

Projet Pensées Profondes

Final report



Fundamental Computer Science Master's Degree 1
September 2014 – December 2014

Adviser: Eddy CARON

Marc CHEVALIER
Raphaël CHARRONDIÈRE
Quentin CORMIER
Tom CORNEBIZE

Yassine HAMOUDI
Valentin LORENTZ
Thomas PELLISSIER TANON

Contents

Introduction	3
1 State of the art	5
2 Data model and communication	8
2.1 Data model	8
2.1.1 Abstract structure	8
2.1.2 Examples	10
2.1.3 Software point of view	11
2.2 Communication	13
2.3 Evolution of data model	14
3 Core	15
3.1 Core	15
3.1.1 Communicating with modules	16
3.1.2 Recursion	16
3.1.3 Optimizations	17
3.2 Libraries for modules	17
4 User interface	18
4.1 Logging	19
5 Question parsing	20
5.1 Grammatical approach	20
5.1.1 Methods	20
5.1.2 Results and discussion	25
5.1.3 Future work	27
5.2 Reformulation: a learning approach	27
5.2.1 How it works	27
5.2.2 Learning	30
5.2.3 Implementation	30

5.2.4	Future work	30
5.3	Machine Learning: Window approach	31
5.3.1	Data set	32
5.3.2	Results	33
5.3.3	Improvements of the algorithm	34
5.3.4	Conclusion	35
6	Back-end	36
6.1	Wikidata	36
6.1.1	Overview	36
6.1.2	APIs	36
6.1.3	Implementation	37
6.1.4	Future work	38
6.2	CAS	39
6.2.1	Input-output	39
6.2.2	Structure	39
6.2.3	Parsers	40
6.2.4	Future work	40
6.3	Spell Checker	41
6.4	OEIS	41
7	Performances	42
7.1	Overview	42
7.1.1	Keyword questions	42
7.1.2	English questions	42
7.1.3	Math questions	43
7.1.4	Weird questions	43
7.2	Comparison with existing tools	43
7.2.1	Nested questions	43
7.2.2	Conjunction questions	44
7.3	Limitations	44
	Conclusion	44
	A Question parsing – Triples tree	47
	B Question parsing – Grammatical rules analysis	48
	C Wikidata API examples	50
C.1	Retrieve entities	50
C.2	Retrieve entities	51
C.3	Do search with WikidataQuery	51

D CAS – C_{AL}C_{AS}^H functions	52
D.1 Infix operators	52
D.2 Functions	52
D.3 Constants	54

Introduction

The *Projet Pensées Profondes* (PPP) provides a powerful, modular and open-source application to answer questions written in natural language. We developed an eponymous set of tools that accomplish different tasks and fit together thanks to a well defined protocol.

These various tasks include data querying (using the young and open knowledge base Wikidata), question parsing (using machine learning and the *CoreNLP* software written by Stanford University), requests routing, web user interface, feedback reporting, computer algebra, etc.

A deployment of these tools, *Platypus*, is available online:

<http://askplatyp.us/>

The official website of the project is:

<http://projetpp.github.io/>

This report will firstly introduce the modular architecture of the project and the datamodel used for communication between modules.

Then, all the modules actually developed in the PPP will be presented in details. Most of them are part of the online deployment, *Platypus*.

Chapter 1

State of the art

Our project is about *Question Answering*, a field of research included in *Natural Language Processing* (NLP) theory. NLP mixes linguistic and computer science and is made of all kinds of automated techniques that have to deal with natural languages, such as French or English. For instance, automatic translation or text summarization are also parts of NLP.

In Natural Language Processing, sentences are often represented in a condensed and normalized form called *(subject, predicate, object) triple representation*. For example, the phrase “Fleming discovered penicillin.” will be represented by the triple (penicillin, discoverer, Fleming). Two triples can be associated to “The president was born in 1950 and died in 2000.”: (president, birth date, 1950) and (president, death date, 2000). This representation has been formalized into the *Resource Description Framework* (RDF) model.¹ It consists of a general framework used to describe any Internet resource by sets of triples. The popularity of this representation and its expressiveness have convinced us to use it in our data model.

Many algorithms have been developed since fifty years with the objective of understanding the syntax and the semantic of sentences in order to extract (subject, predicate, object) triples. Two popular graph representations are widely used:

- parse structure tree. It tries to split the sentence according to its grammatical structure. Figure 1.1 presents the parse structure tree of the sentence *Brian is in the kitchen*. For instance, *Brian* is part of the noun phrase (NN) and is a proper noun (NNP).
- dependency tree. It reflects the grammatical relationships between words. The dependency tree of *Brian is in the kitchen* is displayed in figure 1.2. The relation *nsubj* means that *Brian* is a nominal subject for *is*.

Existing libraries, such as *NLTK*² or *StanfordParser*³ provide powerful tools to extract such representations.

We did not find a lot of detailed articles exposing procedures to get triples from the grammatical structure. For instance [RDF⁺07] tries to perform the extraction using the parse structure tree. However we believe the dependency tree could be more appropriate since it is more focused on the relations between the words than their roles in the sentence. Part 5.1 presents an algorithm we have designed using the dependency tree representation.

We have also observed a growing use of machine learning techniques in NLP that try to automatically learn triples extraction rules instead of building intermediate grammatical representations. We will use such machine learning algorithms in parts 5.2 and 5.3.

The algorithms we focused on can handle factoid *wh*-questions (i.e. closed-ended questions such as *What is the capital of France*) and nominal sentences (e.g. *capital of France*).

¹<http://www.w3.org/TR/rdf11-concepts/>

²<http://www.nltk.org/>

³<http://nlp.stanford.edu/software/lex-parser.shtml>

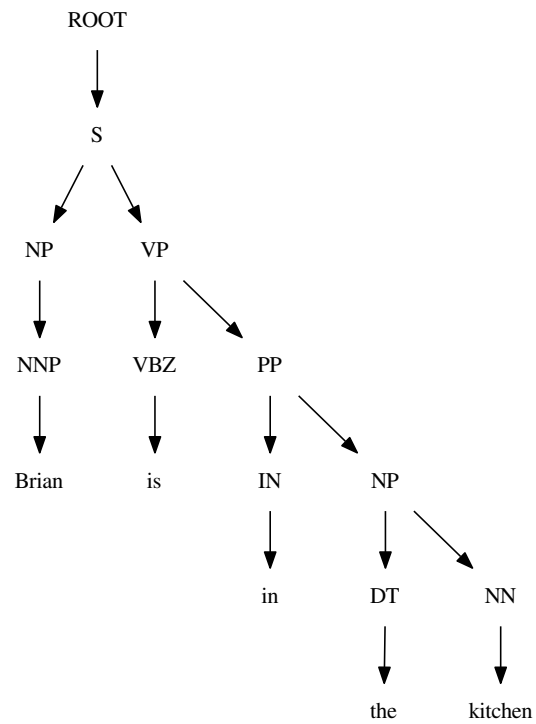


Figure 1.1: Parse tree of *Brian is in the kitchen*.

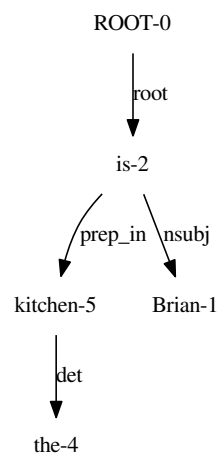


Figure 1.2: Dependency tree of *Brian is in the kitchen*.

Finally, some existing tools are very close to our goal.^{4,5,6,7,8} They allow us to have a clear idea of the state of the art, and what performances in question answering we can expect.

⁴<http://quepy.machinalis.com/>

⁵<http://www.ifi.uzh.ch/ddis/research/talking.html>

⁶<https://www.apple.com/fr/ios/siri/>

⁷<http://www.wolframalpha.com/>

⁸<http://www.ibm.com/watson/>

Chapter 2

Data model and communication

We describe the choices we did about representation of the data and communication between modules. These choices are described precisely in the documentation¹ of the project.

2.1 Data model

First, we present the data model. Its aim is to represent in a well formatted structure all data submitted to or generated by the modules. It must be able to contain raw data (as the input sentence or an output description) as well as structured data (as a parsed question). We will describe the abstract structure without detail of implementation and justify this choice.

2.1.1 Abstract structure

The data model is very close to the first order logic. The structure is defined by induction.

A basic element is called **resource**. A *resource* may represent anything in the world, a string, an integer, coordinates, a function, etc.

A **list** is a finite ordered sequence of *resources*. We identify the *list* $[a]$ containing only one element and the resource a . The use of this identification will be explained later. Moreover, a *list* must have no duplicates. For instance $[a,a]$ is not a list and need to be simplified as $[a]$. We can see *lists* as finite totally ordered set.

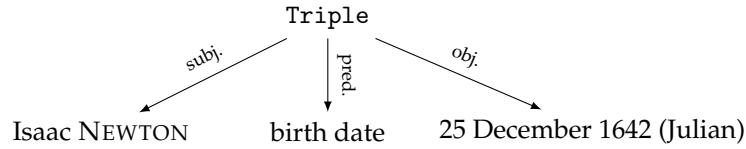
The other atomic elements are the boolean values **true** and **false**. They are not *resources* as previously defined. Nevertheless, we admit *resources* "true" and "false" but we cannot apply logic operation on them.

