Factotum

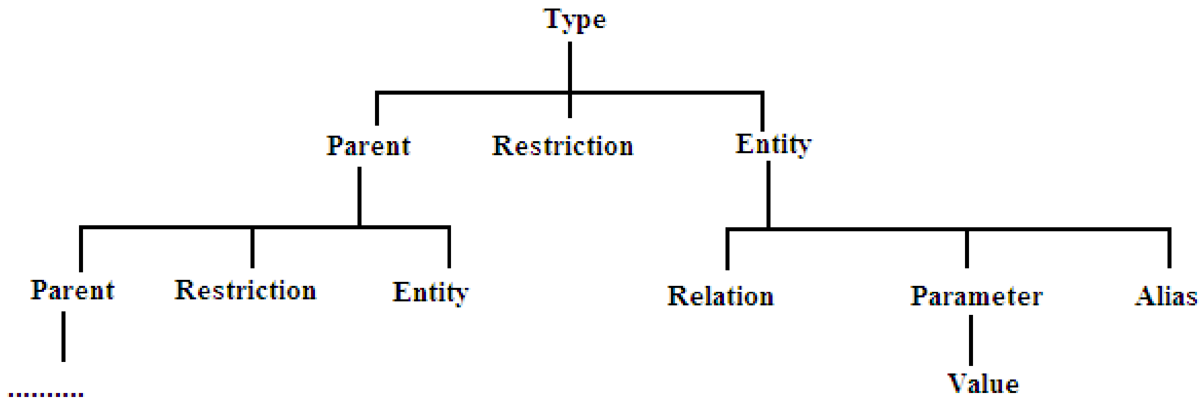
Andrew Shin

Introduction

Factotum is an open source project originally conceived by Robert Uzgalis at UCLA. Its primary goal is to aid the study of humanity, religion, linguistics and so forth, which deal with a large amount of facts that need to be organized, so that the user can easily validate a topic of questions based on the known facts. Claire Abu-Hakima has written the code that corresponds to parsing and lexing aspects of Factotum under supervision of Paul Eggert, and I would like to extend it, based on her work, with emphasis on checking and validating aspects. While Abu-Hakima's work used linguistic data collected from Wikipedia to test it out, I did not use her data due to different marker usage and several other issues. Please refer to the original Factotum repository for details on parsing and lexing, how she made use of Wikipedia language data (www.github.com/claireabu/Factotum). New codes were written in Python 2.6.1. I am working on converting the entire project into Python 3, but there are issues such as indentation from the original code, so it is yet to be uploaded.

Dictionary

My current version reads the information from __.v file, which is generated by Claire Abu-Hakima's original code, based on __.f file, which consist of facts entered by the user, following certain grammar rules (the original code is slightly modified by me to have the text output easier to process). Collecting information and putting it into hierarchical dictionary is done by readVocab() and collectRelation(). This may be optimized in future so that it reads input by importing the original code, rather than off the text file. The text file outputs the vocabulary information in the following format, through which my Factotum collects the information (one can easily modify the code so that it collects information by importing code rather than scraping off from text file):



On the top-level of dictionary is type. Type has a parent type, restrictions that apply to the type, and the entities that belong to the type. Since parent type is also a type, it has the same subfields as type. Also, the entities that belong to a type also belong to its parent type. However, restrictions only apply to the current type, not to their parent type. Consider a type Chinese whose parent type is Asian. Then, a restriction “born in China” must only apply to Chinese, not to its parent type Asian. When a user wants to add a new entity of a type, then Factotum asks the user whether the new entity satisfies each of the restrictions applied to type. If passed for all restrictions, then new entity is added to the type. Each entity consists of the list of aliases, relations, and parameters. Relations include anything that describes entity that does not include numeric values. Note that attributes (for example, “__ is male/female,”) are also included in relations. Parameters are anything that describes an entity that includes numerical values. However, in the parameter key field, the actual numerical values are replaced by (), so that the values can be stored as values for each key (that is, parameter).

