An Investigation into the uses of Factotum and extending its capabilities
By Claire Abu-Hakima

ABSTRACT (4 sentences)
Factotum old
Improved/updated
Lack of ling db, would be super useful
With rudiment data set from wiki, showed its utility

I. INTRODUCTION (1 pg)

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. And so, it was in 1993 that Robert Uzgalis created Factotum, a software tool designed to help with these very issues.

The original program, described in "Factotum 90: A Software Assisted Method to Describe and Validate Data," would allow researchers to uniformly enter their data, and suggest patterns and connections between given facts, creating a sort of taxonomy for the different data. Additionally, Uzgalis made a singular format for entering data allowing it to be language independent, so that facts in any natural language may be used with Factotum.

Another advantage to using Factotum in addition to aiding the researcher store their collected data is the forced formalism of the data through the use of Factotum. The format in which the researcher must shape the data requires him to further think critically of 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.

Over the years progress in the development of Factotum has been minimal and the software has not been used how it was originally envisioned. However now, some seventeen years later, Factotum is being updated into Python from C 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. Additionally, I created a specific dataset for Factotum, a dataset that was 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.

However, given a lack of experts for now, 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; I will give further details on what I learned from this collection of data, as well as what I learned was possible to do with it in Factotum itself.

II. THE PROBLEM (1 page) / MY IDEA (2 pgs)

In this section I will now discuss in further detail what exactly it is I am trying to do with Factotum as well as my idea for a linguistic database.

As a student myself, I always found myself researching different languages online, gathering data from different books and papers, and frustrated at the lack of a single, qualified data base containing (at the very least) meta data for all the languages. I say qualified because presently there exists a sort of database of this nature: Wikipedia.

IV. THE DETAILS (5 pgs)

Now I will discuss the more intensive details of the workings of Factotum, how I wrote and used the vocabulary parser and fact checker, how I created my Lingdata dataset and what results did I get when running the data through Factotum.

HOW FACTOTUM WORKS

Though there is not much documentation on Factotum except for the original paper, I will attempt to give a clear description based upon it as well as from what I have asked and learned from Uzgalis.

- TERMS: marker, alias, entitites, subject, predicate, object, citation

-FACTS: what are facts, entering data/data format, automatic or manual

-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 the mkvocab module (provided by Uzgalis), it can be manually altered (or even created manually) to provide more accurate descriptions for the facts, though it is usually better to protocol to check for errors in the fact file first before

modifying the vocabulary. Human error is more likely in the fact file especially if the vocabulary has been automatically generated.

Additionally, the vocabulary itself has it's own format it must adhere to, which is confirmed by the vocabulary parser (predpar.py) that I wrote and will discuss in greater detail later on.

-predicates, types of rules, what does vocabulary signify, automatic or manual

VOCAB PARSER

One of the main bits of code that I wrote for Factotum was the vocabulary parser, which is contained in the module prepar.py. The ultimate goal of this parser is to check whether the rules contained within the given .v file adheres to the required format of vocabulary rules.

I accomplish this by first translating the designated format of the rules into a grammar which I represent using a dictionary called 'vocab_grammar', where the keys of the dictionary are the non-terminal symbols, and entries for each key represents the right-hand side (RHS) of the grammar rule with each token an item in a list, and where there are multiple RHS's, (for example Pred goes to := Phrase and -= Phrase, etc.), the entry is represented as a list of lists, so that our previous example would be like 'Pred' : [[':=', 'Phrase'], ['-=', 'Phrase']]. This is set as a global variable so that I can access it more readily when doing the parse.

Next I had to get the vocabulary rules from within the file into the appropriate format. And using the helper function 'go_thru_file' I use a similar method as used in the factotum_lexer module, where I read in the file line by line, identifying where the line is a continuation or a clean break and then storing it in a list of lists of the form [[subj, pred]…]. After that, we jum to the function FIRSTPASS, which iterates thru the facts and first tokenizes the predicate using the 'tokenize_pred_sting' function, separating the string pred according to either symbols that are significant in vocabulary rules (which I represent in a regular expression) or just words (separated by white space).

If a rule fails to tokenize, it is thrown out, otherwise we now attempt to parse it using the function PARSEGRAMMAR, the biggest
FIRSTPASS
SECONDPASS
PARSE GRAMMAR
UPDATETYPETREE
	CHECKTYPES/TRACEPATH
addFCHECKDICTEDIT
	GETTERMSYMBS/FINDINTREE
ADDNEWDICT

FCHECK

LINGDATA

Another significant portion of this project lies not merely in the code for the parser, but also in the data itself. Here I will explain why I chose to create my dataset from Wikipedia, what select bits of data I chose to scrape off the Wikipedia language pages, how I managed to scrape that data and how I transformed that raw data into usable Factotum facts.

A. WHY WIKIPEDIA

As I mentioned earlier in the paper, as a student of Linguistics I was quite surprised and frustrated by the lack of a single unified and legitimate source containing information or at least meta-data on all of the world's languages. 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.

