The Story Project

CS343
Fall 2006

---

1 Introduction

The Story Project is about one of the central themes of artificial intelligence: Representing knowledge and reasoning with it. It will give you a chance to see how a real-world "Knowledge Machine" works, and to explore some of the most important questions in AI:

- How can we represent real-world knowledge so that a computer can reason with it?
- What knowledge is necessary for a given problem?
- How can we make a knowledge base perform well on not just one but a wide set of problems?

If nothing else, you will see how hard the problem of knowledge representation really is, and what some of the problems are that stand between us and true AI. Throughout this project, you will feel a constant tension between:

- Choosing a representation that is simple and just powerful enough to answer the current questions. This will make your job easier, but your knowledge base will end up being "brittle", i.e. it will only work on a very narrow set of problems.
- Choosing a representation that is as general as possible. This will make your knowledge base more flexible and extensible, but it's also much harder to do.

Obviously, you will have to decide at some point that your knowledge base is general enough (whatever that means). You will have to make reasonable compromises between simplicity and generality, and you may find that you have to change your mind as you proceed. Also, you will probably find that most of the time, general solutions can be simpler and more elegant than simply hard-coding the answers to the given questions. There is no one right answer to the representation question. Each part of the project will build on the earlier ones. So, at the end, you will have built a single knowledge base that should enable you to answer any of the questions that you have seen.

2 Using KM

We will use KM ("The Knowledge Machine") as our knowledge representation and reasoning engine. KM is a powerful frame-based language developed at UT by Bruce Porter's Knowledge Systems group. To run KM on the department machines you:

1. Download KM (file `km-2-0-39.lisp`) from http://www.cs.utexas.edu/users/mfkb/RKF/km.html. You will also want to familiarize yourself with the KM manual (it will reduce the amount of time spent banging your head on the keyboard later) and print the quick reference sheet available on that site.
2. To make life easy, rename `km-2-0-39.lisp` to `km.lisp`.

3. Start Lisp by typing `acl`.

4. Load KM by typing `(load ‘‘km.lisp’’)` in Lisp.

5. Start KM by typing `(km)` in the Lisp window. You should see a `KM>` prompt now.

Now you are running KM, open your favorite text editor to write your code. Save the code to a file (say `test.km`) and load it into KM by typing `(load-kb ‘‘test.km’’)`.

A few other things about KM:

- If there's an error, KM will start printing every little thing it does, which can get annoying. Turn it off with (untrace).
- The grading program may add instances before loading your knowledge base, make sure that you don't have the command `(reset-kb)` in the files that you turn in. Note that if you use the `save-kb` command then it automatically puts a `reset-kb` at the top of your file, in this case simply delete the `reset-kb` line.
- If you've made changes to your knowledge base, and want to reload it, make sure you use `(reload-kb ‘‘your-filename’’)`, not `(load-kb ‘‘your-filename’’)`.
- Instead of loading a knowledge base from a file, you can just type KM code directly in the KM window. The best strategy is probably to load your knowledge base from a file and enter the questions in the KM window. You can also include questions in the file though.
- KM will run much faster if you compile it. This only needs to be done once, to do this type

  ```
  acl
  (load ‘‘km.lisp’’)
  (compile-file ‘‘km.lisp’’)
  ```

  Then when you load KM instead of typing `(load ‘‘km.lisp’’)` you will type `(load ‘‘km.fasl’’)`.
- To exit acl type `:exit`.
- If you are an emacs fanatico you can run KM in emacs, follow the instructions at http://www.cs.utexas.edu/ mooney/cs351/allegro6-emacs.html (but use acl70). Then in emacs split the window into two, in one start Lisp by typing `Alt-x fi:common-lisp`.

3 The Story Project

Each part of the project will consist of a simple story and a list of questions that can be answered using knowledge from the story. Your job, for each part of the project, will be to encode the facts that are explicitly described in the story, and any additional knowledge that is required to understand the story. When you're done, your knowledge base will not only be able to answer the given questions. If you have done a good job, it will also be able to answer many other questions about the knowledge in the story. Your grade will be based on

- The answers your knowledge base gives to the questions you have already seen. This should be easy if you use the same names for classes and slots as in the questions, and if you test your code on the questions.