A **triple** is a 3-tuple containing in this order:

- a subject: what the triple refers to,
- a predicate: the relation between the subject and the object,
- an object: what the property of the subject refers to.

For instance, we can represent the sentence "Isaac NEWTON was born on the 25 December 1642 (Julian)" by the triple:

¹<https://github.com/ProjetPP/Documentation/>



In other terms, we can consider that the description of Isaac NEWTON has an entry "birth date" whose value is "25 December 1642 (Julian)" or any equivalent value.

It is reasonable to associate to each triple a boolean value depending upon whether the triple is true or not. Generally triples contain *lists* with more than one value. So we define the node *full triple*:

full triple: $\text{list}^3 \rightarrow \text{bool}$

and we define its value as $(a, b, c) = \bigwedge_{i=1}^{n_a} \bigwedge_{j=1}^{n_b} \bigwedge_{k=1}^{n_c} (a_i, b_j, c_k)$ where $a = [a_1, \dots, a_{n_a}]$, $b = [b_1, \dots, b_{n_b}]$ and $c = [c_1, \dots, c_{n_c}]$.

This structure is needed to check if a statement is true. That naturally corresponds to yes/no questions.

A kind of natural yes/no questions which are not handled by *full triple* are existential questions: "Is there a... ?". To represent them, we use a node **exists** (also denoted by $\exists$):

exists: $\text{list} \rightarrow \text{bool}$

and the evaluations is $\text{exists}(l) = \begin{cases} \text{true} & \text{if } l \neq \emptyset \\ \text{false} & \text{if } l = \emptyset \end{cases}$. This node checks whether the list is non empty.

The usual boolean operations: and ($\wedge$), or ($\vee$), not ($\neg$).

and, or, not : $\text{bool}^2 \rightarrow \text{bool}$

The usual operations on lists: union, intersection, difference. They do not keep the order of the *lists*.

union, intersection, difference : $\text{list}^2 \rightarrow \text{list}$

We also allow *incomplete triples*. An incomplete triple is a triple with a "hole": we know only two values and we want the list of all values which satisfy the triple. There is 3 types of required triples: for subject, predicate and object. They have the type:

incomplete triple: $\text{list}^2 \rightarrow \text{list}$

and the evaluations are

- for subject: $\text{incomplete triple}_{\text{subject}}(a, b) = \text{list}(\{x \mid (x, a, b)\})$ denoted by $(?, a, b)$,
- for predicate: $\text{incomplete triple}_{\text{predicate}}(a, b) = \text{list}(\{x \mid (a, x, b)\})$ denoted by $(a, ?, b)$,
- for object: $\text{incomplete triple}_{\text{object}}(a, b) = \text{list}(\{x \mid (a, b, x)\})$ denoted by $(a, b, ?)$.

We define two additional nodes: *sort*, *first* and *last*.

sort: $\text{list} \times \text{resource} \rightarrow \text{list}$

This function *sort* takes a list and a predicate and sorts the element of the list in the increasing order according to the value returned by the predicate. More formally, let $l = [l_1, \dots, l_n]$, p a predicate and a_i such that $\forall i \in \llbracket 1, n \rrbracket, (l_i, p, a_i)$. Then, $\text{sort}(l, p) = [l_{\sigma(1)}, \dots, l_{\sigma(n)}]$ where $\sigma \in \mathfrak{S}_n$ and $\forall (i, j) \in \llbracket 1, n \rrbracket^2, \sigma(i) < \sigma(j) \Leftrightarrow a_i \leq a_j$. This definition assumes that the predicate p associates to the list a totally ordered set.

The function *first* returns the first (smallest if the list is ordered) element of a list.

first: list $\rightarrow$ resource

The function *last* returns the last (greatest if the list is ordered) element of a list.

last: list $\rightarrow$ resource

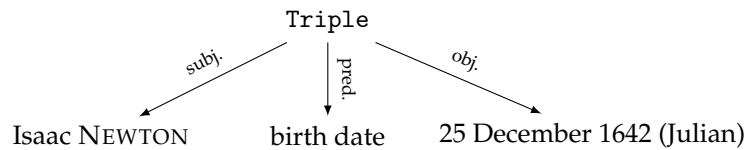
These nodes are useful when a triple can return a list with more than one element and we need only one. The difficulty is to find what is the correct order to choose.

A tree which represents a sentence is called a **normal form** of this sentence. There may be several normal associated to a sentence (especially because a question can be ambiguous).

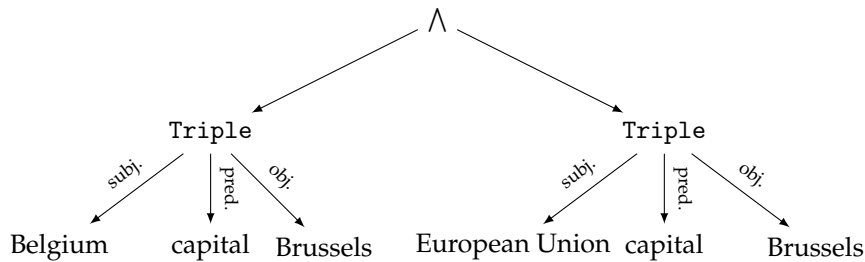
2.1.2 Examples

To clarify the previous definitions, here are some examples of questions followed by their normal form.

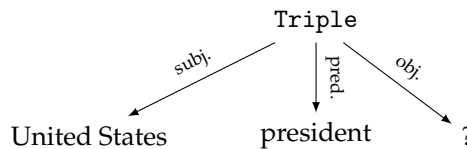
"Was Isaac NEWTON born on the 25 December 1642 (Julian)?"



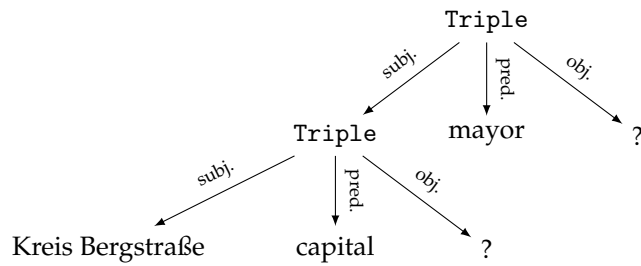
"Is Brussels the capital of Belgium and the European Union?"



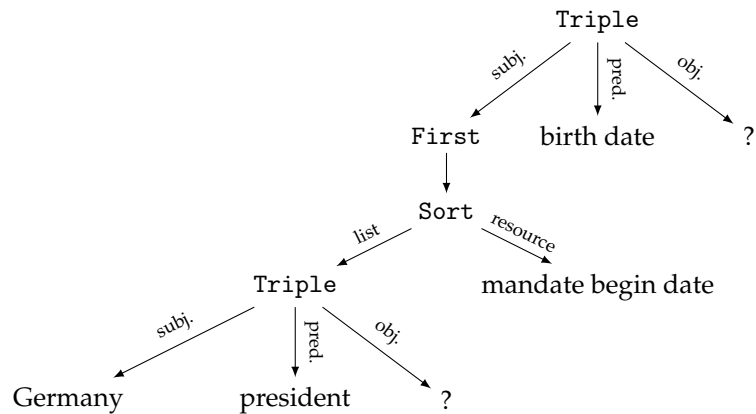
"Who is the presidents of the United States?"



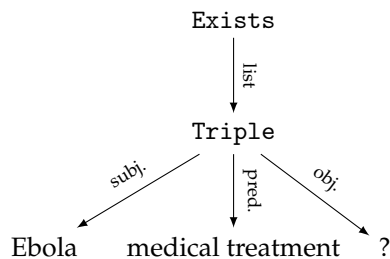
"Who is mayor of the capital of Kreis Bergstraße?"



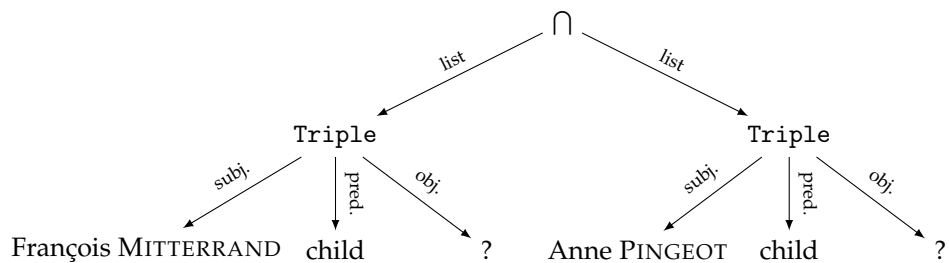
"What is the birth date of the first president of Germany?"



"Is there a medical treatment for Ebola?"



"Who are the children of François MITTERRAND and Anne PINGEOT?"



2.1.3 Software point of view

All normalised structures of the PPP are JSON-serializable, i.e. they are trees made of instances of the following types:

- Object
- List
- String
- Number
- Boolean
- Null

We chose to represent all normalised data as trees. We use the previous nodes and we have a *missing* node to represent the '?' in incomplete triples.

The node *resource* to represent the previous abstract node *resource* and *bool*. Moreover, we identify the *resource* "true" and the boolean value *true* and the same for "false" and *false*. This identification is not problematic for type because we assume modules create only well-formed expressions.

We add to the nodes we defined above some new features:

- a sentence node: a question in natural language like "Who is George Washington?",
- we can precise a type to *resource* and missing nodes. For example, if we choose as range "time" to the missing node ? in the triple (George Washington, birth, ?) this triple with hole can only return time points.