So as I mentioned before, 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 crowd-sourced information, can yield some very useful and interesting results.

B. INFOBOX/WIKIPEDIA LANGUAGES

As I previously mentioned, during my studies I continually found myself using Wikipedia as a reference for mainly meta-data on languages. What I came to notice is

that 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 (note: all Wikipedia pages referenced in this project are in English). So for my Wikipedia test case, I decided to scrape every language page containing the Infobox, and to grab all the information within the Infobox and turn it into Factotum facts.

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.

LANGUAGE FAMILY – ONLY PULL IMMEDIATE PARENTS

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.

C. WIKISCRIPT.py / Scraping

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`.

A quick word on user arguments provided to wikiscript.py; as is protocol, the first [0th] argument is just the name of the program we are running. Next, we have an additional 3 optional arguments: sys.arg[1] is the designated slot to enter the file where you want to write the Factotum facts, sys.arg[2] is the designated slot to enter the file where the permalinks for all accessed pages will be written, and sys.arg[3] is the designated slot to enter a possible url to parse (note: this slot should only be used when you are just trying to access one url at a time for testing purposes). Now, none of these arguments are required: in `wiki_main` there is a quick check at the beginning to see if a file has been entered to write the permalinks, and if not, the file 'wikipedia_links.txt' is created and used. In `writeFacts`, there is also a quick check at the beginning to see if the file to write facts to has been provided, and if not the file 'wikidta.f' is created and used.

1. WIKIMAIN

So `wiki_main` serves as the 'main function' to the whole program. After the if-block checking for the filename, `wiki_main` calls `collectData` to grab all the links that contain the Infobox template (stores them in the list `links`). Note that the url which is necessary for `collectData`, *'http://en.wikipedia.org/wiki/Template:Infobox_language',* is hardcoded into `wiki_main` and stored in `startURL`.

Then only after collecting all possible links does it proceed to go through each link and grab the html source code for that given page as a string (using `pull_content`), storing it as a pair [link, source] in the list `linksplus`. It then goes through each `pair` in `linksplus` and calls `parse_wiki` to go through the source code, pull out the Infobox information and write information in the correct form to a designated fact file.

`Parse_wiki` returns the permalink of the page that was accessed (i.e. what version of the page), which is appended to the list `permaLinks`. Finally after going through all the pairs in `linksplus`, we go through each item in `permaLinks` and write the permalink to a file so that we have a record of what pages we accessed and so that if anyone wanted to recreate the data, they are able to do so.

2. COLLECT DATA

The first half of wikiscript greatly involves the use of the function COLLECTDATA. CollectData takes one argument, the starting url, which is the link to the page which explains the Infobox template for languages ('http://www.*en.wikipedia.org/wiki/Template:Infobox_language').* Next, using the function LINKSTOPAGE, we search for the link that will get us to the page that informs us of what pages link/use the Language Infobox template. LINKSTOPAGE is just a simple function which goes in and searches for the string 'whatlinkshere' and the first link following the term and returns it. Once it returns, I used the urllib2 protocol once again, and open this first page, storing the html source code in the string SRCE.

It is also important to note, that not all the language links are contained on this first page, in fact only 50 are contained on each page, and with around 4000 language pages using the Infobox, that means there are many many pages of links that collectData must go through. Each page of language links also contains a line, visible on the site, 'View (previous 50 | next 50)', which links us to either the previous 50 language links, or the next 50. In this case, we are interested in pulling out the 'next 50' link on every page until there are no more pages left, in which case there would be no 'next 50' link available to pull.

So I have a while loop that continues while NEXTLINK is not empty (eg there are still more pages of language links). Within this loop, we make sure to set LINKS as the empty set, then call the function GOTHRUPAGE taking in the argument NEXTLINK, returning a list of language links that is stored in LINKS, and returning the link to the next page and storing it into NEXTLINK (thus changing/updating NEXTLINK). Then, LINKS gets added onto the end of MOARLINKS, which is the cumulative list of links (while LINKS is just the links from the current page), and the loop continues. Finally when the loop terminates and there are no more links to fetch, COLLECTDATA returns MOARLINKS to WIKIMAIN.

a. GO THRU PAGE

GOTHRUPAGE is the function that takes in the argument LINKSURL, the URL for one of the pages contain links of language pages, and manages to pull all the links as well as the link for the next page. It uses the urllib2 protocol once again to pull the source code, storing it in the variable LISTHTML. First thing after fetching the source code, it

calls GETNEXTPAGE and stores the resulting value in the varible NEXTURL. After this, I shorten the LISTHTML source code, so it just contains the language links. I search for the first instance of 'mw-whatlinkshere-list' as the starting index and ending with the last instance of 'View' (since we've already passed the first instance with 'mw-whatlinkshere-list').

