**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**

**GLOBAL DICTIONARIES**

**VOCAB\_GRAMMAR**

‘Predpar.py’ is one of the most significant and essential modules that I have written for this project and for Factotum; predpar 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 ‘vocab\_grammar.’ ‘Vocab\_grammar’ effectively transliterates the format given in the original paper, into a form (that I have decided upon) which 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 | -= Phrase | ~> Phrase | =>> Phrase | ? ( Cond ) Then | Phrase

The entry in the dictionary looks like this:

**Vocab\_grammar = {... ‘Pred’: [ [‘:=’, ‘Phrase’],**

**[‘-=’, ‘Phrase],**

**[‘~>’, ‘Phrase’],**

**[‘=>>’, ‘Phrase’],**

**[‘?’, ‘(‘, ‘Cond’, ‘)’, ‘Then’][[1]](#footnote--1)\*,**

**[‘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.

**REGEX\_DICT**

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:

**Regex\_dict = {...**

**‘Typename’: re.compile(*'^[-\_0-9a-zA-Z\']+$'*),**

**...}**

**GLOBAL LISTS**

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 (in the case of Factotum, just ‘Words’ and ‘Msg’) 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.

**FLOW**

**PARSE VOCAB**

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. Go\_thru\_file() takes in the file from command line (if the user fails to provide a file name, an error message will appear and predpar.py will quit) and uses a similar method as used in factotum\_lexer.py, reading in the file line by line, identifying where the line is a continuation or a clean break and then adding it (the vocabulary rule) in the form of [subject, predicate][[2]](#footnote-0) on 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’]**

**FIRST PASS**

Next, parse\_vocab passes these newfound rules (stored in ‘facts’) to the function first pass, whose resultant tuple 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 (tokenize\_pred\_string). 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(rule\_pred).

Just a quick note, rules that are type definitions have a slightly different form, and so we must slightly modify rule\_pred. For example: if the rule in the .v file has the form: mortal [beings], it has been stored in the list of rulefacts as [‘mortal’,‘[beings]’] and then rule\_pred is set as only [‘[‘, ‘beings’, ‘]’], ignoring the subject which is unnecessary for parsing the other rules. So in this case, we simply insert ‘mortal’ into rule\_pred so that it now it will parse the type definition correctly as: [‘mortal’, ‘[‘, ‘beings’, ‘]’].

I will go into parseGrammar in greater detail a little further on, but I will point out what is significant in parseGrammar on the first pass. First, l left out purposely earlier an explanation of the global variable ‘PassedThru’ which is set to 0 at the beginning of firstPass, and set to 1 in parse\_vocab after firstPass returns. As I mentioned earlier, the first pass is necessary so that we may build up a type tree that we can check against in the second pass. So since this is the first pass, we need to build up the tree, 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().

**UPDATING THE TYPE TREE- FIRST PASS**

This function takes in three arguments, the token we are currently checking (token), the rule pred list (ruleList), and the parse tree that is being built in parseGrammar (tree). If the token is anything but ‘Typedef’ then nothing happens and the function simply returns. However, if it is ‘Typedef’ (thus indicating that the rule is a type definition), we are going to add it to the global ‘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).

So update\_TypeTree sets new\_type to equal NODE (which is always going to be the first item in the list, and thus ruleList[0]). If ruleList (the type definition in question) is only of length three, we know that it must be the definition of a root node. So we just add to TypeTree new\_type as the key, with the entry set as True, ‘ROOT’ as follows[[3]](#footnote-1):

**TypeTree[new\_type] = [True, *'ROOT'*]**

However, if the length is not 3, then we do not have a root node and have a slightly different method for adding it to the type tree. First we pull out the parent node and store it in the variable ‘head’ and pull the subtype/node in question and store it in the variable ‘subtype’. Next we check if ‘subtype’ is already present in the keys of TypeTree, because we do not allow for multi-inheritance. So if it is already present, then we do have a case of multi-inheritance and do not add this new definition to TypeTree, and just instead print an error message. If it is not present in the TypeTree keys, we then add it to TypeTree and return back to parseGrammar:

**TypeTree[subtype] = [False, head]**

FIRSTPASS\_RETURN

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])**

Also, since on the second pass we need to be checking PhraseName, if the rule has parsed successfully, there is a quick check if the rule contains ‘-=’ (indicating a phrase rule), and then adds it to the global list of phrase names PhraseList. And if the rule has failed to parse, it gets appended to the list of failed rules.