For example, the work of the question parsing module is to transform

```
{
  "type": "sentence",
  "value": "Who is George Washington?"
}
```

into

```
{
  "type":
    "triple",
  "subject":{
    "type": "resource",
    "value": "George Washington"
  },
  "predicate":{
    "type": "resource",
    "value": "identity"
  },
  "object":{
    "type": "missing"
  }
}
```

The basic value types are:

- string,
- boolean,

- time (encoded according to ISO 8061²),
- math-latex (for mathematical formulas written with $\text{\LaTeX}$),
- geo-json (for geographic data, based on GeoJson³).

2.2 Communication

Modules communicate with the core via HTTP requests.

The core sends them a JSON object, and they return another one.

The basic idea is that the core iterates requests to modules, which return a simplified tree, until the core gets a complete response, ie. a tree without any 'missing' node.

During these exchanges, we keep a trace of the different steps between the original request and the current tree. The structure of a trace is a list of such trace items:

```
{
  "module":
    "<name of the module>",
  "tree":{
    <answer tree>
  },
  "measures":{
    "relevance": <relevance of the answer>,
    "accuracy": <accuracy of the answer>
  }
}
```

The measure field contains two values: relevance and accuracy.

- accuracy is a self-rating of how much the module may have correctly understood (ie. not misinterpreted) the request/question. It is a float between 0 and 1.
- relevance is a self-rating of how much the tree has been improved (i.e. progressed on the path of becoming a useful answer). A positive float (not necessarily greater than 1; another module might use it to provide a much better answer).

We do not provide a method to compute these values. We have to trust every modules.

This form allows each module to access to the previous results, particularly to the request of the user. The objects for request and response contain some extra data, such as the language used.

We can use these measure to chose the best answer to display it at the first position in the UI. Modules can use them too. For instance, if there are two modules for question parsing, a module will choose one of the two answers based on these marks.

The data model has been implemented in a nice set of objects in both Python⁴ and PHP⁵ in order to help the writing of modules.

We could define a linear representation for the trace, using the representation of the datamodel, but it is not relevant. Indeed, this information will never be printed on the user interface.

²<http://www.iso.org/iso/home/standards/iso8601.htm>

³<http://geojson.org/>

⁴<http://github.com/ProjetPP/PPP-datamodel-Python/>

⁵<http://github.com/ProjetPP/PPP-datamodel-PHP/>

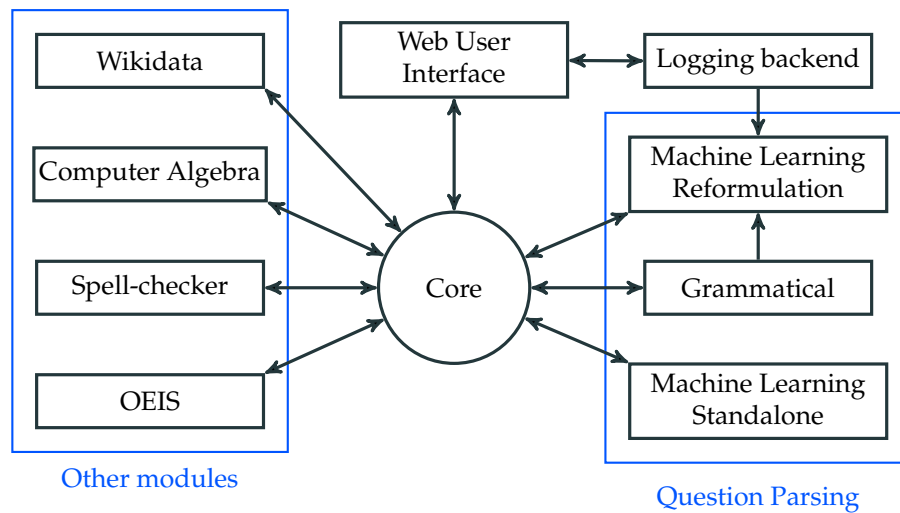


Figure 2.1: Architecture of the PPP

2.3 Evolution of data model

In the first version of the data model, there were only incomplete triples. It allowed us to represent most of simple questions like "When is Douglas Adams born?" but not composed or yes/no questions because its semantic power was intrinsically lower.

Chapter 3

Core

3.1 Core

As its name suggests, the core is the central point of the PPP. It is connected to all the other components — user interfaces and modules — through the protocol defined above. It is developed in Python.¹

The core communicates with the user interfaces and the modules *via* HTTP: each time the core receives a request from an interface, it forwards it to modules using a configurable list of URL of other modules.

An example configuration is the one we use on the production server (truncated):

```
{
  "debug": false,
  "modules": [
    {
      "name": "questionparsing_grammatical",
      "url": "python:ppp_questionparsing_grammatical.requesthandler:RequestHandler",
      "coefficient": 1,
      "filters": {
        "whitelist": ["sentence"]
      }
    },
    {
      "name": "cas",
      "url": "http://localhost:9003/cas/",
      "coefficient": 1,
      "filters": {
        "whitelist": ["sentence"]
      }
    },
    {
      "name": "spell_checker",
      "url": "python:ppp_spell_checker.requesthandler:RequestHandler",
      "coefficient": 1,
      "filters": {
        "whitelist": ["sentence"]
      }
    }
  ],
  {
```

¹<https://github.com/ProjetPP/PPP-Core/>

```

        "name": "wikidata",
        "url": "http://wikidata.backend.askplatyp.us/",
        "coefficient": 1,
    },
    {
        "name": "datamodel_notation_parser",
        "url": "python:ppp_datamodel_notation_parser.requesthandler:RequestHandler",
        "coefficient": 1,
        "filters": {
            "whitelist": ["sentence"]
        }
    }
],
"recursion": {
    "max_passes": 6
}
}

```

This example presents all the features of the core.

3.1.1 Communicating with modules

```

{
    "name": "cas",
    "url": "http://localhost:9003/cas/",
    "coefficient": 1,
    "filters": {
        "whitelist": ["sentence"]
    }
}

```

This is the entry to connect the core to the CAS (Computer Algebra System) module. We provide a nice name (to be shown in logs), the URL to know how to reach it, a coefficient applied to the measures, and filters (cf. chapter 2).

3.1.2 Recursion

```

"recursion": {
    "max_passes": 6
}

```

This is why the core is more than just a router. Not only does it forward requests to modules, and forward their answers to the client, but it can make several passes on a request.

Let's take an example: the question "What is the capital of India?".² We need three passes to answer such a question.

1. Spell-checking, to change it to "What is the capital of India?"
2. Question parsing, to change it to a machine-understable query: "(India, capital, ?)"
3. Database request to Wikidata, to get the final answer: "New Delhi"

After each pass, the router accumulates the different answers, until there is no new answer. Then, it stops recursing and returns everything to the client. There is a fixed maximum number of passes, although it is not necessary for now since modules we implemented cannot loop.

²<http://askplatyp.us/?lang=en&q=What+is+the+capital+of+India%3F>

3.1.3 Optimizations

We implemented two optimizations in the core. Small experimentations showed that it makes the core 25% faster on queries with multiple passes.

The two of them are used in this example:

```
{
  "name": "questionparsing_grammatical",
  "url": "python:ppp_questionparsing_grammatical.requesthandler:RequestHandler",
  "coefficient": 1,
  "filters": {
    "whitelist": ["sentence"]
  }
}
```

First, the filters. They can be either a whitelist or a blacklist of root node types, that the core will read to know if the module might do anything with such a tree. Obviously, the question parsing modules cannot do anything with something that is not a sentence, so it is useless to query it with something that is not a sentence.

Secondly, the “python:” URL. The core can call directly Python modules without using HTTP; and thus avoids the cost of serializing, using IPCs (Inter-Process Communications), and deserializing (twice). However, we need to run the module in the same process as the core, we therefore only use it for light modules (questionparsing_grammatical, spell-checker and OEIS).

3.2 Libraries for modules

We also provide a library in charge of handling and parsing HTTP requests following the format defined in the data model. It was originally used in the core, but then we split the code, as modules can reuse it without a change. This class is an abstraction over a Python HTTP library (python-requests), allowing module developers to focus on developing their actual code instead of handling communication.

These has proven to be efficient, since the spell-checker and the OEIS modules were created very quickly, using existing code.

It also exports its configuration library too, since modules share the same way of handling configuration (a JSON file, whose path is given *via* an environment variable).

For instance, here is the configuration file of the CAS module (Computer Algebra System), used to allow system administrators to set memory and time limits to computation:

```
{
  "max_heap": 10000000,
  "timeout": 10
}
```

Chapter 4

User interface

The user interface is composed of one web-page developed in HTML 5 with some pieces of JavaScript and CSS.¹ We have taken care of having an interface that fits nice on both huge screens of desktop computers and small screens of phones using responsive design techniques.

