An Investigation of the use of Factotum in creating a resource of Language Information

Claire Abu-Hakima

2011

Table of Contents:

I. A Brief Introduction

The job of a researcher requires one to gather all possible relevant data in hopes of supporting an already stated hypothesis or discovering a new one. However, the task of organizing and sifting through all the data is often tedious and identifying unknown patterns is not always obvious.

Facotum is a software tool for organizing research data. The original program, developed by Robert Uzgalis and described in "Factotum 90: A Software Assisted Method to Describe and Validate Data," allows researchers to uniformly enter their data, and help them infer patterns and connections between given facts, creating a sort of taxonomy for the different data. Additionally, Factotum's format for entering data allowing it to be language independent, so that facts in any natural language may be used with Factotum.

The forced formalism of the data through the use of Factotum requires researchers to further think critically about their work and to come closer to realizing what it is they wish to demonstrate through this precision as well as what the data is actually showing them.

The original version of Factotum was written in the C programming language, and recently, Factotum has been translated into Python, with some parts even being rewritten and along with a new interesting idea for its use. I've rewritten the parser for the vocabulary of the data (a set of rules designating the format of the data), which makes sure that the format of the vocabulary rules themselves is correct, as well as the fact checker, which uses the generated vocabulary to check if the entered facts to adhere to those specified formats. Both are in Python 2.7 and are contained in `predpar.py` and `fcheck.py` respectively. Additionally, I created a specific dataset for Factotum that is in fact the main driving force of the project.

As a student of Linguistics, I've always found the lack of a single unified and legitimate resource or database for the world's languages to be surprising and, at times, frustrating. And so not only would Factotum provide a great format for organizing (mainly meta-) data on the world's languages, but could also be used to demonstrate the connections between different language families, unforeseen ties to be further explored, as well as gaps in knowledge. A Factotum database with collaborators from different experts on languages would be not only an excellent source for researchers just seeking essential meta data, but also a way to preserve and easily spread this knowledge.

To demonstrate the feasibility of this approach, I created a rudimentary data set made up of facts pulled from articles and mainly the information boxes on the Wikipedia language pages to demonstrate the potential that Factotum holds. In this paper I hope to explain what I learned from this collection of data, as well as what I learned was possible to accomplish with Factotum itself.

II. The Problem and the Objective

In this section I will now discuss in further detail what exactly it is I am trying to do with Factotum as well as what exactly it is that I set out to accomplish and demonstrate with this project.

The main problem I strive to solve is the frustrating lack of a single, qualified source or data base containing at the minimum meta-data or a high level over view of all (or most) of the world's languages. When doing research, I am often forced to jump around between different books, journals, and papers, with competing theories, differing terminologies, and hardly ever an explicit answer to my question.

With this project I hope to show how Factotum could help with the creation of a unified source, which if filled with information provided by the top authorities in each respective language field, could not only prove to be an incredibly useful resource filled with the desired data, but also allow advances in the field. New and undiscovered connections between languages or within the language itself may appear, providing an opportunity for an expert to explain this presence, or exhibiting a gap in the knowledge that researchers in the future may attempt to clarify.

My goal with this project is to show that even with a 'naïve' (not expert) dataset of facts on the world's languages provided by Wikipedia, a Factotum language source has great potential and would allow for: the discovery of errors in the recording of information about languages; the forced formalization of linguistic theories, so that nuances of are not lost in the description and are better articulated so that others may better understand these ideas as well as allowing for better support of the theory if it is now clearer; the ability to easily see what information is already available, recorded, and (presumably) thoroughly researched, all in one place—no jumping around as before, but we still maintain the ability to track where the information came from as Factotum allows for the citing of sources right next to the data; the presence of new connections that may lead to new discoveries or force researchers to reexamine and scrutinize their theories to account for it.

The potential for a linguistic database using Factotum is quite great, and I hope to demonstrate this on a smaller scale by grabbing linguistic facts from Wikipedia language pages, forcing formalization on them by putting them into Factotum data format, analyzing these facts and producing a set of grammar rules for them to follow, checking to make sure these grammar rules adhere to the format of grammar rules, and then going back and checking the facts against these rules, thus demonstrating that the facts are well formed, formal, and adhere to unified format. This last bit while not an obvious step in the greater use of a Factotum data base, is an important step, assuring that all the data follows a (relatively) unified form, and checking this is what forces all contributors to have the same formalized format of facts, making it easier for the user to navigate and comprehend the presented.

III. The Dataset

One of the major components of this project is the dataset I elected to use, since everything revolves around the data in Factotum and I chose to create my dataset from the language pages from Wikipedia. Here I will explain my decision to use Wikipedia facts, what select bits of data I chose to scrape off the Wikipedia language pages, and what is significant about the use of Wikipedia for a language database.

A. Why Wikipedia?

When I mentioned earlier my frustration, all that I was searching for was a quick reference to a language so that I could get an overall idea of what sort of language it was, the language family, and some other bits of information concerning either syntax or phonetics and related languages. However when searching for an academically sound source, I often would have to leaf through papers and papers to find the one little bit of information I was seeking, often times with varying opinions and unclear answers, or no answers at all. Time after time I found myself discouraged with and tired of searching for the simplest bit of information and having to do the run around, and so would often find myself turning to Wikipedia for reference despite what my professors had advised me.

While it may the case that Wikipedia is not the most respected or complete source of information in academia, one cannot deny the sheer amount of crowd-sourced knowledge available in a single place. For essentially any language I would look up, there would be an entry, and though not always comprehensive or necessarily correct, it would provide the basic information I was seeking for back reference, while I could still go and look for more specific topics in scholarly journals and publications. Wikipedia provided the perfect platform to begin research and though I searched for a more reliable source, I continually found myself coming back to it.

The potential for Factotum in the linguistics community is quite great, and to demonstrate that, I used information pulled from Wikipedia for my test case to show how Factotum, even if used on not necessarily accurate and crowd-sourced information, can still yield results that are useful.

Obviously Wikipedia is a massive resource and I could not possibly use all the minute and nuanced details provided by the site, so I had to narrow down what information from Wikipedia I would feed to Factotum. While it may seem obvious, it is important to note that I only accessed Wikipedia pages in English (beginning with: en.wikipedia.org), and the pages I accessed would be language pages, that is Wikipedia pages strictly describing the language, not pages about countries or specific cultures but language pages. So my dataset is only going to use the English Wikipedia pages on the world's languages.

I myself would use these Wikipedia language pages primarily as quick references for meta-data on the language at hand. Every language page in Wikipedia has as part of it's template an 'Infobox' referred to as 'Infobox _language' containing the most rudimentary information about any given language. While language pages would vary in length and presence of sections and headings, they all contained this Infobox. And so I decided that this would be the snippet of information I would find every language page that contains ‘Infobox_language’ (conveniently there is a link to them from the template page for the Infobox), and pull from each of these pages to create my dataset for Factotum. I would use code I created, wikiscript.py, to scrape every language page containing the Infobox, and to grab all the information within the Infobox and turn it into Factotum facts that my parsers may use.

B. Information to be pulled from Wikipedia – The Infobox

The Infobox is structured with 3 main sections with three separate headings and subheadings within it (not all of which are required). The first section is required, it’s heading is the name of the language (e.g. French), along with it’s native name (e.g. Français). Within this section there are many subheadings (recall not all are required), including different types of subheadings depending on whether a given language is a natural language, constructed, or a sign language. The subheadings that pertain to mostly natural languages (some for sign languages) include ‘Pronunciation’, ‘Spoken in’ or ‘Signed in’, ‘Region’, ‘Extinct Language’ or ‘Language extinction’, ‘Total speakers’ or ‘Total signers’, ‘List of languages by number of native speakers’ or ‘Ranking’, ‘Language family’, ‘Standard forms’, ‘Dialects’, and ‘Writing system.’ The subheadings that refer to constructed languages include ‘Created by’, ‘Date founded’, ‘Setting and usage’, ‘Category (purpose)’, ‘Category (sources)’, ‘List of language regulators’ or ‘Regulated by.’ Essentially the first section outlines the most basic facts about the language.

I will briefly note that for the ‘Language family’ heading, the whole branching structure is provided for the language in question, but I elected to only store the most immediate parent family, making the type tree in Factotum more managable as well as providing some interesting results later on.