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]**

**PARSE\_VOCAB CONT’D**

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.

**CHECKTYPES**

I will briefly discuss check\_types(); check\_types() iterates through the keys of TypeTree twice, first finding the paths from the type to the root, and then identifying which ones failed to reach the root, removing those from TypeTree, and adding them to the return list of types to be deleted from the parsed\_rule list back in parse\_vocab.

The first iteration loop (searching for paths) creates an empty mini dictionary, and then checks for entries in TypeTree that have the Boolean set as ‘False’ (as we know from update\_ParseTree, root nodes are already set as ‘True’ when added to the tree). When one is found, **tracePath**() is called, passing the arguments subtype (the key we are currently examining in TypeTree) and mini (the mini dictionary). TracePath() is a recursive function which records the path (in a list) from the subtype a root node, while also checking for any loops in the path by recording already traversed nodes (subtypes) in the mini dictionary, and returns the path list if the root node was reached successfully or the empty list if there was a loop or dead end along the way.

Upon the return of tracePath, check\_types goes through the returned path (if it is the empty list, check\_types just continues iterating through TypeTree), and for each type branch listed in the path (including the type we were originally checking), looks up the entry in the TypeTree and changes the Boolean to ‘True’ and appends the type to TypeList: if the type is in the path to the root node, this must mean that this type can obviously reach the root node as well. Once check\_check types is done with this loop, it then goes through and identifies which items in TypeTree still contain a ‘False’, remove them from the type tree and add them to the deletion list (which is ultimately returned).

**PARSE\_VOCAB: SECOND PASS/RETURN**

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 new\_dict and fcheck\_dict (which may be used to check facts in fcheck.py) using the functions add\_new\_dict and add\_fcheckDictEDIT respectively, which I will discuss later. Finally, after all the rules have been passed through this second time, parse\_vocab (and predpar.py) returns 5 things which will ultimately be used in fcheck.py:

**return (second\_parsed\_rules, failed\_rules, new\_dict,**

**TypeTree, fcheck\_dict)**

**PARSE GRAMMAR**

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 (as is the case of ‘Words’ and ‘Msg’), 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 [‘**problem’, …]** (assuming we have already validated the type and matched the pattern), the pair would appear as:

**[‘Typename’, ‘problem’]**

If we had the token ‘**Words’** and the remainder of the rulePred as **[‘is’, ‘an’, ‘NP-complete’, ‘problem’]** the pair would look like:

**[‘Words’, [‘is’, ‘an’, ‘NP-complete’, ‘problem’]]**

Now that I have shown what parseGrammar is supposed to return, I will explain how it works.

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! Now remember 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 (only ‘Words’ and ‘Msg’). 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.

**AFTER SECOND PASS – CREATING A NEW 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 briefly discuss now the two functions that are used to created these dictionaries: add\_new\_dict and add\_fcheckDictEDIT respectively.

**ADD\_NEW\_DICT**

Add\_new\_dict takes in three arguments, the subject of the rule that was just parsed (subj), the parse tree returned by parseGrammar for the rule predicate, and ‘dict’ the dictionary we are creating/have been creating after parsing each rule, in this case, new\_dict. The function returns an “entry” for new\_dict, which is the revised entry for the subject given, that gets inserted into new\_dict after add\_new\_dict returns, thus either creating a new entry for the subject or writing over it and updating it in the process with this modified version.

Add\_new\_dict checks if the subj is an already present key in the passed in dictionary: if it is, the ‘subj\_entry’ is set to equal the entry from the dictionary (which is a dictionary), and if not, subj\_entry is set as an empty dictionary. Next, add\_new\_dict iterates through each pair in the parse tree passed in. If the token in the pair (the first item in the pair) is already present in the entry (subj\_entry), then add\_new\_dict has to check to make sure that the right hand side of the pair (the match) is not already present in the entry for the token to avoid repeating rules. In both cases the following steps are very similar so I will try to explain both.