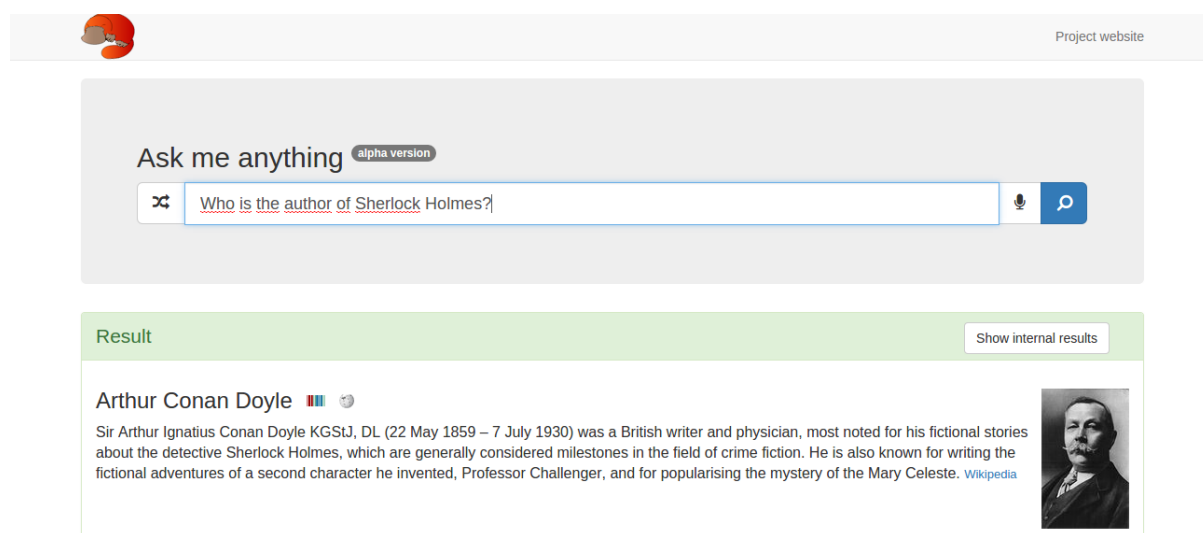


Figure 4.1: The user interface (Wikipedia header)

It is made of one huge text input with a button to submit the query and another one to get a random question from a hand written list. The text area allows to input question in natural language or using the data model notation like (Douglas Adam, birth date, ?) to find the birth date of Douglas ADAM. A parser written as a Python module² using the PLY³ library converts the data model notation into the internal JSON representation.

When a query is submitted, the URL of the page is updated accordingly in order to allow the user to directly link to a given question. We use also an experimental web browser feature, Web Speech⁴ (fully implemented only in Chrome in December 2014), to allow the user to input the query using a speech input and, when this feature is used, the UI speaks a simplified version of the query results.

This interface relies on some famous libraries like jQuery,⁵ Bootstrap,⁶ MathJax⁷ to display mathematics

¹<https://github.com/ProjetPP/PPP-WebUI/>

²https://github.com/ProjetPP/PPP-datamodel-Python/blob/master/ppp_datamodel/parsers.py

³<http://www.dabeaz.com/ply/>

⁴<https://dvcs.w3.org/hg/speech-api/raw-file/tip/speechapi.html>

⁵<http://jquery.com/>

⁶<http://getbootstrap.com>

⁷<http://www.mathjax.org/>



Ask me anything

alpha version



Where is the capital of Argentina?



Result

Show internal results

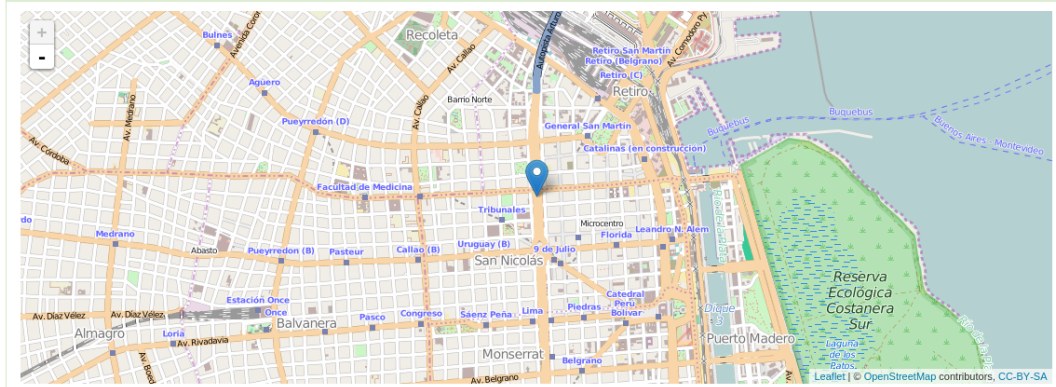


Figure 4.2: The user interface (OpenStreetMap location)

and Leaflet⁸ for maps.

4.1 Logging

All requests sent to the PPP are logged in order to be able to see what the user are interested in and maybe use them as data set for the question parsing modules that use Machine Learning. This logger is powered by a backend that stores data gathered via the user interface in a SQLite database.

The logger has a simple public API to retrieve last and top requests in JSON (JavaScript Object Notation): <http://logger.frontend.askplatyp.us/?order=last&limit=10> to get the last 10 requests and <http://logger.frontend.askplatyp.us/?order=top&limit=5> to get the top 5.

In order to use these data for machine learning we may also want, in the future, to log user feedback in addition to the requests themselves: after showing the user the way we interpreted their question alongside the answer to their request, we would provide them a way to give us feedback. What we call feedback is actually a thumb up / thumb down pair of buttons, and, if the latter is pressed, a way to correct the requests parsing result so it can be fed to the Machine Learning algorithms.

⁸<http://leafletjs.com/>

Chapter 5

Question parsing

The purpose of the three following modules is to transform questions into trees of triples, as described in section 2.1, which can be handled by backend modules.

The difficulty of this task can be illustrated on the following example:

Where was the president of the United States born?

A first possible tree is: (president of the United States, birth place, ?). However, the entity “president of the United States” is probably too complex to occur in a database.

On the other hand, the following tree is much more easy to process: ((United States, president, ?), birth place, ?). In this case, the president of United States is identified (Barack Obama), the triple becomes (Barack Obama, birth place, ?), and finally the answer can be found easily in “Barack Obama” occurrence.

Our goal is to product simplified and well structured trees, without losing relevant information of the original question. We are developing three different approaches to tackle this problem. The first tries to analyse the grammatical structure of questions, the two other ones implement automatic learning techniques.

5.1 Grammatical approach

A natural way to map questions into normal forms is to analyse the grammatical structure of the sentence. Some algorithms use the concrete syntax tree (or parse tree) representation to perform this task ([RDF⁺07] or [Def09]). Another popular representation is the dependency tree. This structure describes grammatical relationships between the words of the sentence. Although the dependency tree is often used in Question Answering ([ZNF06] and [ZNF07]), algorithms to produce triples from it are pretty rare and poorly described.

The aim of this module is to provide a full algorithm that maps questions into normal forms, mainly using the dependency tree representation output by the Stanford Parser (see [dMM13]). We will focus on factoid *wh*-questions.

5.1.1 Methods

The normal form is obtained by applying some operations on the dependency tree output by the Stanford Parser. We detail throughout this section the different steps of our algorithm and we apply them on the example:

Where was the president of the United States born?

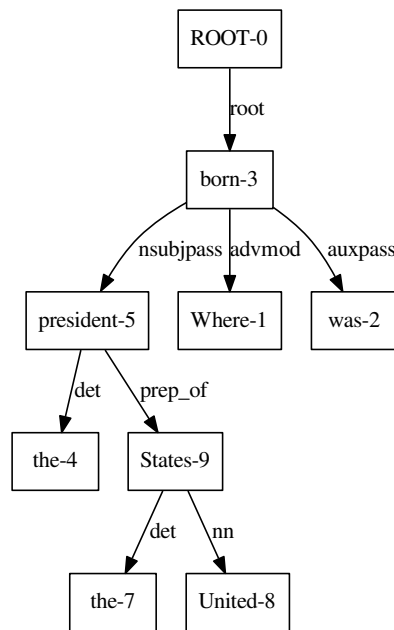


Figure 5.1: Dependency tree of *Where was the president of the United States born?*

Stanford dependency tree

The Stanford Parser¹ is a natural language parser developed by the *Stanford Natural Language Processing Group*. It is well-documented and considered as a “state of the art” program. The Stanford Parser provides classical tools of NLP theory (part-of-speech tagger, named-entity recognizer, constituency parser, etc.) but also a very powerful dependency parser.

A dependency tree reflects the grammatical relationships between words in a sentence. Figure 5.1 provides an overview of such a tree in our example. For instance, the edge:

$$\text{president} \xrightarrow{\text{det}} \text{the}$$

means that *the* is a determiner for *president*.

The Stanford typed dependencies manual ([dMM13]) details the full list and description of possible grammatical dependencies.

Dependency tree simplification

The simplification of the dependency tree consists of several operations applied on the tree to change its shape and prepare the production of the normal form. At the end of the simplification, all the dependency tags (the Stanford Parser uses more than 50 grammatical dependency tags) have been replaced by a small subset of eight (new) tags.