Next there is the optional second section, with the heading ‘Official Status,’ containing information about where the language is official and listing all the countries or domains where it is (subheading: ‘Official language in’), listing where it is a minority language, (subheading: ‘Recognised minority language in’) and what body regulates the language (subheading: ‘List of language regulators,’ or ‘Regulated by’).

Finally there is the third section, ‘Language codes’, and while on the Wikipedia page itself there is a map highlighting the regions where the language is spoken in varying degrees (e.g. mother tongue, official language, second language, and minority), I cannot represent this information visually in Factotum, and the information is already contained in the first section and the ‘Official status’ section which I may more easily translate into Factotum facts. But the rest of the information in the section I can represent In Factotum, with the subheadings representing different language codes ‘ISO 639-1’, ‘ISO 639-2’, ‘ISO 639-3’, and ‘Linguasphere Observatory’ or ‘Linguasphere.’

And so from these three sections is where I would pull my data for my test case. It should be noted however that given that I’m pulling the information off of Wikipedia, not all information is going to be accurate, there is going to be some controversy regarding some of the Infobox data, and I am limited in recording disputing interpretations just by what different language pages are available and what information is provided in the Infobox.

When I initially was creating my own data set sans Wikipedia, I was overwhelmed by all the design decisions I was forced to make concerning each language, but with the Infobox, I get a clean cut of each language without much difficulty. However, there are a few things would be useful to include in addition to the Infobox, which I will mention briefly later on. The fact file that I use with all the Wikipedia facts may be found in Appendix A. But now, I will discuss the workings of Factotum in greater detail.

IV. Overview of Factotum

Factotum is designed as a tool available to researchers to help them work with their data. In particular, “[a] major research task is to sort the data into related groups that create objects and attributes and try to see the fundamental relationships between objects that make sense out of the data” (Uzgalis 1993, 8), a task that Factotum is precisely equipped for. In this section, I will just be giving a high-level over view of Factotum, providing extra details only for items pertaining to my project’s use of Factotum; a more in depth look may be found in *Uzgalis*.

A. Using Factotum

First, the way Factotum is actually used is that the user enters their text-based data into a .f file (fact file/data file) with the freedom to specify the form of the data (8), while also adhering to the general formatting guidelines for Factotum facts. Next, Factotum builds and analyzes these facts (data) and the relations between them, creating a ‘vocabulary’ (21) which “provides a specification of the syntactic and semantic structure for the user’s data.” (34) And so, with the facts and now this vocabulary, the user may use the set of Factotum ‘tools’ to discover more information on the facts as well as manipulate them. The tools listed perform operations ranging from reformatting the facts, checking facts consistency, re-organizing the facts, and sorting the facts in different ways (37). For this project, I only implemented one tool, `fcheck`, which checks for the facts for consistency with the vocabulary and I will expand on it in greater detail further on. Now that we know how the user may use Factotum, I will go on and explain some of the details.

B. Factotum Terminology

It is necessary first to briefly define and describes some terms that I will be using throughout the paper. First there are ‘markers,’ which are symbols at the beginning of every fact, indicating how to treat this fact (14). While the full list of markers are listed in the Uzgalis paper, I will only be using markers which refer to formal facts: ‘ * ‘, ‘ “ ’ , ‘ : ’, ‘ :” ‘ , and ‘ –‘. Facts all begin with either a subject name, or one of these markers. If there is no necessarily explicit subject name, but a unique subject is needed, the marker ‘ * ‘ is used at the beginning of the fact in place of a written subject name. If a fact has the same subject as the previous fact, there is no need to write out the whole subject again, so ‘ “ ‘ is placed at the beginning of the fact, indicating that this fact has the same subject as the previous line. If a fact goes on for more than a single line, the ‘ – ‘ marker is used to indicate that this new line is still part of the same fact, since normally a new line indicates a new fact. The ‘:’ marker indicates the presence of a ‘predefined’ fact, which I will discuss in more detail in the section on facts. The ‘:’ is used in conjunction with the subject and not in place of it as with the other markers; but if the subject is the same as the previous line, the marker ‘:” ‘ may be used to indicate the presence of a predefined fact with the same subject.

Another term which is used quite frequently is ‘Subject.’ The ‘Subject’ of a given fact is what the fact itself pertains to. For example, in the given fact “`French Regulated by Académie française`” the subject is `French`, since that is what the rest of the fact is referring to. It should be noted that the subject of a fact is always in the initial position, and that the ‘rest of the fact’ may refer to by another term, ‘predicate.’ The ‘predicate’ is the remainder of the fact that is not the subject.

Subjects may also have ‘aliases.’ An alias can be a single word or they may be represented as a series of words (or as often referred to here, tokens); it should be noted that subjects are only allowed to be a single token (14-15). So say that `Français` is an alias for `French`, and our new fact begins with `Français`. That new fact may have `Français` as its subject, but since `Français` is an alias for `French`, this fact is linked to and thus really refers to the subject `French`.

So now we have all this facts with either unique subject names, or subject names that are in fact aliases for other subjects, thus referring to the same item. This item is what is called an ‘Entity’ in Factotum—“the first thing that Factotum processing does is collect together all facts with the same subject,” so all facts with same subject names, as well as all facts with alias referring to that same particular subject name are grouped together into an Entity, which infact shares the same name as the subject (18). It should also be noted that each Entity name must be unique, and if tow or more entities are created with the same name by accident, they will simply be merged together (18). Additionally, a subject name can be marked as an alias for a primary entity name.

The user may also indicate the subjects and entities to be of a certain ‘type’, where the type is any identifier chosen by the user. Types come into greater play in our discussion of the vocabulary and predefined facts.

Another term that will be used fairly heavily is ‘Object’ but will be discussed when discussing the vocabulary. Now that we have covered the main terms, I will now discuss the two main components of Factotum: the Data and the Vocabulary.

C. Data and facts in Factotum

The “data” in Factotum is stored in a file ending in ‘.f’ that contains all the facts that the researcher has entered. A fact is an observation or record of an external event represented in computer symbols, where this event may be true, imaginary, literary or formal; it also may be associated with a citation that justifies the observation. In essence, a fact is just an observation which as been recorded in a Factotum data file. (13)

A fact in appearance is much like a short sentence or series of words one would write out, but with the subject of the fact (as defined earlier) always in the initial position, and the predicate is comprised of everything following it. Predicates of facts are generally free form and the “domain of the researcher,” allowing them to “express attributes of a subject or relations between a subject and some or entity or entities” (17). Citations for the fact may be included at the end of the predicate, marked in between a pair of square brackets ‘[‘ and ‘]’; if there are multiple citations for a given fact, nested brackets may be used (17-18).

Recall that the ‘ “ ‘ marker indicates that the subject is the same as the previous line (fact). Here are a few examples of facts from my Wikipedia data, found in the `_wikidata_.f` file and the appendix[1]:

```
Arabic Total speakers Approx. 340 million native speakers
" Writing system Arabic alphabet
Bulgarian Total speakers 12 million
" Official language in Bulgaria
Bambara Spoken in Burkina Faso
Faroese Writing system Latin (Faroese variant)
" Spoken in Faroe Islands
```

Earlier, I briefly mentioned ‘predefined facts,’ which are facts beginning with the ‘:’ marker with a specified significance; the ‘:” ‘ indicates a predefined fact with the same subject as the previous line. Predefined facts in this project have three forms[2], one indicating the mapping of an alias to a subject (which is its primary term), another mapping a subject to a primary entity name, and finally, one which specifies the type of the subject (15). For this project, aliases are how I represent the native names of the language, types are how I represent language families, and I leave the entity naming to the vocabulary. Here are examples of predefined facts indicating the type and specifying the alias respectively:

```
:Greek [Hellenic]
:"<- Ελληνικά
```

[1] In Uzgalis there are additional markers indicating ‘informal’ facts, but in this project I will only be dealing with formal facts in my data

[2] In Uzgalis, there are four forms, but I will not be handling the predefined fact which includes another file.

So we can see that Greek is in the Hellenic language family (where Hellenic is the most immediate parent language family), and the native name for Greek (in Greek) is Ελληνικά.

Facts are to be broken apart and analyzed by Factotum, so that the tools may be used on these facts and to create a vocabulary for these facts to also be used by these tools.