In both cases, add\_new\_dict then checks for any angle brackets ( ‘<’ ,’>’) in the right-hand side of the pair (second item) using the function checkBrackets. While angle brackets are permissible in writing the vocabulary rule to indicate objects, facts will not have brackets and so this helper function notes where we should remove the brackets so that the grammar may be better ported over.

Next, if it was the case that indeed the token is already present in the subject entry, add\_new\_dict, the entry for the token is iterated through, and if the right-hand side rule from the pair is already present, then it moves on to the next pair in the parse tree. Otherwise, if it iterates through the entire entry without a match, the right-hand side rule is then added to the token entry. (if it was not the case that the token was present in the subject entry, the token is simply added as a key with the right hand side as it’s entry). Once every pair in the parse tree has been gone through, add\_new\_dict returns the resulting subject entry.

In the case where there is no subject entry, a new empty dictionary is created as the subject entry, and all the same steps as above are followed, but using the newly created entry instead of a subject entry from the dictionary passed in to the function.

And as I mentioned earlier, when add\_new\_dict returns with the entry, new\_dict updates the entry of the subject with this returned mini dictionary from add\_new\_dict.

**ADD\_FCHECKDICTEDIT**

So after I had done all of the above with new\_dict, I realized that that dictionary was not an effective way to store the grammar rules for checking the facts, and additionally subjects of the vocabulary are unnecessary, and that only rules beginning with ‘:=’ are needed to determine the syntax of the facts.

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'*]].** If the pair is not present, then nothing further happens and the function simply returns. However, if the pair is indeed present, then we need to build up the fcheck dictionary, by first removing this pair (since ‘:=’ is not actually a symbol found in facts).

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 (atree) in turn calls ‘findInTree,’ which looks up the first pair in the remainder parse tree with nonterm as the first item in the pair, and returns the second item (the right-hand side).

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 findInTree). 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.

Right before add\_fcheckdictedit returns, the end result looks much different than that of add\_new\_dict, where the former has transformed the parse tree into a list of literals, so that it is easier to check the facts. For example, the parse tree of the rule:

**‘:= <problem> transformed to <problem> polynomially’**, appears as:

**[['Start', ['Pred']],['Pred', [':=', 'Phrase']], ['Phrase', ['Obj', 'Words', 'Phrase']], ['Obj', ['<', 'Typename', '>']], ['Typename', 'problem'], ['Words', ['transformed', 'to']], ['Phrase', ['Obj', 'Words']], ['Obj', ['<', 'Typename', '>']], ['Typename', 'problem'], ['Words', ['polynomially']]]**

gets transformed to the following after add\_fcheckdictEdit:

**Phrase: [['Type: problem', 'transformed', 'to', 'Type: problem', 'polynomially']]**

**RETURN PARSE\_VOCAB**

And so by the end of parse\_vocab, new\_dict and fcheck\_dict are 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.

**FCHECK**

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 to 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 eac 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.

**FIRST PASS**

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 CHECK\_PREDEF(fact).

First\_pass loops through the facts, one by one, tokenizes 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.

-check type hierarchy (no loops/all reachable)

-remove bad types

- go through remaining facts: search for instances of multialiases: and combine them into a single token (otherwise they will be spread out)

-return facts, and failed

SECOND PASS:

In fact\_checker loop again, calling parse\_facts and adding into the list of parsed facts or list of failed facts

PARSE\_FACTS

-Extremely similar parser as in predpar: which was the point

-if token in dictionary keys: recursively call parse\_facts, otherwise check for regex—but we have special cases now: check if token is/contains:

- TYPE: use isDescendant on local[n], update n if true, if have reached end of rtuple, return tree and local (if not descendant – break)

-LABEL AND TTYPE (list):

-LABEL 🡪 func checkLabel

-TTYPE 🡪 func checkTtype

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.

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

Appendix: all modules, test.v, res predpar from test.v, res predpar lingdata?

1. \* 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: ‘\?’, ‘\(‘, ‘\)’ [↑](#footnote-ref--1)
2. 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). [↑](#footnote-ref-0)
3. Note that when the root node is added, new\_type is also added to the global TypeList, since root nodes are always reachable (bool is True) while other types don’t get added to TypeList until after the path has been checked. [↑](#footnote-ref-1)