Preprocessing First of all, we perform multiword expressions recognition in order to merge all the nodes of the tree that belong to a same expression. We merge especially neighbour nodes with a same

¹<http://nlp.stanford.edu/software/corenlp.shtml>

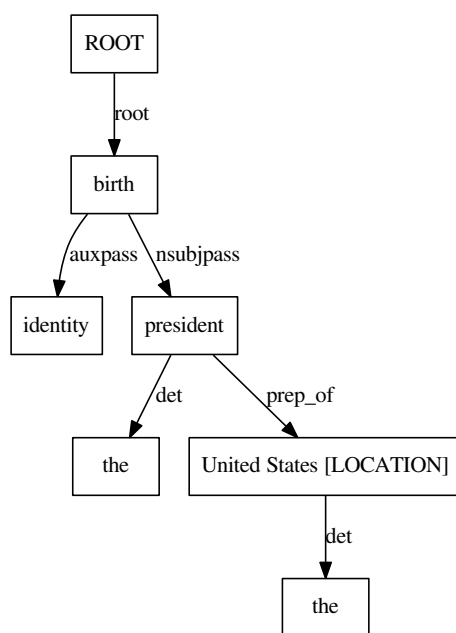


Figure 5.2: Preprocessed dependency tree of *Where was the president of the United States born?*

named-entity tag (for instance, *United* and *States* are tagged `LOCATION` and therefore merged into *United States*). Other kinds of merging are performed in section 5.1.1, using the grammatical dependencies.

Then, the question word (Who, Where, How tall, etc.) is identified and removed from the dependency tree (it will be used later to build and improve the normal form). This operation is performed using a list of about 30 question words and looking at the two first words of the sentence (they are supposed to contain the question word).

Finally, lemmatization is applied on nouns (nouns are put in singular form, e.g. *presidents* is mapped to *president*) and nounification on verbs (verbs are transformed into some related noun, e.g. *born* is mapped to *birth*). These two operations are performed thanks to the *NLTK* library²:

- Lemmatization is made by the *WordNetLemmatizer* class.
- Nounification is more complex. It uses *NLTK* and the *WordNet* synonym sets to find all the nouns related to the verb. Then, it keeps only the nouns which have the most occurring stem (using the Porter stemmer algorithm of *NLTK*). Finally, it returns the most occurring noun.

Preprocessing is illustrated in figure 5.2.

Global transformations Two kinds of transformations modify the global shape of the tree. These operations are applied for `amod` dependencies if the output node of the dependency edge is a leaf and has a `JJS` (superlative) or `ORDINAL` part-of-speech (POS) tag. They are also applied for `conj` dependencies.

Global transformations add a new node and re-balance the tree. Figure 5.3 illustrates the transformation applied for conjunction dependencies (`conj_or`, `conj_and`, etc.). Figure 5.4 gives the transformation for an `amod` dependency (the POS tag of *highest* is `JJS`).

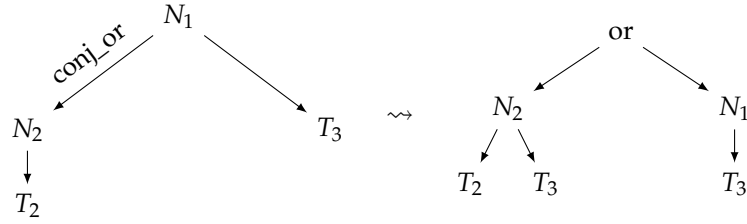


Figure 5.3: Remove conj_or dependencies

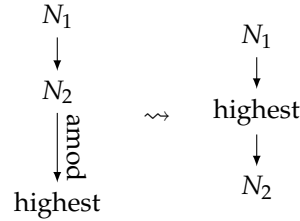


Figure 5.4: Remove amod dependencies

Local transformations Finally, all remaining edges are analysed locally. We apply one of the following rule to each of them, depending on their dependency tags:

- **Merge** the two endpoints of the edge. It is a new step of multiword expressions merging. This rule is applied for *nn* dependencies for example:

birth $\xrightarrow{nn}$ date becomes: birth date

- **Remove** the sub tree pointed out by the edge. This rule is applied for *det* dependencies for example:

president $\xrightarrow{det}$ the becomes: president

- **Replace** the dependency tag by a *triple production tag*. We have currently defined eight different types of *triple production tags*: $R_0, \dots, R_5, R_{spl}, R_{conj}$. These tags are used to produce the final result. We attribute the same tag to dependency relationships that must produce the same type of nodes or connectors in the normal form. For instance, we replace *prep* and *poss* dependencies by the tag R_5 :

president $\xrightarrow{prep_of}$ France becomes: president $\xrightarrow{R_5}$ France

Finally, depending on the question word we add some information in certain nodes (for instance, if the question word is *where* we try to add the word *place* into the child of the root of the tree). This extra transformation is supported actually for about 30 question words.

Figure 5.5 illustrates the dependency tree simplification applied on our example.

Construction of the normal form The final step of the algorithm maps the tree obtained at the end of section 5.1.1 to a normal form that follows the data model presented in part 2.1. The transformation is a recursive function `Normalize` that takes a tree T as input and outputs the normal form. We give the main rules applied to obtain `Normalize( $T$ )`. Trees are denoted by $T_{\dots}$ and nodes by $N_{\dots}$. For a tree T (resp. a node N), $\underline{T}$ (resp. $\underline{N}$) represents the words contained in the root of T (resp. in N).

²<http://www.nltk.org/>

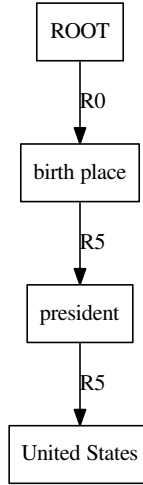


Figure 5.5: Simplified dependency tree of *Where was the president of the United States born?*

Rule R	$\text{Normalize}(N \xrightarrow{R} T)$
R_0	$\text{Normalize}(T)$
R_1	$\underline{T}$
R_2	$\text{Triple}(\underline{T}, \underline{N}, ?)$ if T is a leaf $\text{Normalize}(T)$ otherwise
R_3	$\text{Triple}(?, \underline{N}, \text{Normalize}(T))$
R_4	$\text{Triple}(?, \text{Normalize}(T), \underline{N})$
R_5	$\text{Triple}(\text{Normalize}(T), \underline{N}, ?)$

Table 5.1: Normalization rules for $R_0, \dots, R_5$

First of all, if N is a leaf then $\text{Normalize}(N) = \underline{N}$ (*value node*).

Then, rules R_0 to R_5 are used to produce the different types of possible triples. Table 5.1 gives $\text{Normalize}(N \xrightarrow{R_i} T)$ when T is the only child of N and $R_i \in \{R_0, \dots, R_5\}$. For instance, $\text{Normalize}(N \xrightarrow{R_3} T)$ is $\text{Triple}(?, \underline{N}, \text{Normalize}(T))$.

When the root N has more than one child, all linked by a rule in $\{R_0, \dots, R_5\}$, we apply the rule of figure 5.6.

Rules R_{spl} and R_{conj} are produced by the global transformations performed in part 5.1.1. When rule R_{spl} occurs, the root node contains a superlative or an ordinal and it has only one child. Depending on the superlative/ordinal (we have listed the most common ones), Normalize outputs the relevant connector nodes. A general example is presented in figure 5.7. Rule R_{conj} is used for conjunction. A general example is presented in figure 5.8.

All the rules we use are available in our implementation³ and in appendix B.1. Figure 5.9 gives a

³<https://github.com/ProjetPP/PPP-QuestionParsing-Grammatical>

$$\text{Normalize} \left(\begin{array}{c} \swarrow R_{i_1} \quad N \quad \searrow R_{i_n} \\ T_{i_1} \quad \dots \quad T_{i_n} \end{array} \right) = \bigcap \left(\text{Normalize}(N \xrightarrow{R_{i_1}} T_{i_1}), \dots, \text{Normalize}(N \xrightarrow{R_{i_n}} T_{i_n}) \right)$$

Figure 5.6: Normalization rule for $R_0, \dots, R_5$ ($(R_{i_1}, \dots, R_{i_n} \in \{R_0, \dots, R_5\})$)

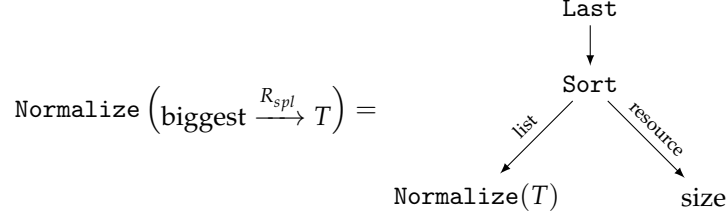


Figure 5.7: Normalization rule for R_{spl}

possible normal form obtained in our example. Appendix A.1 presents the formal representation of this normal form, following the data model.

5.1.2 Results and discussion

The previous algorithm has been implemented in Python 3. We use the collapsed dependency tree output by the *CoreNLP* parser with the flag `-makeCopulaHead`. We access *CoreNLP* using a Python wrapper⁴ we have patched to support Python 3. The code is available online³. It includes a documentation and demo files that allow the user to quickly test the algorithm. Moreover, when the user enters a question in our search engine, he can get the answer and visualise the normal form by clicking on the `Show internal results` button.

We have displayed in figure 5.10 four normal forms produced by our algorithm on questions used in TREC-8 contest. These results are quite representative of what our tool can do and they are close to the expected normal forms. We have also noticed that our algorithm also supports some *nominal sentences* such as “*capital of England*”.