D. Factotum Vocabulary

The vocabulary is contained in a separate file ending in '.v' which contains rules that dictate the format of facts in the .f file. Though the vocabulary may be generated automatically using mkvocab.py (provided by Uzgalis), it can be manually altered (or even created manually) to provide more precise rules for the facts; it usually better to use the generated vocabulary at first to detect errors in the fact file, as it is more likely that errors are present in the file entered by the researcher than produced by the machine. For this project I will rely on a vocabulary, _wikidata_.v, produced by mkvocab.py from _wikidata_.f. The vocabulary is a necessary part of Factotum because it helps Factotum to understand what part of the facts are references to other subjects and entities and expressions of relations – breaking down the facts is key in processing all the data with Factotum tools. Essentially, the vocabulary provides rules for the syntactic and semantic structure for the facts (data) provided by the user (34).

There are several types of vocabulary rules described in Uzgalis but in this project I will be parsing six of these rules: ':=' (specifying the syntax of the fact), '-=' (specifying the syntax of the fact), if-then blocks, '~>' (implied), '=>>' (generated), and [] (type defintion). Every vocabulary rule begins with a subject which is not necessarily connected to or related to the subjects in the data file, but helps to group vocabulary rules together. This subject is then followed by a predicate, beginning with a symbol indicating what type of rule it is (e.g. :=), which is then followed by a phrase string. The beginning of every phrase string must begin a reference to a subject or entity, or '<>' a placeholder for any subject or entity since all facts begin with a subject; multiple references to entities are allowed in the phrase rule (21).

For the purposes of this project, my vocabulary parser, predpar.py , parses all of these aforementioned vocabulary rules for correct form, but does not act on the true purpose of the implication (~>) or generating (=>>) rules, who's predicates are meant to add additional facts to the data file. The majority of the rules in this project are either specifying the syntax of the fact or type definitions and so those are the rules I will be discussing in this section, descriptions of others maybe found in Uzgalis.

First, I will briefly describe what an object is and how it may be used. An object in a vocabulary rule is always marked with angle brackets '<', '>'. If there is nothing between these angle brackets, then this means in the fact that any subject type may go in that that place. But if there are items between the brackets, this can mean several things. First if it is just word token(s) between the brackets (no colon ':' or equal sign '='), this means that either a type-name or phrase-name is represented by the object. While the lexically a type-name and phrase-name are indistinguishable, we determine if it is a type by checking for the presence of the tokens in the type tree (types may be multiworded), or we determine if it is a phrase, by checking it against entries in a phrase list compiled by ‘-=’ phrase rules. If it is indeed in the type tree, then the object place in the fact must be of that specified type (27). If it is a phrase-name, then whatever falls into the object place in the fact must follow the syntax specified by that specific phrase-name rule. For example, in the rule:

```
    NP-Complete := <problem> is an NP-complete problem
```

The fact must begin with a subject that is of type problem, followed by the word tokens “is an NP-complete problem.”

If a colon ‘:’ is present in the object then this refers to labels and token type specification. If the object is of the form < *word token* : >, this means that the word token is a ‘Label.’ In the fact, whatever word falls in the place of this object is given the label specified by the word token in the vocabulary rule. For example in the following vocabulary rule, whatever represents the number of edges in a given geo-metric-figure is given the label no-of-edges:

```
    Polygon := <geometric-figure> has <no-of-edges:> edges
```

If the object is of the form <: *letterkey* >, this means that the letter key a token-type specification. The token-type-specification is represented as one of four character symbols: ‘n’, ‘s’, ‘w’, meaning (respectively) number, string, and word. So this means that the object is specifying that whatever falls in this place in the fact must be of this specific token-type: a number, a string, or a single word (27). For example in the following rule, the fact describing a subject of type geometric-figure, has a certain number of edges, that must be indeed a number as noted by the ‘n’ inside the brackets:

```
    Polygon := <geometric-figure> has <:n> edges
```

Finally there is the object that combines these specifications of label and token-type-specification, of the form: < *label* : *token-type-specification* > in which whatever falls in the object’s place in the fact must be of the given token-type, and is then given the label provided by label (27). So combining these two for our example, we get:

```
    Polygon := <geometric-figure> has <no-of-edges : n> edges
```

This rule now assures that the number of edges is indeed a number, and then labeling it as ‘no-of-edges.’

Finally, we have the objects in rules that combine labels with phrase-names and type-names of the form < *label* = *phrase-name* > and < *label* = type-name > respectively. So after checking if the item in place indeed matches the type-name or phrase-name, a label is then created of this phrase-name or type-name, and the item in place is given this label (27).

And so, objects are a major component of the vocabulary, providing more structure for representing the relations between objects and subjects along with the syntactic format. Again, Uzgalis provides greater detail but I have discussed what I believe is sufficient for understanding the use of the vocabulary in my project. I will now discuss the actual workings my parsers and code, along with their results.

V. Code

In this section I will describe the main modules I have written for this project. But first I will briefly illustrate the flow of the program in the following diagrams. Figure one depicts how the is collected, with `wikiscript.py` producing the data file `_wikidata_.f.` This is then passed to mkvocab.py, which automatically produces a vocabulary and presents other information on the data, which will be in `_wikidata_.orig.v.` Next, `pullvocab.py` is called to grab only the possible vocabulary rules from `_wikidata_.orig.v` and places them in the file `_wikidata_.v.`

The next figure, Figure 2 illustrates the flow of the main programs, where `fcheck.py` requires `_wikidata_.f` as input, and also calls `predpar.py`. In turn, `predpar.py` requires a file with the vocabulary rules as input, and then proceeds to call `capitlaization_unify.py` to get rid of duplicate vocabulary rules with only differences in capitalization.

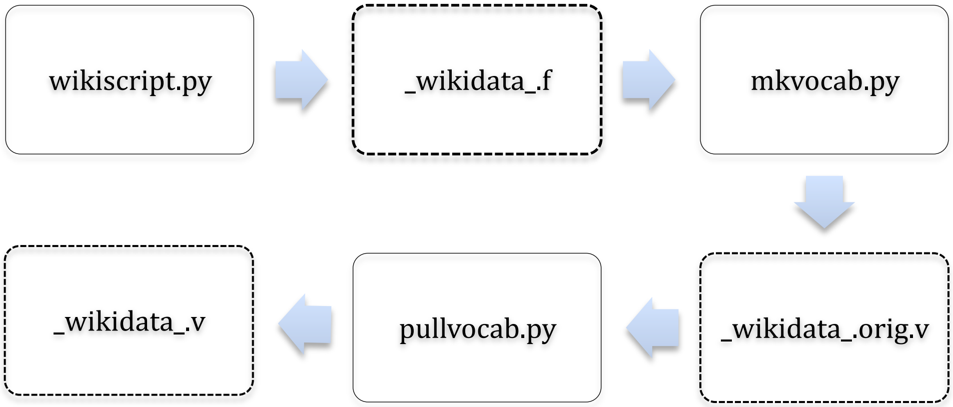

Figure 1: where the Python modules have solid lines (and end in .py), and fact and vocabulary files have dashed outlines. The arrows indicate input and output of the files.

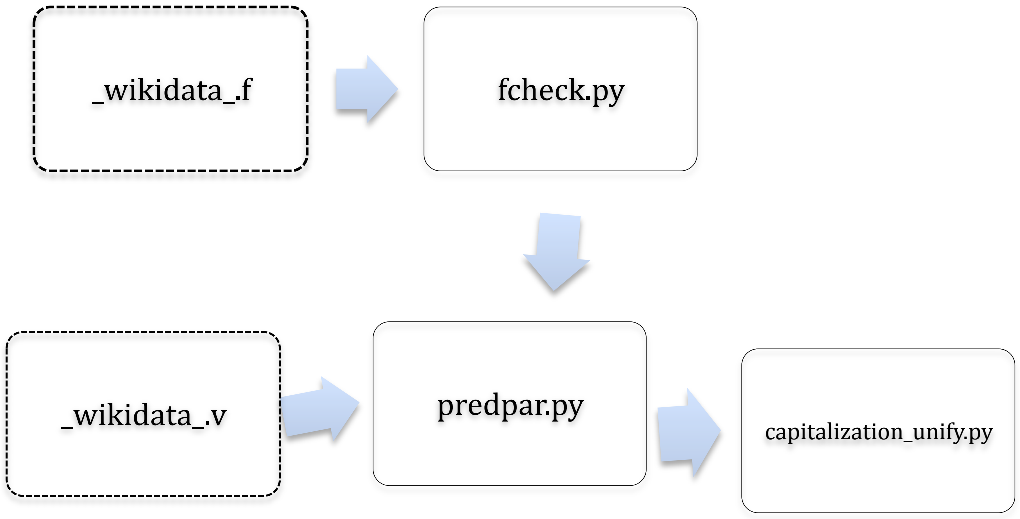

Figure 2: Python modules have solid lines and the .f and .v files have dashed lines. Fcheck.py takes the .f file as input, and also calls predpar.py, which in turn takes the .v file as input and calls capitalization_unify.py

