CS109B PROJECT: A "NATURAL LANGUAGE" QUERY SYSTEM

Due Friday, May 22

The purpose of this lab is to explore

1. Some data structures for database queries, and
2. The power of some surprisingly simple techniques for adding "intelligence" to systems — up to a point; we'll discuss this matter at the end.

The Data

The data will consist of files `hg1` through `hg7`, `mg`, `pg`, and `fg`, for homeworks 1 through 7, the midterm, project, and final. This data will be taken from CS109A grade files, with people's names replaced by (random) two-upper-case-letter codes. The files will be placed in the class directory `/usr/class/cs109b`. Each file will consist of a sequence of lines, and on each line will be two letters, a tab, and an integer. For example, the line

```
AB	123
```

says that student `AB` got grade 123 in whatever assignment is represented by the file.

Queries

This project is open-ended, but we'd like you to succeed by being clever, not by writing a lot of code. The goal is to be able to handle queries expressed in English sentences. For simplicity, we'll assume no punctuation or capitalization, except in student names. Numbers will always be represented by digits, e.g., "10" rather than "ten." We're not going to limit the nature of queries, except to say that they must be answerable from the data described above.

Here are some examples of queries. It is not essential that you be able to handle all of these, nor will queries necessarily be limited to these types.

```
what was student ABs grade in homework 3
student AB what grade on homework 3 did he or she get
what is the sum of all homework grades for student AB
how high was the average score on homeworks for AB
who got 100 on the midterm
which students scored better on the project than on homework 7
did AB do better than CD on the midterm
who did the best on homework 7
```

A Possible Strategy

The problem of understanding language is far too hard. What we can do is develop some simple strategies for *guessing* what the query means. For example, in the first query above, the words `what` and `grade` might suggest we need to look up a grade for some student in some file. The file is suggested by the word `homework` and the number 3, while the student is suggested by `ABs`. (Remember that capitals clearly indicate a student name.)

Here is a possible architecture for your program. Before accepting any queries, your program reads the ten data files and stores them in suitable data structures. It is acceptable if you preprocess the given data into one file and read that, or even if you somehow incorporate the data into your program. (In real life we could never get away with that, because the data changes.)

1. Read a query, break it into words.
2. Scan the query to recognize certain "syntactic categories," such as <student name>.†
3. For each of a list of *rules* (described below), examine the query to see whether it matches the *pattern* of the rule (also described below), and if so, perform the action associated with the rule. The *action* would be some sort of inference about what the query was saying, and we'll discuss that below as well.
4. Based on the inferences made from the rules, determine what the query was most likely saying, in terms of — for example — relational algebra operators.
5. Execute the guessed query from (4), and print the answer.

Patterns, Rules, and Actions

A *pattern* is some condition about the words and syntactic categories in a query. For example, the word `what` followed by the word `grade`, perhaps separated by other words, is a pattern. The syntactic category <studentName> by itself is a pattern.

An *action* is something that reflects a conclusion drawn because we have seen a pattern.

1. As a simple example, if we see the pattern <studentName>, i.e., any example of two capital letters, we might take the action of setting the variable `StudentName` to the name of the student seen.
2. For a more complex example, if we see the pattern `what...grade`, we could set the variable `Grade` to `"*"`, meaning that the grade is unknown and must be discovered by the query. However, this strategy is a bit too naive. If we consider the first sample query, we see both this pattern and the pattern `what...student`. Yet the student is specified and the grade is not. How do we tell which is specified??
3. Another example: if we see the pattern consisting of the word `average`, then we set the variable `Average` to `true`, meaning that we must take the average over whatever values we later decide the query is looking for.

A *rule* is a pair consisting of a pattern and an action to take when we see that pattern.

Inferences

Once we have applied all the rules and taken whatever actions are dictated by the rules whose patterns match the query, we need to discover what the query is. We might examine

† The idea of syntactic categories is described in Chapter 11. We do not suggest you attempt a full context-free parser for English. It is too complicated and context-free grammars are not a very good model of natural language anyway. However, there may be some ideas from Chapters 10 and 11 that you will find useful.

the collection of variables and make decisions such as "if `StudentName` holds the name of student S, variable `Assignment` holds the value A, and `Grade` holds value `"*"`, then the query is $\pi_{Grade}(\sigma_{Name=S}(A))$. Here, A is intended to stand for the file holding the data for assignment A. That is, look up in the file A the line for the student S, and print the second, or Grade, component.

Drivers and High-Level Programming

One way to implement rules is to write a piece of code for each one, and execute them in turn. Often, a better way is to develop a notation encoding rules and write a *driver* program that looks at a rule and executes it. For example, we might represent patterns by a notation similar to that used by the UNIX shell to represent file names, where characters represent themselves (but you need words instead of characters), and * represents "anything." However, you may wish instead to allow sets of words, rather than lists, i.e., words appearing in any order as a pattern. Anyway, it's up to you what notation you adopt. Actions could be represented by a variable and a value, for example. Again, we're not suggesting that this notation is adequate; only you can decide on the right notation for you.