These results prove the relevance of studying the dependency tree to extract triples. Indeed, it can be seen that the structure of the dependency tree is very close to the structure of the normal form we build. The constituency tree (as in [RDF⁺07] or [Def09]) does not allow to handle as complex sentences as we do. Naturally, questions with complex grammatical structures are more likely to be misparsed. We also do not currently support all kinds of questions (Yes/No questions for instance).

$$\text{Normalize} \left(\begin{array}{c} \swarrow R_{conj} \quad \text{and} \quad \searrow R_{conj} \\ T_1 \quad \quad \quad T_2 \end{array} \right) = \bigcap \left(\text{Normalize}(T_1), \text{Normalize}(T_2) \right)$$

Figure 5.8: Normalization rule for R_{conj}

⁴<https://bitbucket.org/ProgVal/corenlp-python/overview>

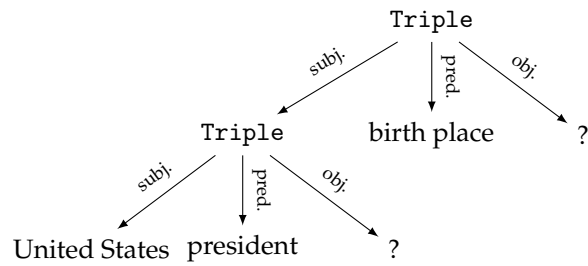


Figure 5.9: Possible normal form for *Where was the president of the United States born?*

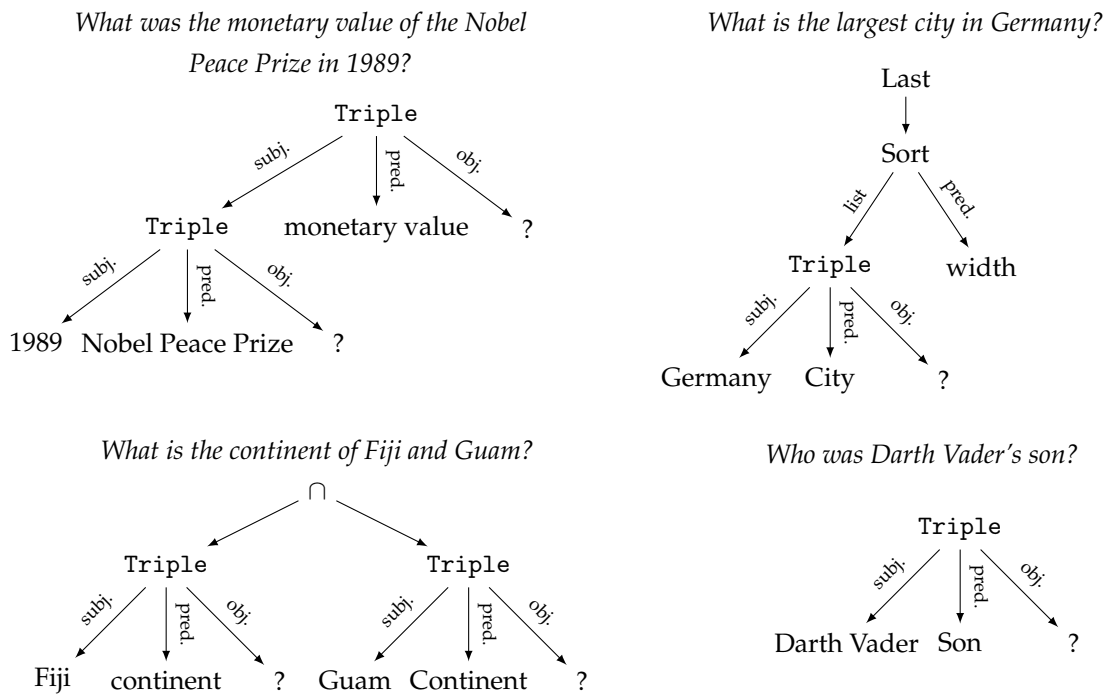


Figure 5.10: Example of normal forms produced by the algorithm

5.1.3 Future work

Future work will consist in improving the different parts of the algorithm (multiword expressions recognition, better analysis of grammatical dependencies). The Stanford Parser uses automatic learning techniques to determine how to build the dependency tree. We plan to train it on our own annotated data set in order to improve its analyses of interrogative sentences. Moreover, our approach can be easily transposed to languages using similar grammatical dependencies than English. Finally, the extraction of triples is useful into many fields of NLP theory, other than Question Answering. For instance, our algorithm could be adapted to automatic text summarization or speech processing.

5.2 Reformulation: a learning approach

Efficiency of neural network nowadays can be very good. An approach based on network is used here. But, the aim is not to learn answering question. There are two reasons for that. First the knowledge changes over time, e.g. the president name depends on the date. Then, something simple is wished. The learning process should also not be continuous. Learning the answer implies to know proper names which increase the size of the dictionary, that means increasing the learning time.

The module presented here tries to reformulate a triple into a more suitable triple for the Wikidata module. It means adapting it as most as possible for this module, correct the mistakes of grammatical parsing, as it is illustrated in figure 5.11.

5.2.1 How it works

The grammatical module builds a tree representation of the question. The reformulation module works on this output, so with a tree representation. The assumption “all useful information is kept in this form” is also necessary. The module has also a dictionary. There are four steps to do. First a pre-treatment replaces proper names and numbers with tags, such that all words of the triple are in the dictionary, then the request is projected in the math space of request. Two last steps revert the two first, but the reverse projection should produce a better request, which mean the final request is adapted for the Wikidata module.

For compatibilities reasons, we do not work with the actual data model which is too complicated. In fact we need a regular triple representation, that means all operators must be transformed, this step is easy. For unitary operators, $O(f)$ becomes $(f, O, ?)$; for binary operators, aOb becomes (a, O, b) . This operation does not weaken the expressive power.

WHEN WAS THE PRESIDENT OF THE UNITED STATES BORN?

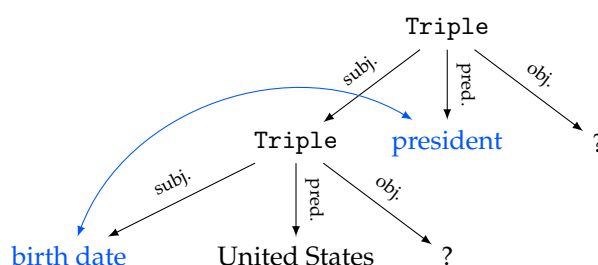


Figure 5.11: An example of bad triple formation, “president” and “United States” terms are in the wrong place. The reformulation module should solve this kind of problem.

Mathematical spaces

The mathematical space of request $\mathcal{R}$ is simply the subset of the $\mathbb{R}$ vector space of dimension 50, where all vectors have a Euclidean norm of 1. A triple is formed of a subject request, a predicate and an object request, it is also natural to consider the triple over $\mathcal{R}$, which elements order with always be subject, predicate, object. Let us call vector the elements of $\mathcal{R}$.

The choice of a 50-dimension can be modified if necessary, but taking a higher dimension could slow the learning process, and with a lower space we could lose some expressive power, which may lead to very poor results.

Dictionary

The dictionary defines matching between English words and vectors triple, which is the base of the process. We use triple because a word could have different meaning depending on his nature, so predicate vector for a word is not the same as subject vector which is not the same as object vector. We will denote $m.s$ subject vector of word m , $m.p$ and $m.o$ for predicate and object.

Pre- and post-treatment and treatment

We evoke some reasons not to handle proper name directly (the dictionary size would increase and names could change), there is another argument: there is an arbitrary number of proper names, because you can invent some new names every time. That is why they are replaced in a request by a tag $NAME_i$, $i \in 1, 2, 3$. It allows us to have three different names in a question, and that is sufficient. The same happens to the numbers. Tag UNKNOWN finally represents the “holes” in the request tree. At the end, after the treatment we replace tags with corresponding real names or numbers in the final tree.

Recognizing a number is not hard, in fact it is just checking if the character sequence looks like numbers optionally followed by a dot and numbers. If the sequence is not a number or a “hole” and is not in the dictionary, the module treats it as a proper name.

Project a request form to a vector

The model allows an arbitrary request form, it means we have a tree with an unknown depth, and to keep the complete question information we should handle it so. But the request are triple, so it is easy to transform. First with the dictionary all words are replaced with vectors, of course we take the vector corresponding to the function of the word in the tree. Then we have to ascend the whole tree to compute the root value.

Let define a matrix compact which takes a triple of vector and merge them in one vector, here merging is not an intuitive operation, in fact we don't know how to merge this triple, that is why all coefficients of the matrix are unknown. To compute what we call a vector, output should be normalized.

Then, apply this operation bottom-up in the tree. The main idea is each node value represents the subtree request. A tree illustrates this on figure 5.12.

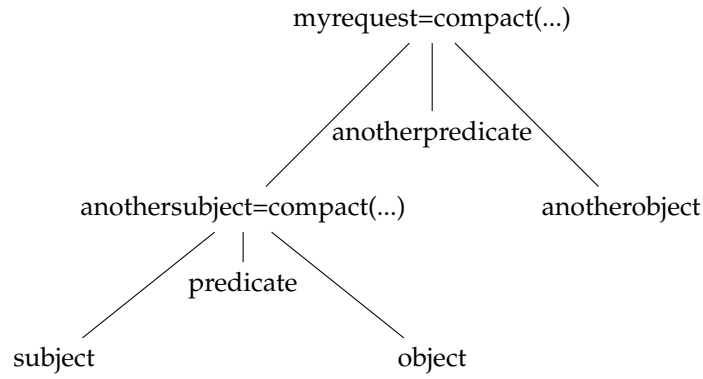


Figure 5.12: The process of projection, an example

Reconstruct a request from a vector

This operation is quite symmetrical of the projection, with a matrix uncompact we can obtain a vector triple from a vector, and recursively build a tree. The question is how to know if we should reapply the function uncompact or leave the vector as a leaf? First say if a triple predicate is never a tree, then object and subject will be let as leaf if a known word is near enough. Finally each node is replaced with the nearest corresponding word in the dictionary. For example, a vector in middle of a triple which is also a predicate is replaced by the word with nearest predicate's vector.