Grammar

The following grammar expressions are enforced for newly entered facts. The same rule applies when the user needs to modify the __.f file.

```
subject predicate
entity [type]
[type] >>parent type
[type] #restriction
```

I have not set up the marker and expression for aliases yet.

Code

As described above, readVocab() reads lines from ____.v file generated by mkvocab.py upon ____.f, and it puts type, entity, parent, and restrictions information into hierarchical dictionary. Since it is scraping off from text file, it simply looks at each line and checks whether the line contains a clause that indicates the presence of ‘fact’ elements. If so, it puts the element into corresponding part of hierarchical dictionary. I have repeatedly stated that this needs to be optimized by importing original code.

readVocab() collects all elements of a fact except relation. Because relations are currently handled without any marker to indicate them, they are handled by a separate function collectRelation(). It works in a practically identical way as readVocab().

updateDict() deals with the property that entities of one type should also belong to its parent type, which I briefly mentioned. Iterating through type dictionary, if its parent key is not empty, all the entities of the type are copied into the entity field of the parent type. Note that relations, parameters, and aliases for the type are not copied.

getTypeof() simply returns the type of an argument entity.

hasNumeric() and returnNumeric() are made to differentiate parameters from relations. hasNumeric() is a boolean function that returns true if an argument contains a numeric in it. returnNumeric() retuns the numeric values within the argument as a list. It returns a list instead of a numeric value because there might be more than one numeric value in the argument. The numeric values in the list are later replaced by () in the parameter so that we can store parameter and corresponding value separately.

validateNewFact() is where the user enters new facts and gets a check whether it is a valid fact. First, the user enters a new fact according to the enforced grammar rule. In current grammar rule, the first token of the input can start with either a name of subject or a type, indicated by []. So we first check whether the first token is a subject or a type by checking the presence of []. If it is a subject, then we check whether the rest of the input has to do with type information, such as type, restriction, or parent. All of them are distinguished by distinct markers, so we check for the presence of each marker. If none of the markers is present, it means the input is relation (or parameter). We use the hasNumeric() function from above to check whether it is relation or parameter. If parameter, we replace the numeric values with () and puts the values separately for corresponding parameter key.

If we do find a marker, then we know by looking at the marker what element it is, so we get to checking part. If the subject is entered with a type and the subject is not in our fact dictionary, then we ask if the subject satisfies the type restriction. If it passes all restrictions, then it is validated and added to the dictionary. Otherwise, it is not validated. Checking the parent and restrictions for a type is straightforward, done just by looking at the dictionary’s corresponding key.

Example

Suppose we have the following predefined facts:

Andrew [Korean]
Ichiro [Japanese]
[Korean] >>Asian
[Japanese] >>Asian
[Korean] #isBornInKorea
[Japanese] #isBornInJapan
Andrew is male
Ichiro plays baseball
Ichiro hit 15 home runs in 2009

If any of the predefined facts is entered as a new fact, then it gets validated as it should. If, in a new fact, a new entity is said to be of one of the predefined types, the program asks whether the new entity qualifies for restrictions of that type. If the answer is yes, then the new entity is added to the type's entity list. If not, then it is not validated. However, if a new fact involves a new type that is not in the predefined facts, then we cannot check whether the statement is valid. It currently just gives dictionary key error, and I will fix it to give a proper warning message. If a new type were to be added to our facts, then the user should modify the ____.f file and make new vocabularies out of it.

Andrew [Korean] → validated
David [Korean] → Is David born in Korea? (y/n) → validated → new entity list
Andrew [Chinese] → keyerror

Likewise, pre-defined parent relations between types are validated, but if child type is not defined, it gives keyerror. If parent type is not defined previously but the child type is, then we know the statement is not right. Relations work in the same manner.

[Korean] >>Asian → validated
[Japanese] >>Japanese → not validated
[Korean]>>European → not validated
[Chinese] >> Asian → keyerror
Ichiro plays baseball → validated
Ichiro hit 6 home runs in 2009 → not validated