A. The Vocabulary Parser: `Predpar.py`

i. Essential Definitions

'`Predpar.py`' is one of the most significant and essential modules that I have written for this project and for Factotum; `predpar.py` is the vocabulary parser and its function is to check whether the vocabulary rules recorded in a given '.v' file adhere to the required format of vocabulary rules, designated by the grammar contained by a global Python dictionary I created, '`vocab_grammar`.' This dictionary effectively transliterates the format given in the original paper, into a form that the module may more easily use. The keys of the dictionary represent the non-terminal symbols (or Left-hand side of the grammar rule, LHS) and are represented as strings, and entries for each key represents the right-hand side (RHS) of the grammar rule. Since there may be multiple symbols (or as I will refer to them, tokens) in the RHS of the grammar rule, each RHS will be represented as a list of the tokens (represented as strings), even if it only contains a single symbol.

For example, for the rules X → Y Z, and W → V, the dictionary entries would be as such:

```
     Example_Dictionary ={ 'X' : ['Y' , 'Z'],
                           'W': ['V']   }
```

Also, if there are multiple options for the right-hand side of a given non-terminal (which is almost always the case in `predpar.py`), then the entry is represented as a list of lists, a list of all possible right-hand sides, which as we remember are already represented as lists.

So to use an example from the vocab_grammar itself, we have the rule for the predicate ‘Pred’ which has multiple options for the RHS:

Pred → := Phrase I -= Phrase I ~> Phrase I =>> Phrase I ? (Cond) Then I Phrase

The entry in the dictionary looks like this:

```
Vocab_grammar = {... ‘Pred’: [[‘:=’, ‘Phrase’],
                              [‘-=’, ‘Phrase],
                              [‘~>’, ‘Phrase’],
                              [‘=>>’, ‘Phrase’],
                              [‘?’, ‘(‘, ‘Cond’, ‘)’, ‘Then’][3*],
                              [‘Phrase’]]...}
```

Since this parser requires me to go through this vocab_grammar multiple times and every time I want to check on a rule, so it was imperative that first I, understand the structure so that I could write the code and rules, and it is also exigent that the structure be easy to iterate through, and of course for the parser to understand so that it may match the rule against it.

Also I should note that so items in rule are not so easily defined as terminal and non-terminal symbols. While the token ‘:=’ is straightforward to match against, tokens such as ‘Words, ‘Typename’, and ‘Label; are not so easy to match against a single symbol, and other tokens such as ‘Op’ and ‘Exp’ may have multiple permissible options for a single symbol . So I have created a special method for dealing with these sorts of cases. Though technically entries in the rule that fall into the category ‘Words’ or ‘Op’ are terminal symbols, I keep these terms as non-terminals. Though these non-terminals do not appear in ‘vocab_grammar’ they do appear in the additional global dictionary I created, ‘regex_dict’. The keys in ‘regex_dict’ are all tokens which may not be represented as simply a single symbol, such as the non-terminals I previously mentioned. The entries for these keys are the regular expression pattern which best represents what is allowed for the token. Regex_dict uses the method from the regular expression library (re) re.compile to create the desired pattern in the dictionary entry, and so each entry is not a string, but a regular expression.

For example, from the paper and my discussion with Uzgalis, ‘Typename’ must be at least one character long, with no whitespaces, and may contain lowercase letters, uppercase letters, hyphen, underscore, digits, or apostrophes (inclusive). This would be represented in the regex_dict as such:

[*] note: ‘?’, ‘(‘ and ’)’ are represented here as literals for the sake of example here, but in predpar.py they are represented in a format which allows them to be understood properly by regular expressions: ‘\?’, ‘\(‘, ‘\)’

```
Regex_dict = {...
            'Typename': re.compile('^[-_0-9a-zA-Z\']+$'),
            ...}
```

This works well for all tokens that do not have a strict single symbol definition, however there are the two special cases of 'Words' and 'Msg'. While we can define a single word, and a single msg, these tokens allow more than one match, there may be more than one word (set of characters without whitespace within it) in 'Words', and more than one word in 'Msg.' So to account for this, I have the small global list 'Repeat' which contains tokens which are defined in regex_dict, but allow more than one term/word for a match. This is essential for parsing, and is used and checked in the main parsing function, parseGrammar, which will be discussed at length later on. Here is what this global looks like :

```
Repeat = ['Words', 'Msg']
```

Also, there is another small global list, Second_check which contains 'Typename' and 'Phrasename.' Prepar.py makes two passes through the vocabulary, and on the first pass, it attempts to pull out all possible type definitions (typenames), and it is only on the second pass that it can accurately check if an item in the vocabulary rule we are checking is actually of this type ('Typename'). Since in the regex_dict, the pattern entries for 'Typename' and 'Phrasename' are indistinguishable, it is only the type tree 'TypeTree' is built after the first pass that we can check if indeed a given item is a properly defined 'Typename' and the way parseGrammar knows to check the type with the typetree on the second pass is by noticing' (by using the function needs_sec_check) that the token ('Typename') is in this list, 'Second_check.' Note that if after getting the second check (check_second_check) the item fails to be a properly defined type, it returns false so that the parser may move onto the correct rule (hopefully containing 'Phrasename').

Now that I have explained how I represent the core definitions essential to predpar.py, I will discuss the flow of the module, with noted detail on the parsing function, parseGrammar.

ii. Flow of Predpar.py

When running predpar.py, parse_vocab() serves as the 'main' function, controlling the flow of the module and directing how everything goes. It first grabs the rules we are to parse from the .v file, using the function go_thru_file() and storing the results in the list facts using a similar method as factotum_lexer.py, adding each item in the form `[subject, predicate]`[4] to the list of rules(facts) it will ultimately return; so `go_thru_file()` returns the rules as a list of lists.

For example, if a rule in the file is represented as: NP-Complete := <> is an NP-complete problem, it will look like this in after being pulled out of the file and right before being added to the full list:

```
  ['NP-Complete', ':= <> is an NP-complete problem']
```

Next, `parse_vocab` passes these newfound rules (stored in '`facts`') to the function `first pass`, whose results are stored in the lists `parsed_rules` and `failed_rules`. `First_pass` iterates through the facts, first tokenizing the predicates using the function '`tokenize_pred_sting.`' Here we take tokenize to mean the separation of meaningful symbols in the predicate string and representing them as a list of tokens. The significant symbols range from designated symbols of ':=' and '<' or ')' to just words separated by whitespace. What is considered a token is represented in a regular expression pattern called tokens within the function. If a rule fails to tokenize, it is thrown out (e.g. added to the list of failed rules), otherwise it is stored in `rule_pred`; we now attempt to parse it by calling `parseGrammar`. Rules that are type definitions have a slightly different form, and so we must slightly modify `rule_pred` before passing it into `parseGrammar`.

During the first pass, `parseGrammar` checks the global variable '`PassedThru`', which is set to 0 at the beginning of `firstPass` and set to 1 in `parse_vocab` after `firstPass` returns. The first pass is necessary so that we may build up a type tree that we can check against in the second pass and `PassedThru` is what I use to indicate to `parseGrammar` that this is the first pass though the rules (if it is equal to 0) and it needs to call the function `update_TypeTree().`

Type definitions in Factotum go as following: NODE [HEAD] , where NODE is the current item and a new type, and HEAD is the parent of NODE, and NODE is a subtype of HEAD. If HEAD is empty, that means that node is in fact a root node and head of a type tree. There are always at least 3 items in a type definition, NODE, and the brackets '[' and ']'; if HEAD is present, then there is one additional item, making the definition of length four. The global '`TypeTree`' is a dictionary where the keys are subtypes/ the NODEs, and their respective entries are a list of two items, the first being a Boolean (True or False) and the second being the head/parent node (or 'ROOT' if root node). The Boolean represents if the type is reachable and can be accessed from the root node (True), but this gets checked later on (note: root nodes are automatically reachable, so immediately set to True).