However, once you have a notation and a driver, you get two benefits.

1. Programming becomes much more succinct. You can write a pattern with many fewer keystrokes than you can write the code to search for it. This factor is important, because the more rules you use, the more precise your interpretations of queries can be.

2. Programming becomes much more natural, and (we hope) less error-prone. The description of a pattern will be much closer to what the pattern "looks like" than would be some code to implement a search for the pattern.

Language Issues

You may write in ML or Pascal; we'll consider other languages if the TA's are willing to grade such programs. If you write in ML, you need to know a few carefully guarded secrets.

First, there really are arrays in ML. We can create an array A of n elements (numbered 0 through $n-1$), each element initialized to value x by

```
val A = array(n,x);
```

For example, an array of headers for a hash table with 100 buckets could be created by

```
val headers = array(100,[]);
```

Initially, each bucket would be the empty list.

The value of the ith element of array A is produced by the expression `A sub i`. To store a value v in the ith element of array we use the expression `update(A,i,v)`. For example, if A is the hash table described above, h is the hash function used, and `insert(x,L)` inserts element x on list L, then

```
fun hInsert(A,x) =
    let
        val bucket = h(x);
        val L = A sub bucket (* L is the old list for bucket h(x) *)
    in
        update(A,bucket,insert(x,L))
    end;
```

inserts x into the proper bucket of hash table A.

Second, we need to do character input in ML. There is a mechanism in ML, although we did not cover it previously. We'll provide a file `io` in the class directory that will give you a function `read` to read a file and turn it into a list of characters. To make your life simpler, we'll also provide in this file a function `group` that turns a list of characters into a list of words by observing where the blanks and newlines are. Finally, comments in file `io` will give you some clues on how to do your own input if you like.

What to Hand In

In addition to an appropriately commented listing of your code, we need an informal description of

1. The data structures used to store the information.
2. What did you assume about the kinds of queries you could handle? How did you implement these queries using the data structures from (1)?
3. What algorithms and strategies did you use to deduce what query was being asked by a sentence?
4. What are your basic concepts, e.g., did you try to identify the type of question being asked by finding "question words"? What parts of your program did you do with a driver and a notation for patterns, rules, etc.? What notation did you use?
5. Examples of the sorts of things that your program can handle and things that it cannot. You should show us your program's response on those of the sample queries it can handle, but we're more interested in general classes of queries. For example, can you handle queries that use concepts like sum or average?

Grading

We'll allocate points approximately as follows.

1. 50%: Getting something working, i.e., a program that doesn't bomb out on well-formed data and answers at least a few queries correctly.
2. 15%: Using data structures that are reasonably efficient. Because the data is not large, you may not feel the effect of inefficient data structures, but you should know enough from 109A to understand the big-oh complexity of the structures and operations you select.

3. 15%: Using appropriate architecture. Writing drivers for parts such as pattern search, rules, and possibly the interpretation phases.
4. 20%: Scope of your project. The greater variety of queries you are able to field properly the better you will do in this category.

We'll arrange a time for demonstrations of projects, but the demo will not be required and will not influence your grade.

Some Theological Comments

Many see this sort of exercise as "Artificial Intelligence," although I prefer to see it as a matter of design and implementation of algorithms and data structures, in order to accomplish something that looks hard. The fact that the goal and the question of what is a "satisfactory" program is more vague than if I had — say — assigned you to write a program that sorts integers, is sometimes said to give the problem an "AI flavor." However, in all sorts of real-life programming, the criteria for success are vaguer than we would like.

As you work this problem, you will probably find that you can do surprisingly well with little mechanism. This is typical of many styles of programming, such as "neural nets," where on selected problems the mechanism astounds with its ability. However, experience has shown that when you try to push a simple mechanism it doesn't budge as easily as you might expect. You will find that a few word-patterns give you a lot of correct answers without the system obtaining any deep understanding of the sentence. If you try to perfect your program so it always "understands" the query, you find it extremely hard to make progress beyond the first steps. Understanding language is most likely a fundamentally hard problem, and progress against it will continue to be painfully slow.

Teams

You should work in teams of two or three (no more). See handout 1 for conditions regarding teams. If you need a partner or partners, you might advertise on the class newsgroup.

Final Word

This is a challenging problem. I encourage you to form teams early and start thinking about the architecture. You are welcome to bounce ideas off me and the TA's by mailing to cs109q@theory. We'd much rather discuss things in April and give you some hints than find that come May you are using a strategy that can't be successful.