- The answers your knowledge base gives to a set of questions you haven't seen before. The questions will use only the names for classes and slots from the known questions, so don't worry about names too much. However, the new questions will test how well you encoded the given knowledge, and how general your knowledge base is.
- A few paragraphs on how you encoded the knowledge in the story. You don't have to describe every little feature of your knowledge base. Anything that proves that you yourself have worked on solving the problems is fine. Write how you solved a certain problem, and what the alternatives were. Write how KM was driving you crazy and why. Write how the English story was ambiguous and what that meant for your code. Or simply describe your knowledge base. Don't write more than about half a printed page.

3.1 An Example Story

Here's an example for a simple story and a set of questions: Story: All cars have four wheels. Fred drives a Junk-O-Matic. All Junk-O-Matics break down all the time. Fred's Car is red.

Questions:

What color is Fred's car?

```
KM> (the color of (the Car owns of *Fred))
(*Red)
```

How many wheels does Fred's Junk-O-Matic have?

```
KM> (the wheels of (the Junk-O-Matic owns of *Fred))
(4)
```

Note that the questions are given both in English and KM, so they give you hints on the kinds of frames and slots you need. It's obvious that in order to answer the questions, you will need classes called Car and Junk-O-Matic. It's also clear that you'll need two instances called *Fred and *Red, and slots called color, owns and wheels. If you look at the story, you'll notice that there's some knowledge that is just too obvious to be mentioned. In this case, it's "A Junk-O-Matic is a kind of car". Without that knowledge, the system would not be able to conclude that Fred's Junk-O-Matic has four wheels (because it's a Junk-O-Matic, which is a kind of car, which means it has four wheels like all the other cars), so we need to include it in the knowledge base. The story above might be encoded as:

"Every car has four wheels"

```
(every Car has
(wheels (4)))
```

"Fred has a Junk-O-Matic"

```
(*Fred has
(owns ((a Junk-O-Matic))))
```

"A Junk-O-Matic is a kind of car"

```
(Junk-O-Matic has
(superclasses (Car)))
```

"Junk-O-Matics break down all the time"

```
(every Junk-O-Matic has
(breaks-down (t)))
```

"Freds car is red."

```
((the Car owns of *Fred) has
(color (*Red)))
```

This is not the only way to encode the knowledge, and it's not perfect. For example, it ignores the phrase "all the time", and it might have been better to have a "parts" slot in the Car frame, because then we could add other parts that every car has, like an engine or a steering wheel. However, the way the questions were encoded took some choices away from us. On the other hand, the code can answer a lot of other questions. For example, we could ask for all people who own Junk-O-Matics:

```
KM> (the owns-of of (every Junk-O-Matic))
(*Fred)
```

Or, we could ask for all the cars Fred owns:

```
KM> (the Car owns of *Fred)
(_Junk-O-Matic1)
```

We could also create a Junk-O-Matic and see if it breaks down all the time:

```
KM> (the breaks-down of (a Junk-O-Matic))
(t)
```

A knowledge base that doesn't exactly encode the knowledge in the story would probably give wrong answers. Say instead of encoding that every Junk-O-Matic breaks down, we had encoded that only Fred's Junk-O-Matic breaks down:

```
((the Car owns of *Fred) has
(color (*Red))
(breaks-down (t)))
```

In that case, the knowledge base would have given a wrong answer. Or say we forgot to include the fact that a Junk-O-Matic is a kind of car. Then, again, the knowledge base would have no way of knowing that Fred's Junk-O-Matic is a car and therefore has four wheels.

4 Submitting

You can work in teams, and it's no problem if everybody in a team submits the same code. But everybody has to write his or her own description of the knowledge base or the problems you encountered while developing it. You don't need to include questions in the code. Please do include your name(s) in all files though. Please call the files `storyX.km` and `storyX.txt` (where $X \in \{1, 2, 3\}$ corresponds to the three assignments) to make them easier to find. To turn in your project, log on to a cs machine and type

```
turnin --submit jmugan storyX storyX.km storyX.txt
```

You can change the files by submitting again, and you can check if everything worked using

```
turnin --list jmugan storyX
```

Have fun!