[4] It is important to note that 'subjects' in the vocabulary rules only apply to the vocabulary rules and not actual facts (which are checked in fcheck.py). Subjects in predpar.py group together different vocabulary rules. Subjects of rules do not correlate to subjects of actual facts, and do not specify which facts the rule applies to (it may apply to any fact with any subject).

So when parseGrammar returns back to firstPass, it may have updated the TypeTree using the abovementioned function, but in the majority of cases, it simply attempted to parse the rule, and returns the parse tree upon success. If it was indeed successful, then the rule's subject (rule[0]), string version of the predicate (rule[1]), and the tokenized predicate (rule_pred) all get appended to the list of successfully parsed rules:

```
    parsed.append([rule[0], rule[1], rule_pred])
```

Once firstPass has iterated through all the rules, performing all these steps (tokenizing, parsing, and appending to designated list), it returns a two-itemed list, containing the lists of successfully parsed rules (stored in parsed) and rules that failed for one reason or another (stored in failed): return [parsed, failed]

After getting the results from firstPass and storing them in parsed_rules and failed_rules respectively, parse_vocab calls on check_types() to go through the newly formed type tree and make sure that all types are reachable and valid. check_types() will correct the type tree of any issues itself, but cannot modify these erroneous type definitions from the rules themselves, and so it returns a list of types to be removed from the rules, which parse_vocab stores in removeT. If r removeT is not an empty list, then parse_vocab iterates through items in removeT, and the parsed_rules; if the first token in the rule (the "subject") matches an item in removeT, that rule is removed from parsed_rules, and added to failed_rules.

So after check_types() returns and parse_vocab modifies parsed_rules and failed_rules accordingly, it goes through and does a second parsing pass of the vocabulary rules. It loops through the parsed_rules, making sure that the global PassedThru is set to 1 (importantly not 0), and calling parseGrammar on the rule, and storing the result in 'second_pass'. If the rule fails to parse on this second pass, it is appended to failed_rules. But if it does indeed parse, we add it to the new list of parsed rules 'second_parsed_rules' and adds it to the new grammar dictionary fcheck_dict (which may be used to check facts in fcheck.py) using the function _fcheckDictEDIT_. Finally, after all the rules have been passed through this second time, parse_vocab (and predpar.py) returns 5 item which will ultimately be used in fcheck.py:

```
    return (second_parsed_rules, failed_rules, new_dict,
            TypeTree, fcheck_dict)
```

iii. Parsing function: parseGrammar

Finally we get to the most important function of predpar.py: parseGrammar. This recursive function takes in the rule predicate to be parsed (rulePred) and a starting symbol: a non-terminal LHS of the rule, which we know is present in the keys in the vocab_grammar dictionary. When parseGrammar is called for the first time in firstPass or during the second pass, the starting symbol passed in will always be 'Start' given the structure of vocab_grammar, however on subsequent recursive calls, it may be any other non-terminal token; also parseGrammar is a sort of top down parser, going depth-first and left to right in the rules.

Upon successfully parsing the rule, parseGrammar returns a parse tree and the remainder of the rule predicate list (which should be empty if parseGrammar has finished). The parse tree (stored in the variable tree) is represented as a list, containing all the grammar rules that parseGrammar has gone through while parsing the rule predicate. Each grammar rule within tree is represented as a pair with the form: [LHS, RHS], where LHS (variable key in parseGrammar) is the non-terminal/key in vocab_grammar that was used, and RHS is the one right-hand side rule of the non-terminal that was used to help complete the parse. It should be noted that RHS is just one rule, not the entire entry for the non-terminal key in vocab_grammar. For example: if we have a predicate and the first two rules of it's parse are: Start → Pred and Pred → := Phrase, then the beginning of the parse tree would look like:

```
tree = [['Start', ['Pred']], ['Pred', [':=', 'Phrase']...]
```

In parseGrammar, this is accomplished at the beginning of each iteration through a RHS rule (variable rtuple) with the line:

```
tree.append([key, rtuple])
```

It is important to note that representing the pair for non-terminals with entries in regex_dict differ slightly from the above example. The non-terminal token refers to a pattern which if was put in the pair, would not display properly in the output and (more importantly) would not be informative—there would be no immediate way to tell which items in the rulePred matched this pattern, especially if the user was not aware of what the pattern is and cannot cross-reference the results. So in this case, parseGrammar does not follow the same method, but makes the pair with the form: [REGEX_LHS, MATCH] where REGEX_LHS is the non-terminal token and MATCH is item in the rule predicate that matches the regular expression pattern designated by the entry for the non-terminal token in regex_dict. Note, if the pattern allows repetition, MATCH is not a single string, but a list of all consecutive strings that match the pattern. For example: if we have the token 'Typename' and the first item in rulePred is ['French', ...] (assuming we have already validated the type and matched the pattern), the pair would appear as:

```
['Typename', 'French']
```

If we had the token 'Words' and the remainder of the rulePred as ['Regulated', 'by', 'the', 'Académie', 'Française'] the pair would look like:

```
[‘Words’, [‘Regulated’, ‘by’, ‘the’, ‘Académie’,
‘Française’]]
```

Now that I have shown what `parseGrammar` is supposed to return, I will explain the inner workings.

First, `parseGrammar` looks up and grabs the entry for the starting symbol (`key`) in the dictionary `vocab_grammar` and stores the entry as ‘rules.’ Next it iterates through them one by one (each `rtuple in rules`). For each iteration of a RHS rule (`rtuple`), the parse tree is reset to the empty list, the current key and `rtuple` added onto it as I described earlier. The list ‘`local’` is just a copy of the rule predicate passed into `parseGrammar`, and ‘n’ keeps track of what point `parseGrammar` is in ‘`local’`, and count keeps track of where `parseGrammar` is in ‘`rtuple’.` The protocol followed for parsing is the essentially the same for both when the entry contains only a single token: [‘`Pred’`] and list of tokens: `[':=', 'Phrase']. But` there is a distinction provided by an if-statement, so I don’t iterate through the characters of a single token, when I mean just iterate through the list of tokens in `rtuple`.

So if the entry currently being examined does indeed more than one item (‘`token`’), we iterate through them: if the token in question is a non-terminal and a key in `vocab_grammar`, we recursively call `parseGrammar` with this token as the starting symbol, and local from the point n onwards (`local[n:]).` If it is successful upon it’s return, then we extend the parse tree (`tree`) and update local. If count is equal to the length of `rtuple` (we have reached the end of the RHS rule and were successful), then we return the tree and local, otherwise we reset n to 0 (since local has been updated, we have a new starting point) and continue onwards through the rest of the tokens in `rtuple`. However, if it was not successful, then we break out of going through this `rtuple`, and move onto the next option found in `rules`.

If the token is not a non-terminal or is not a key in `vocab_grammar`, this means it is a regular expression! Recall that we have two types of regular expressions: those that are actually entries in `regex_dict`, and those that are just symbol matches. The function `check_regex()` checks if the token is present in `regex_dict`. If the token is not, we jump down and do a simple `re.match` with the token, and the current item in local (`local[n]`)–if there is a match, we increase n by 1, check if we are at the end of the `rtuple` (and return the tree and the rest of local if we are) or otherwise continue. If there is no match, then we break and move onto the next option in rules (the next rtuple).

If the token is indeed in `regex_dict`, we store the pattern and check if `local[n]` matches the pattern—if it fails to, we break out of this to the next option in rules, but if there is a match, we then check if `PassedThru` is greater than zero. If it is, we check if the token requires us to have a second check (only `Typename` or `Phrasename`) and then proceed to use `check_second_check` to verify if `local[n:]` is a valid `Typename` or `Phrasename`, either continuing if it is valid, or breaking out of the current `rtuple` if it is not.

Next, we check if the token is contained in the list `Repeat`. If it is, we go through local, updating n along the way, until `local[n]` no longer matches the regular expression pattern given by the token.

Then after both these steps (and whether repeated or not), the tree is updated to include the token and the 'local' match as described earlier in the section about the parse tree and entries from `regex_dict`, update n, and returns the tree and updated local if we have reached the end of the current right-handside rule (`rtuple`), otherwise we continue onto the next token in `rtuple`.

