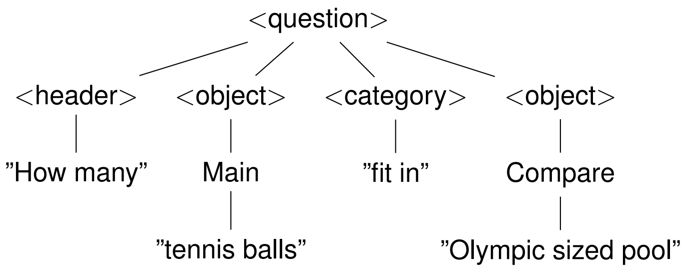
We will need to combine the data in a productive manner that helps us arrive at the answer. The calculations involved in this example would be multiplying 0.05 by 8623000.0. Furthermore,

Representation

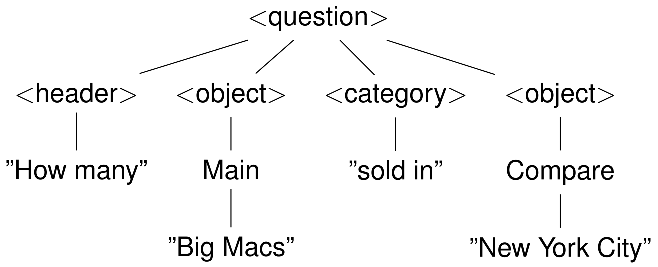
Our program is a database, which will represented by a table data structure. The rows will correspond to the name of the object, and the column will correspond to the tag that that object is associated with. So in the ”fetching” action, the program would match the name of the object of interest (a string) 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

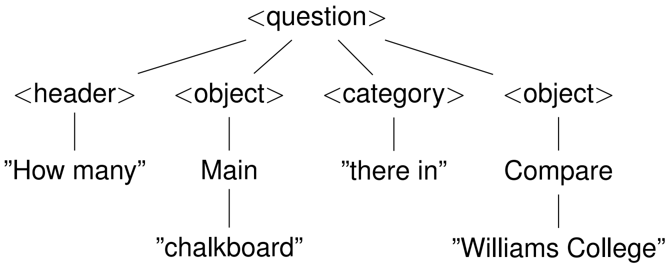
#1) How many **tennis balls** can fit in an **Olympic sized pool**?



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



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

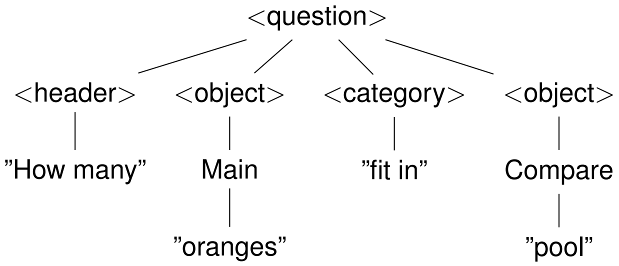


Evaluation (as of 11/29/18)

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.

Let's consider the question: "How many t can fit in a pool?"

Our parser returns the 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 example, the categor "fit in" leads the interpretor to get to the map related to volumetric data. 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 18831290.65125 oranges! (given the pool is Olympic sized)

The interpretor evaluates the numeric data in different ways for different categories. For example, if the category is "sold in" the float values will be multiplied instead of being divided to give an answer to questions like "How many Big Macs are sold in New York City every day?"

(g) Remaining Work (as of 11/29/18

We have updated the program to have a REPL with information on all the actions that the user can take. KCICK now reads in text files (default as data.txt) to initialize the database. The user can add entries to the database, which will overwrite the data.txt file. Once the user restarts KCICK, the given data input can be used to answer new questions.

Here are some ideas to we will be pursuing in the near future:

- more categories (”there in”)
- more headers (”how often”)
- make program case insensitive to input