Defining "near enough" is difficult. To avoid infinite loop we will take depth of nodes into account. We must take $\delta > 0$ a precision factor and g a growth factor. If d is the depth of node n , near enough means distance is bounded by $\delta * g^d$ with regard of Euclidean norm.

The algorithm reconstruct is also:

Algorithm 1: RECONSTRUCT From vector to tree

Input: $\delta > 0$ and $a \in A$

Output: request tree

$(s, p, o) \leftarrow \text{uncompact}(a)$

Find m s.t. $\|m.s - s\|_2 < \delta$ is minimal

if m exists **then** $s \leftarrow m$

else $s \leftarrow \text{reconstruct}(\delta * g, s)$

Find m s.t. $\|m.o - o\|_2 < \delta$ is minimal

if m exists **then** $o \leftarrow m$

else $o \leftarrow \text{reconstruct}(\delta * g, o)$

$p \leftarrow \text{argmin}_m(\|m.p - p\|_2)$

return (s, r, o)

You can find a pictured version of the algorithm on figure 5.13.

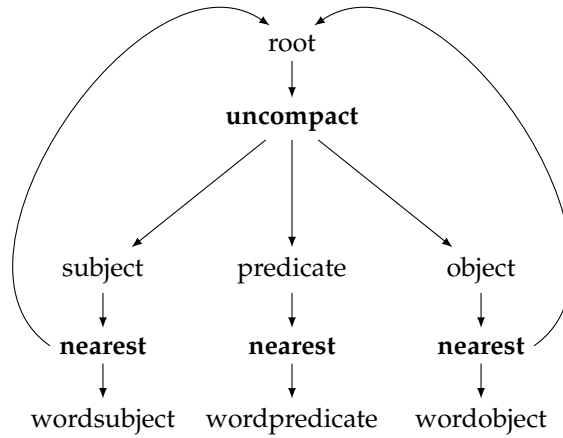


Figure 5.13: The algorithm RECONSTRUCTION in a scheme

Remarks

Matrix compact and uncompact are not bijective after restraining the request space to existent requests. Applying uncompact then compact should give the identity, with of course an error margin, but when compacting then uncompacting we only have to find an equivalent request, i.e. with the same meaning but with another formulation. If it were not the case it would produce exactly the same output as the input, that would be useless.

5.2.2 Learning

Training the model cannot be done easily with a classic back-propagation because we have no idea what is the best triple for a question. The solution is also to try to change a little the matrix, and see what changes improve the better the quality of result, it is however very bad in cost.

Now to compute the quality of the result, i.e. the score of the module we use a semantic distance: it is a norm depending on the meaning of words. To be simple we base it on the relation graph “instance of” of Wikidata assuming it is a directed acyclic graph. We compute the nearest common ancestor and we add the distance between this node and the two word nodes.

5.2.3 Implementation

The module is written in C++, with threads to speed up the matrix multiplication. To interface it with the others, it is built as a shared library with python library. The dictionary has been generated using the clex.⁵

5.2.4 Future work

The learning process has not been performed, and so the reformulation module is not yet operational. As latency with Wikidata is very high (several seconds) we have to download Wikidata and run it in local.

Then we could think of different manners to train the model.

⁵<https://github.com/Attempto/Clex>

Finally we could think about a good manner to find the nearest word in the dictionary. It can be done with a linear time, but considering there are 100 000 words in the dictionary, cost is not negligible. As we work in dimension 50, it is not a trivial problem to find an efficient method to obtain a logarithmic complexity. Kd-trees allow a search in log-time (with preprocessing); but with dimension 50, the constant factor 2^{50} is too large. One line of research were to use distance sensitive hash.

5.3 Machine Learning: Window approach

The goal of this module is to produce triples from sentences and to be an alternative to the grammatical approach.

We used machine learning algorithms in order to produce triples from scratch, without any grammatical library like *Stanford CoreNLP*. Our work is mainly based on the paper [CWB⁺11].

Motivations come from three points:

- Because triples are linked to the semantic of the sentence, and not directly from the grammar, we could expect that avoid grammatical tools can be a good idea.
- It has been shown that a machine learning approach can produce, for a large panel of different NLP problems, very good solutions, closed to *state of the art* algorithms [CWB⁺11].
- Grammatical approaches fail on keyword sentences, like "Barack Obama birth date" because this kind of sentences does not respect the syntax of English.

Due to the restricted amount of time, we emphasis on keyword questions (keyword questions can not be analysed with grammatical tools) and we limit ourself to a restricted version of the data model:

- Only one level of depth: for example the sentence "What is the birth date of the president of the United States?" will be converted to the triple: (president of the United States, birth date, ?).
- We do not support other kinds of connectors such as *first*, *sort*, etc.

We use a look-up table and a window approach neural network as explained below. The complete module⁶ was written in Python 3. We use the scikit-learn library, NumPy and NLTK as external tools.

Our algorithm can be described with three components: the look-up table, the window approach, and the classifier.

Look-up table

The look-up table is a dictionary that associates to each word w a vector $V_w \in \mathbb{R}^n$, where n is the number of parameters used to encode a word (we used $n = 25$). If two English words w_1 and w_2 are synonymous, then $\|w_1 - w_2\|_2$ is small.

The construction of the look-up table is described in [CWB⁺11] and used unsupervised machine learning techniques. We used the pre-computed look-up table found here: <http://metaoptimize.com/projects/wordreprs/>

We also add one parameter to know if the word starts with a uppercase letter or not. Finally, words are embedded in vectors of dimension 26.

⁶<https://github.com/ProjetPP/PPP-NLP-ML-standalone/>

Window approach

We use a window (as explain in [CWB⁺11]) that focuses on one word to perform the classification. For instance, if the sentence is "What is the birth date of the president of France?", and the word to classify is "date", for a window size of 7, the window is: "is the birth **date** of the president".

We use this window because classifier algorithms usually work with a fixed number of input parameters.

The window size is a meta parameter to choose. This window has to be large enough that we can decide in the context of the window in which category a word is. We used a window of size 9.

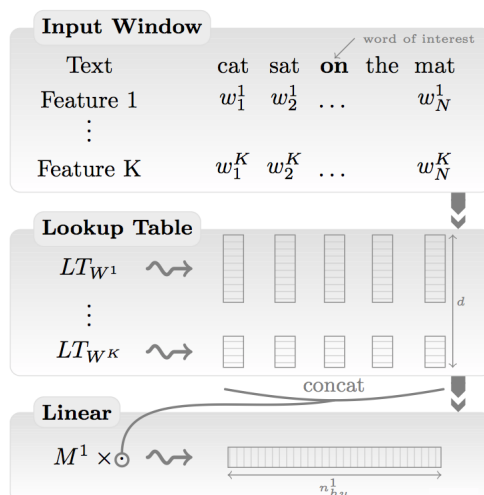
The classifier

If the question is "What is the birth date of the president of France?", and the focus of the window approach is on the word **date**, then the goal is to classify the word **date** into one of these four categories: *subject*, *predicate*, *object*, *to forget*.

The classification of each word of the sentence into these four categories finally gives us the desired triple.

The principle of the complete algorithm can be summarize in the figure 5.14.

Figure 5.14: The architecture of the algorithm, as described in [CWB⁺11]



Because we have a small data set of annotated questions (see below), we use a linear model: a linear model has the advantage to have few parameters. For example with a window approach of size 9 and words embedded in vectors of dimension 26, this gives us $26 \times 9 \times 4 = 936$ parameters to learn.

In a first time we implemented our own linear classifier, but to improve the execution time, we finally use the *LDA* classifier from the library scikit-learn. Only a few seconds of computation are now needed to successfully train the model.

5.3.1 Data set

Because we use supervised algorithms, we need a data set of annotated questions, in order to learn our model. This data set was mostly built manually, because we did not find on the internet a data set that directly answering the problem of triple extraction. Building this data set is a fastidious work. Currently our data set is composed of 300 questions. We also wrote a script that generate keywords

questions. This script gives us 500 keywords questions and makes the module specialized on keywords questions.

We now denote $\mathcal{D}_{GC}$ (for *Data set Grammatically Correct*) the data set of grammatically correct questions, $\mathcal{D}_{kw}$ for the data set of generated keywords questions and $\mathcal{D}_A = \mathcal{D}_{GC} \cup \mathcal{D}_{KW}$ the complete data set.

5.3.2 Results

In this basic configuration, we can measure the accuracy of our algorithm, defined as the ratio of correctly classified words, for a specified data set.

Our methodology is the following: for a specified data set $\mathcal{D} \in \{\mathcal{D}_A, \mathcal{D}_{GC}, \mathcal{D}_{KW}\}$ we split $\mathcal{D}$ in two parts: the training set and the test set (90% of the data is for the training set and 10% is for the testing set). We learn our classifier with the training set and then evaluate it on the test set.