And so we continue on, iterating through right-hand side rules as we come across non-terminals until all are iterated through and parsed in the above manner, and all tokens in local are matched, making local an empty list. If all this works out and returns a parse tree (along with the empty local list), this means that the rule predicate has been successfully parsed according to the grammar rules which dictate the format of the vocabulary.

iv. Creating a usable Grammar

So after the second pass through the rule predicates, those that have been parsed successfully are stored in a list of successfully parsed rules (as mentioned earlier) and then uses the parse tree to help form a new grammar that may be used to check facts against their vocabulary rules in fcheck.py, also mentioned earlier.

So I actually create two 'grammar' dictionaries, `new_dict` was my first version before realizing that it would not work well as a grammar for facts, and `fcheck_dict` the more useful of the two for fcheck.py . The primary difference between the two grammar dictionaries is as follows: `new_dict`'s keys are organized by the subjects of the vocabulary rules and includes all vocabulary rules. Each subject entry is a dictionary of a mini grammar (all the vocabulary rules with the same subject). For example two different subjects in `new_dict` may both have different entries for 'Start', 'Phrase', etc. On the other hand, `fcheck_dict` ignores the subjects of the vocabulary rules, and only includes rules which begin with the symbol ':=' indicating 'rule is,' meaning that this string specifies the syntax of a fact and so is the only type of rule that is really applicable to facts.

So while an entry in `new_dict` may look as follows:

```
New_dict = {'NP-Complete':
               {'Obj' : ['Typename'],
                'Typename': [ANY, problem],
                'Pred' : [':=', 'Phrase'],
                ‘Start’ :['Pred'],
                ‘Words’: ['is', 'an', 'NP-complete', 'problem'],
               ‘Phrase’: ['Obj', 'Words']} ,
                ‘mortal’: {‘ Start’:['TypeDef'], ...},
            ...}
```

An entry in `fcheck_dict` would appear as follows:

```
Fcheck_dict = {‘Phrase’:
                [['Type: ANY', 'is', 'an', 'NP-complete',
                'problem'],
                ['Type: problem', 'transformed', 'to', 'Type:
                problem', 'polynomially']
                ...],
               ...}
```

I will only discuss the function used to create `fcheck_dict` , `add_fcheckDictEDIT,` as it is more pertinent to the rest of the project.

`Add_fcheckDictEDIT` takes in two arguments, the parse tree of a given vocabulary rule and a dictionary (`fcheck_dict`) and though it does not return anything in particular, it builds on `fcheck_dict` throughout the function. First it checks if the parse tree contains ‘:=’ by checking for the pair *`['Pred', [':=', 'Phrase']],`* since as discussed earlier in the section going over Factotum, vocabulary rules beginning with ‘:=’ indicate that they specify the syntax of the fact. However, I came to find that when running my *`_wikidata_`*`.f` facts through the automated vocabulary generator, none of my rules began in this manner, so I modified the function to allow also for the pair `[‘Pred’, [‘Phrase’]]`. If neither of the pairs are present, then nothing further happens and the function simply returns. However, if a pair is indeed present, then we need to build up the fcheck dictionary, beginning immediately after this pair.

Then, the function goes through the right-hand side of the next pair, checking if each item is a key (nonterminal) in `vocab_grammar` or `regex_dict`. If it is not, then the item is simply appended to the entry (which will ultimately be inserted into the dictionary). But if it is in either of the global dictionaries, then add_fcheckdictedit calls the function, `getTermSymbs`. `GetTermSymbs` takes in the item (nonterm) and the remainder of the parse tree in turn calls ‘`findInstanceInTree`,’ which looks up the first pair in the remainder parse tree with `nonterm` as the first item in the pair, and returns this ‘branch’ of the parse tree, as well as the index of this branch in the tree, so that we may update the tree.

Once this returns, getTermSymbs then iterates through this second item, checking if it is in the keys, if it isn't, then brackets are removed and the terminal item is appended to the resultant list, otherwise it is matched with Typenames, Labels, and Ttypespecs, so that it may be better transliterated to the dictionary (and then subsequently calls findInstanceInTree). Typenames are stored as a single token in the item list, instead of having a separate entry for 'Typename' and the actual name of the type is displayed as:

```
[...'Type: problem',...].
```

'Label' and 'Ttypespec' may displayed alone in a similar manner:

```
[...'Label: donor'...]
[...'Ttype:s',...]
```

However, both can be used together, where Ttype determines what the token type is for the item with the label, in which case I have them displayed as a list together:

```
[...,['Label:money', 'Ttype:n'],...]
```

If the item does not match one of these three non-terminals, then getTermSymbols is recursively called with this item and the remaining parse tree (a counter is updated with every iteration, showing how much of the parse tree to pass). When this recursive call is returned, then the item list is extended by the return value.

And so by the end of parse_vocab, fcheck_dict is entirely built up, as well as the list of parsed rules (and their respective parse trees), failed rules, and type tree, which all get returned at the end of predpar.py.

B. The Fact Checker: Fcheck.py

Fcheck.py is the second module I wrote for Factotum. This is the module that is mainly focused on checking the facts (data found in the .f file) against the vocabulary rules (.v file) such that the facts follow the allowed format specified by the given vocabulary. Additionally, it is expected that both the vocabulary file (.v) and fact file (.f) are passed to the module as command line arguments (in that order). The 'main' function in this module is 'fact_checker()' which controls the flow of the fact checker, much like parse_vocab() in predpar.py.

After initializing some variables, fact_checker() calls on the function 'check_vocab()' which calls on prepar.py's parse_vocab() to check the vocabulary found in the provide .v file as well as generate a useable grammar dictionary. When check_vocab() returns back to fact_checker(), all the return values are stored (and if it is found that voacabulary failed to parse, the program ends).

Next, using the function 'go_thru_factFile()' (similar to the go_thru_file of predpar.py), fcheck grabs all the facts from the provided .f file and stores each fact as a pair, of [marker, factstring] if a marker is present, or simply [subject, factstring], into a single list of facts.

Once this returns with our list of facts, fact_checker (similarly as in parse_Vocab) does two passes through the facts. The first pass is in fact operated by a seaparate function: first_pass.

i. The first pass through the data

The purpose of the first pass is to pull out the predefined rules which contain information that will be needed before completely going through all of the facts definitively. Predefined rules include type definitions, as well as aliases, and a rule is checked by the function 'isPredef(fact)' and so if a rule is of the predefined form, isPredef(fact) will return true, and then the fact is analyzed using the function checkPreDef.

First_pass loops through the facts, one by one, tokenizing each one using the tokenizing function found in predpar.py before checking if the fact is a predefined fact (by using isPredef). If it is in fact a predefined rule, it should begin with colon (:), however I also make an exception for the markers that can only be found in predefined rules (even if colon not present) such as alias arrows, and type markers, and simply printing a notice that the colon was forgotten. Next, if it is indeed predefined, checkPreDef is called: Type definitions are added to the type hierarchy and then checked for completeness. Aliases are added to the two dictionaries, one mapping the alias to it's primary term and the other where the Subject/Primary term is mapped to all it's aliases. It is important to note that while one entity (primary term) may have multiple aliases, a single alias may refer to only one entity. But if a fact is not a predefined fact, it is simply added onto a new list of facts.

Once all the facts have been iterated through, the types are checked using fcheck_types, which goes through the type hierarchy in a similar manner as the type checker in the vocabulary parser (Predpar.py). It iterates through each type, and searching for a path to the root as well as loops, marking those nodes where a path has been found, and after checking all of the types, removing those that failed to reach a root node from the type tree. (Note: since types, which are predefined facts, are never added the list of facts, we do not have to remove them from the list of facts).

There is one more check to do before moving onto the second pass, and that is to denote instances of 'multialiases' within the remaining facts (not the predefined ones). Recall my earlier discussion of primary terms and secondary terms (aliases). It may be the case that the secondary term (alias) is more than one word, which I refer to as a 'multialias.' Now even though we've stored the multialias and can access it, even if it is present within the fact, it may be tokenized into separate words, rendering it unrecognizable. So in this last check, I search through the remaining facts for the presence of a multialias as separate tokens; if it is present, then I replace this tokens with a single string, making it possible to identify them when parsing. After this final check is complete, the facts are returned along with the list of failed rule (e.g. rules which failed to tokenize).

ii. Second pass through the data: `parseRD_Facts`

Now that we've sorted out all the predefined rules, we can move onto the second pass through the facts, which will actually parse the facts. This is controlled through the 'main' function (`fact_checker`) with a loop, which iterates through each rule, calling the parsing function, `parse_Facts`, on each one, and upon it's return, either adding the fact and it's parse tree into list of parsed facts or into the list of failed facts. Immediately after a fact is parsed and success/failure has been determined, it gets written to a file of successfully parsed facts and another file of facts that failed to parse, making it easier to see right away how the parser is progressing.