Next I call GRABLINKS to grab all the links on that page, and store the returned links into the list LINKLIST. GOTHRUPAGE then returns a tuple, the list of links on the page LINKLIST and the next link, NEXTURL.

`b. getNextPage` is a small helper function that basically searches for 'View (previous 50 | next 50)' and pull it apart to access the link. It searches the source code for the first instance of 'View (', then updates the string (locally) so that it may then search for the first instance of '|' (since it is definitely after any previous links). Finally if there is an immediate link present, it pulls it out and returns it as the nextpage link, and if there is no link present (i.e. on the last page) then I return the empty string instead.

c. GRAB LINKS is a recursive function, which takes in a single argument, the string BLOCK (the source code of a single page of language links) and pulls the links one by one and then returns LLIST, a list of all the links. It searches for the first instance of the string '/wiki/' ; if /wiki/ is not present, the function returns LLIST (which should be empty), since not finding /wiki/ means the function has reached the end of the block. But if '/wiki/' *is* present, I update BLOCK to begin with '/wiki/.' Next, the first instance of double quotes ('\"') is searched for, which would indicate the end of the link. If no quotes are found, the function returns. But if the quotes are found, it means the link has been found, and we may store it in the variable LINK, and add `'http://www.en.wikipedia.org'` to the beginning of it. However, if the link contains colon ':', I have chosen not to include it because those pages follow slightly different formats and were not all pages about just the language itself, so in these cases, I set LINK to the empty string. BLOCK is then updated, beginning right after link. Then, GRABLINKS is called again, with the updated BLOCK as the argument, with the results stored in the variable RES. Once GRABLINKS returns, if LINK (if not the empty string) is added to LLIST, and if RES is not the empty set, it is also added to the end of LLIST. GRABLINKS then returns LLIST back to GOTHRUPAGE.

3. So after COLLECTDATA has returned and we have gone through the loop pulling content for all of the links (using PULLCONTENT, another simple helper function, this one that just calls the urllib2 protocol to request the page of a given link and then returns the source code), the second half of the program begins, with the iteration through the links using PARSEWIKI.

PARSEWIKI takes two arguments, URL and HTMLSOURCE, and takes care of several things. First, it pulls the version ID of the page using GETID. At the end of PARSEWIKI the URL gets updated (and returned) to reflect the version ID, so that anyone may go back to the correct page from which the information was pulled. Next, it narrows down the HTMLSOURCE to just contain Infobox information. Then this narrowed down HTMLSOURCE gets cleaned up using REMOVETAGS, which removes everything between angle brackets (< and >), just leaving the plain text raw data, and CLEANUP, which gets rid of excess newlines and adds in markers where whitespace is no longer an adequate marker of separation.

Then PARSEWIKI takes the string and calls SPLITSECTIONS on it, which then returns three strings: HEAD, OFFICIAL, and CODES. Once these are returned, if they are not the empty string, their respective parsing functions are called: PARSE_MAINSECTION, PARSE_OFFICIALSTATUSSECTION, and PARSE_LANGUAGECODESECTION.

As I mentioned in the previous section about the Infobox, there are 3 sections, with only the first main one being required and the other two ('Official' and 'Language Codes') being optional. SPLITSECTIONS takes in the rawest form of the html Infobox data, and pulls it apart by assuming the first section is at the beginning, then searches for the terms 'Official status' and 'Language codes' to demarcate the edges of the other sections, and three separate strings are returned; if a section is not present then simply an empty string is returned in its stead.

Each section, has its own parsing function, PARSE_MAINSECTION, PARSE_OFFICIALSTATUSSECTION, and PARSE_LANGUAGECODESECTION respectively, but each performs the same task. There are three global variables, HEADINGS_MAIN, HEADINGS_OFF, and HEADINGS_CODE, each containing all possible headings for their respective section. Each parsing function first goes through their headings, seeing which are present in the infobox section raw data, recording in one list the heading, and recording in another, the index/point of the heading in the raw data. Knowing what headings are present and where, the parser then separates the data into separate entries in a dictionary of facts, using CLEANUPFACT along the way to remove citations, do some comma splicing, and fixing excess entries. Finally the parser uses the function WRITEFACTS to write all the entries in the fact dictionary to a file in the designated Factotum form.

WRITE FACTS is a relatively simple function, since the facts are passed to the function in a dictionary, and the subject is already designated. There is a global variable ('started') which determines if the subject should be included or not, and so the subject or marker is written, followed by the heading and it's entry. There is a special case for 'Language family' where type markings need to be included, and another special case for the Alias symbol '->.' After all the facts are added to the file, WRITEFACTS returns.

D. Factotum fact form

E. what additional information could be added to infobox/data set
-word order
-what sounds present in language
- typology
-grammar (mood, tense, gender, case, etc)

#resulting vocabulary
resulting profile

RESULTS
MEASUREMENTS

V. RELATED WORKS (1-2 pgs)

VI. CONC AND FURTHER WORK (1/2 pg)

Sources