`ParseRD_Facts` is an extremely similar parser as in `predpar`.py, taking in the fact, and a starting symbol, but unlike `predpar.py`'s `parseGrammar`, it also takes in Boolean '`factFinished`' indicating whether the fact has been completely iterated through, and an integer '`next`' storing our place in the fact. This is important because unlike in `predpar.py`, where I organized the vocabulary grammar so that the longest match would be guaranteed, the `fcheck` grammar dictionary is not in order and so I have to account for this in the parser this time using `factFinished` and a global integer `entrypoint` which is reset with every fact that is parsed.

`ParseRD_Facts` begins with grabbing the starting symbol rules from the grammar dictionary and iterating through them; like in `predpar.py`, there is a separation between rules that contain a list of tokens (the norm) and rules that contain a single item (such as `[Start, [Phrase]]`).

If there is only a single item in the rule, we check if it is a terminal or non-terminal symbol; if it is a terminal symbol then we try to match it with the fact, following the same process as described below for rules with more than one token. However, if this single-itemed rule is a non-terminal, we advance the global `entrypoint` by one before calling on the parser recursively with the non-terminal. We do this to ensure that the rule will get the longest possible parse for the fact, and not return early without having the fact parsed all the way, or not attempting longer parses after realizing the shorter parse.

If there is more than one token in a rule, we iterate through each token in the rule, checking if it is a terminal symbol using `isTerminal`,which returns true or false. If it is not a terminal, then we recursively call the parser on this token, but without advancing the `entrypoint`, because since this non-terminal is within a loop for tokens, if it fails it will still be within the loop of rules and don't have to worry about returning early.

If it is indeed a terminal symbol, we then go and try to match the terminal symbol using the function matchTerminalSymbol, taking in the token in question, the fact, the index, the parse tree, the rule and the count in the rule. This function deals with not just simple symbol matching, but also the cases where there are types, labels, token-type specification, using the functions isDescendent, checkLabel, and checkTtype respectively to check whether the current position in the fact matches against them.

If the token is not a match, then we break out of the current rule and move onto the next one. However if it is a match, then we only return if we have reached the end of the fact stream, and then end of the rule; if this is the case, then we may set factFinished to True, and return the parse tree, the index, and factFinished. If the fact is not finished, but we have gone through all tokens in this rule, factFinished is set to false. Next we check if the entrypoint is greater than 1, if it is then we are fine to return because it will return to some other point in that will have access to more tokens and symbols, otherwise, we know that the return would lead us to a dead end.

Once the rule has been parsed through (or not), it returns back to the "main" function fact_checker, where it is then added to either the list of successfully parsed rules or failed rules and ultimately returns.

C. The Wikipedia scraper: Wikiscript.py

Now I am going to explain in some detail the code I used to access and scrape these Wikipedia pages and how I transformed the raw data into usable Factotum facts. I created the module 'wikiscript.py' to perform these automated tasks with the help of the urllib2 library for pulling content off the sites, and also the string and sys libraries for general use. The three biggest functions contributing to the general flow of wikiscript are wiki_main, collectData, and parse_wiki.

So wiki_main serves as the 'main function' to the whole program, first checking if there is a file provided already listing all the links to access or that have already been accessed. If this file is empty, this means that 'wikiscript.py has to go through and grab all the language pages with infoboxes, and all their respective content, which is the most time consuming component of organizing the data. If the file is full however, this means that we know which pages to access and jump down to the second section of wiki_main. Appendix G contains all the links that I accessed when using this module.

In the former case (empty file), collectData is called to grab all the links that currently contain the Infobox template. Note that the url which is necessary for wiki_main,'*http://en.wikipedia.org/wiki/Template:Infobox_language'*, is hardcoded. Then only after collecting all possible links, it returns to wiki_main, and proceeds to go through each link, grabs the content from the link, records the permalink in the links file, checks if a file exists with the permalink as the name, if it is present and the file has content, it moves onto the next link, however if is empty, it writes the html content to the file; if the file doesn't exist, it creates a new file with the permalink as the filename, and stores the html content in this file. Once this is complete, it moves onto the next section of `wiki_main` (which is also where it jumps to if a file of links is already present).

This section then opens the file containing all of the permalinks, searches for the matching file name with the permalink, opens this file and calls `parse_wiki` to then grab information on the Infobox from the content, separate it into the three designated sections, and write all the facts from this language page's Infobox to the fact (.f) file either specified by the user or created the fact writing function `writeFacts`.

If for some reason this fails, `wiki_main` attempts to parse the content one more time but instead pulls the content manually from the permalink page online rather than checking for a file which may not exist in this case. If any link fails to parse, an error message is printed.

After going through all the links provided, it returns and has produced a full .f fact file.

VI. Results and Performance

i. Results and Issues

So first, I had to gather all the facts and I did so by running `wikiscript.py` and getting the resulting `_wikidata_.f` file, containing all the facts gathered from the Infoboxes. Next, I used `mkvocab.py` with this fact file to produce vocabulary rules for this data and placed them into the `_wikidata_.orig.v` file; `mkvocab.py` outputs a lot of other information as well which I did not need to use, so I created and ran the quick `pullvocab.py` which just grabs the possible vocabulary rules out of `_wikidata_.orig.v` and places them into the `_wikidata_.v` file (Appendix B). So when I called `fcheck.py`, I provided both the .v file and the .f file, since `fcheck` calls on `predpar`.py which parses through and soley deals with the vocabulary rules.

After a bit of adjustment to `predpar.py`, I was eventually satisfied with the parsing of the vocabulary rules and the grammar dictionary it produced for the facts. The results from `predpar.py` may be found in Appendix C.

However, after the first initial trials of parsing the facts, I was realize that the vocabulary produced by mkvocab.py had inserted blank types (<>) in places where a merely a word token should be present. For example, nearly every language has a fact about it's official status of the form:

```
Arabic Official language in Egypt
```

However, the vocabulary rules failed to include "<> Official language in.." anywhere, and instead had " <> <> in…" which is incorrect. "Official" and "language" are not types and should not be types, since with this language data I decided that types should only represent languages and different language families, and the vocabulary rules in this case assumed them to be types. I encountered the same problem with facts of the form “Language codes ISO,” and “Language codes linguasphere.” After some testing with various parts of the .f file, I found that when only small portions of the fact file were processed by `mkvocab.py` or when the subject was used in place of the quote marker, then the vocabulary rules appeared on their own just fine, or but for some reason they refused to form when the data was entered all together. So I elected to modify the .v manually, just by replacing the second <> in “<> <> in…”, “ <> <> ISO…”, and “<> <> linguasphere…”, with the relevant (and correct) word phrases. I also found that vocabulary failed to place parentheses in some rules where they should be present, and at times had lopsided parentheses where in the facts they were complete, so I went ahead an corrected those as well. While this helped a great deal with the results of fcheck, this really was not the root of the problem.

As I mentioned earlier, the way I handle types for this data is that when processing the Wikipedia page, I set the first parent family of the language as the head of the language in question. Next, I rely partially on the automated vocabulary, which provides some roots, generating a partial tree. And finally in `fcheck.py`, I find the predefined rules in my fact file that represent the immediate parent relation, and add those to the Type Hierarchy. However, since we do not have an exhaustive list of all languages and all immediate parent nodes, the tree will not be a complete map of all the world’s language families as would happen theoretically with a complete and accurate dataset; so if I find that any parent node does not in fact have an entry of its own in the type tree, I add it as a root, so that there are more types available for parsing the facts.

However, the automated vocabulary relies a little too heavily on types (as I mentioned ealier), requiring them where they may not be applicable. For example, for the language entry Kashmir we have the following fact:

```
" Region Northwestern region of the Indian subcontinent
```

Which is seemingly straightforward enough. But if we look to a vocabulary rule we find:

```
*   <> Region <> of the Indian subcontinent
```

While the initial requirement for a type is fine and expected, and to match “any” type, the item in question still must be defined or noted as a type at some point, but “Northwestern region” is not defined anywhere in the fact file or vocabulary file. I even changed my definition of “any” type to include items not even defined as types, “Northwestern region” is still two words long, and there was no reason to mark it as a multi-worded alias, so it would still fail to parse because this restriction that is incorrect. Now knowing how much the vocabulary favors types, I would have perhaps structured the facts in a different manner but for now, I just changed my definition of “any” type to really include any word token, to get maximum parses.

Additionally, I attempted to address the issue of multi-worded aliases in `fcheck.py` since the lexer taking apart the facts only recognizes the subject of the fact as the first word, and not multiple words. In the predefined facts with type definitions, it is easiest to see if a subject may have multiple words, such as these subjects “Standard Tibetan,” “Kharia Thar,” and “Guatemalan Sign Language.” So it is often the case that the true subject which may have multiple words is stored in the type hierarchy, whereas the assumed subject which is only one word, does not, leading to perhaps a failed parse. And so when I came across multi-worded subjects in type definitions, I added them to the list of multi-aliases and the multi-alias dictionary, so that they could be better detected and a correct parse could be linked to the type. So now all types should link correctly to the recorded subject and multi-worded types may be accessed when checking types.

While these results may not demonstrate precise connections between languages or novel discoveries, it is the first step towards creating clean facts about different languages, so that other Factotum tools may be used to work more with this data to better organize and understand it. This is an essential and necessary part of the process if we wish to create a resource for everyone to use, running the facts and vocabulary back and forth against each other forces the researcher to gain understanding of the “distinctions, similarities, and relationships in the data,” (17) making the facts more and more reliable and consistent.

I also encountered issues when dealing with characters that are not in the normal ASCII characterset since I ended up using Python 2.7, and while this created some issues, I managed to work around them and switching to Python 3 would help resolve this in the future.

I have two versions of the precise results of `fcheck.py` provided in Appendix D and E. Appendix D contains the successful parses of facts and the facts that failed when I insisted that Type: ANY require that the item have been defined explicitly as a fact. Appendix E on the other hand contains the successful parse and failed facts of when Type: ANY did not require any base definition, and returned true for any token.

ii. Performance and Time measurements

When I ran `fcheck`, I also called the profiler `cProfile` to which shows results for both `fcheck` and `predpar`, telling us where they are spending the most time, and doing what exactly so that they may be improved in the future. The detailed results of cProfile may be found in Appendix F, but I have a snippet below. To clarify the headings: `ncalls` is the total number of calls (including self-recrusion) and if two numbers present, the first is actual number of calls and the second is the number of primitive calls; `tottime` is the time spent in this function (not counting sub functions), `percall: tottime / ncalls; cumtime:` total time spent in the function , and `percall: cumtime` /(primitive calls):

```
 ncalls    tottime  percall  cumtime  percall filename:lineno(function)
```

```
168312249   180.564    0.000  260.985   0.000 fcheck.py:141(isDescendant)
412399764   331.440    0.000  449.581   0.000 fcheck.py:273(isTerminalSymbol)
412371930   683.703    0.000 3803.405   0.000 fcheck.py:281(matchTerminalSymbol)
55668/27834 972.748    0.017 5342.630   0.192 fcheck.py:353(parseRD_Facts)
560341/26287 18.498    0.000   38.472   0.001 predpar.py:308(parseGrammar)
75881/11711  0.503    0.000    0.815    0.000 predpar.py:751(getTermSymbs)
ncalls    tottime  percall  cumtime  percall filename:lineno(function)

 13124      25.532    0.002   31.663    0.002 predpar.py:833(add_fcheckDictEDIT)
   47596    3.989    0.000   18.251    0.000 predpar.py:871(tokenize_pred_string)
        1    0.059    0.059    1.164    1.164 predpar.py:957(go_thru_file)
233789545  256.378    0.000 2726.503    0.000 re.py:134(match)
    13177    0.010    0.000    0.042    0.000 re.py:188(compile)
233802722  385.421    0.000 2235.280    0.000 re.py:229(_compile)
402173811  146.543    0.000  146.543    0.000 {method 'get' of 'dict' objects}
416899359  121.361    0.000  121.361    0.000 {method 'keys' of 'dict' objects}
168315565  154.538    0.000  154.538    0.000 {method 'replace' of 'str' objects}
```

In this snippet I have pulled out the functions that take up some of the most time, which is not surprising in some cases (such as the big recursive parsers), but in other cases it is not obvious how many times it is called or how long it takes. CProfile calculated the total time to run as “5485.638 CPU seconds” which is roughly 90 minutes, which makes since given there are around 4000 languages and all of their facts to go through, as well as the vocabulary rules.

We can see that we spend the most time in the fact checker parser, `parseRD_facts`, even without the sub-functions as shown with `tottime`, with `matchTerminalSymbol` coming close for time. Nearly tied for most called, including self-recursion are `isTerminalSymbol`, `matchTerminalSymbol`, and `{method 'keys' of 'dict' objects}`, which makes sense, given that the first two are within the function that the most time is spent in, and the last one is always being used in the parsers, and checking up on types and aliases. It is also interesting to note that a good amount of time is spent working in the `re` library matching and creating regular expressions for the vocabulary.

If these modules were to be modified for more efficient performance, a more efficient method for dictionary look up would be in order, as well as perhaps a faster way to match symbols in the parsing functions.

VI. Conclusion and Further Work

So as we can see, using Factotum requires one to be quite certain of one’s data, and though Wikipedia proved to be an excellent source in terms of breadth and quantity of facts from even a small Infobox, it is still crowd-sourced and is not as necessarily consistent or reliable as information that could be provided by linguists and experts in different languages. The creation of the vocabulary and it’s subsequent use as a

grammar against the facts attests to this, as I myself had to go in and do a few modifications, and still not all the facts were able to be completely parsed.
Though creating and parsing the vocabulary and data is only the first part in the process of building a language resource from Factotum, there is great potential in doing so, especially if the facts are contributed by more knowledgeable persons, and greater attention is spent on the vocabulary file to more precisely express the intent of these facts.

In addition to the sort of meta-data provided by the Infobox, it would be useful to include in facts for each language information about what the word-order is for the language (e.g. SOV, subject object verb), what sounds are present in the language, the typology of the language, and more details about the grammar.

With this project I demonstrated how one may easily collect data from Wikipedia, and proceed to have it to fit in Factotum and create a vocabulary for it. These steps, along with thinking critically about these facts to make them more consistent, modifying the vocabulary to better suit them, and continuing this cycle until they both work together perfectly, will help to create a foundation for other Factotum tools to build upon, so that all of this information on the languages may be better organized, accessed and understood.

What I learned from this collection of data is that even the most seemingly simple and rudimentary of facts found in the Wikipedia Infoboxes proved to be more complex than at first glance, which is in fact better if I am to demonstrate that one should use Factotum for more precise and intricate linguistic data. These facts that I chose to use proved to be difficult and unusual to express in Factotum fact form, and showed that the manner in which facts are expressed greatly effects the generated vocabulary, which in turn demonstrates quite clearly how reliable and well formed one's facts are. And so, with a clear and accurate data set, an equally robust vocabulary should be able to be produced, together forming an ideal resource for linguistic students and others to access clear information about different languages, while also lending itself to other Factotum tools which will allow researchers to further discover and explore the realm of languages.

Bibliography

Appel, Andrew W. *Modern Compiler Implementation in ML*. Cambridge: Cambridge UP, 1998.

Beazley, David M. *Python: Essential Reference 4th Edition*. Indianapolis, IN: Sams Publ., 2010

Berners-Lee, Tim. “Semantic Web Road Map” 1998

Chen, Peter Pin-Shan. "The Entity-Relationship Model-- Toward a Unified View of Data."*ACM Transactions on Database Systems* 1.1 (1976): 9-36.

Grune, Dick, Koen G. Langendoen, Ceriel J. H. Jacobs, and Henri E. Bal. *Modern Compiler Design*. Chichester: J. Wiley and Sons, 2001.

Teory, Toby J., Dongqing Yang, and James P. Fry. "A Logical Design Methodology for Relational Databases Using the Extended Entity-Relationship Model." *Computing Surveys* 18.2 (1986): 197-222.

Uzgalis, Robert C. “Factotum 90: A Software Assisted Method to Describe and Validate Data.”