Remark: to compute the accuracy of one variant of our algorithm we split the data set into testing and training set randomly, and we restart an experience 50 times. This makes us sure of the precision of the estimated accuracy.

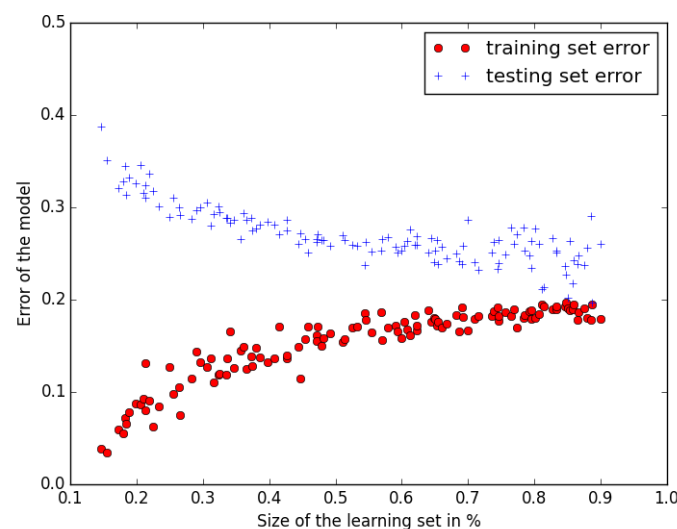
Data set	Testing accuracy	Training accuracy
$\mathcal{D}_{GC}$	75%	81%
$\mathcal{D}_{KW}$	98%	98.3%
$\mathcal{D}_A$	83%	86%

We can conclude that this version of the algorithm is excellent for keyword questions, and not really good for grammatically correct questions.

Bias vs Variance test

We can plot the *Bias versus Variance* curves:

Figure 5.15: Bias vs Variance curve for the data set $\mathcal{D}_{GC}$



These curves show us that there is a small gap between the two performance curves for the data set $\mathcal{D}_{GC}$: the data set of grammatically correct questions is too small to learn correctly the model.

However, when we add the keyword questions, the gap disappears: the model is learned correctly and the performances are good.

Figure 5.16: Bias vs Variance curve for the data set $\mathcal{D}_{KW}$

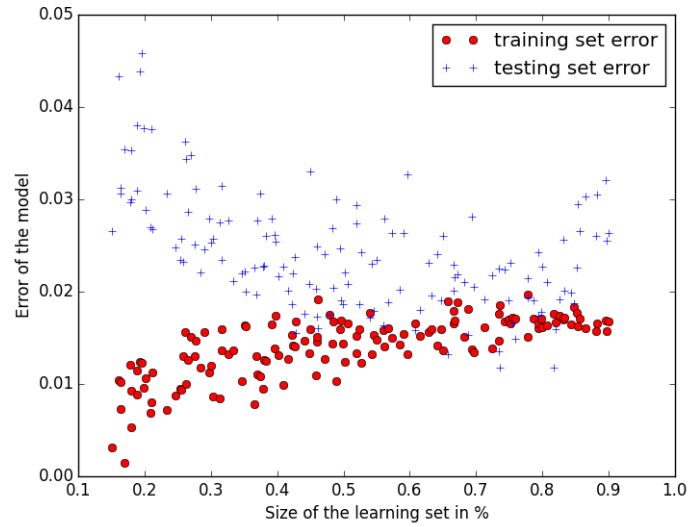
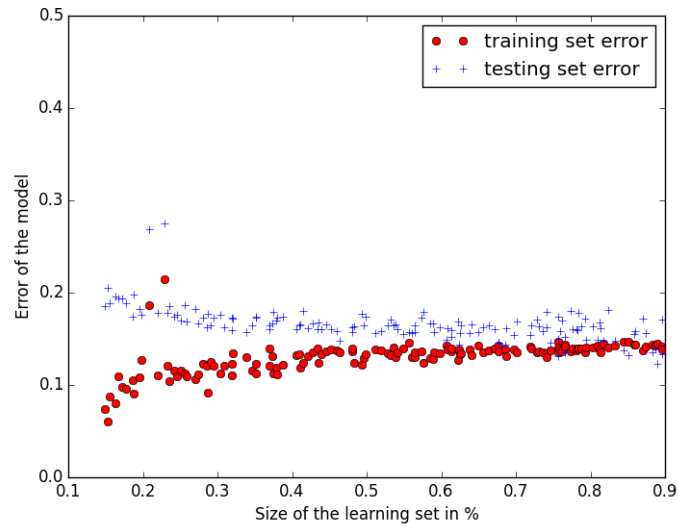


Figure 5.17: Bias vs Variance curve for the data set $\mathcal{D}_A$



5.3.3 Improvements of the algorithm

POS TAG features

In order to help the classifier, we added POS TAG features: each word of the sentence is tagged with a grammatical tag (VERB, NOUN, PRON, ADJ, ...)

We use the NLTK pos tagger to do that. With such a tag, a word is encoded as a vector of dimension $26 + 11 = 37$.

This new feature improves the accuracy of the algorithm of a few percents:

Data set	Testing accuracy without POS TAG	Testing accuracy with POS TAG
$\mathcal{D}_{GC}$	75%	77.2%
$\mathcal{D}_{KW}$	98%	98.9%
$\mathcal{D}_A$	83%	86%

5.3.4 Conclusion

The initial goal was to make a competitor to the grammatical approach to transform English questions into a structure defined in the data model. Due to the limited amount of time, we did not succeed to improve sufficiently the accuracy of the classifier to make this approach as good as the grammatical approach.

We decided then to focus on keywords questions, which are very important for a framework of question answering. For this task, with a good data set of annotated keyword questions, this algorithm is really efficient.

Chapter 6

Back-end

6.1 Wikidata

6.1.1 Overview

Wikidata module¹ is our main proof of concept which aims to demonstrate the ability of our framework to allow the easy creation of modules able to answer thousand of questions. It tries to answer general knowledge using the data stored in Wikidata.²

Wikidata is a free knowledge base hosted by the Wikimedia Foundation as a sister project of Wikipedia. It aims to build a free, collaborative, multilingual structured database of general knowledge (see [VK14]). It provides a very good set of APIs that allow to consume and query Wikidata content easily. Wikidata is built upon the notion of entities (called items and stored in a separated wiki pages) that are about a given subject. Each item has a label, a description and some aliases to describe it and statements that provides data about this subject (see figure 6.1.1). Each statement is built around a property and a value such that $(item, property, value)$ may be seen as a valid triple. Properties are entities like items and so may have labels, descriptions, aliases and statements. For more information see.³

We have chosen Wikidata instead of some other famous knowledge base like DBpedia⁴ or Freebase⁵ because Wikidata is under a CC0 license⁶ that allows to reuse data without constraints and is fully multilingual, a characteristic that will be very useful in order to adapt the PPP to other languages than English. More, as one of the members of the team is involved in Wikidata it allows us to avoid to lose time by learning the other knowledge base APIs.

6.1.2 APIs

Wikidata provides an API that allows to retrieve items and properties from there ids and an other that returns items or properties that match a given label. For more information about them see the appendix C.

Some libraries are available to interact easily with the API. The two major ones are:

- **Wikidata Toolkit**⁷ written in Java and allows to interact easily with the API and dumps of Wikidata content that are regularly done.

¹<https://github.com/ProjetPP/PPP-Wikidata/>

²<http://www.wikidata.org/>

³<http://www.wikidata.org/wiki/Wikidata:Glossary>

⁴<http://dbpedia.org>

⁵<http://www.freebase.com/>

⁶<http://creativecommons.org/publicdomain/zero/1.0/>

⁷http://www.mediawiki.org/wiki/Wikidata_Toolkit

George Washington (Q23) [edit]

American politician, 1st president of the United States (in office from 1789 to 1797) [edit]

Also known as: [edit]

[In other languages](#)
[Statements](#)
[Wikipedia pages linked to this item](#)
[Wikinews pages linked to this item](#)
[Wikiquote pages linked to this item](#)
[Wikisource pages linked to this item](#)
[Wikivoyage pages linked to this item](#)
[Pages on other sites linked to this item](#)

In other languages [edit]

français

George Washington

premier président des États-Unis d'Amérique

Also known as:

Statements

cause of death

bloodletting q356405 [edit]

▼ 0 references [add reference]

[add]

RSL identifier

000025437 [edit]

► 1 reference [add]

Figure 6.1: The beginning of a Wikidata item

- **wikibase-api**⁸ written in PHP that only allows to interact with the API but has the strong advantages to share a lot of code with the MediaWiki extension that powers Wikidata.⁹

We have chosen to use the PHP API because one of the members of the team has contributed in a significant manner to it and so was able to have a working module quickly.

In order to do queries in Wikidata content, like retrieving all presidents of the United States, e.g. all items with the property "position held" (P39) with value "president of the United States of America" (Q11696) we had to rely on an external service, called **WikidataQuery**¹⁰ that allows to do such requests. This was needed because the Wikidata API does not allow to do such queries yet. We have written a small standalone and reusable wrapper to the API of this service in PHP.¹¹

6.1.3 Implementation

In order to write this module nicely we have written a small PHP library, LibModulehttps://github.com/ProjetPP/PPP-LibModule-PHP, that provides a framework to create PPP modules and diverse utilities to simplify nodes like union or intersection that do not requires specific knowledge about the resources themselves.

⁸<https://github.com/addwiki/wikibase-api>

⁹<https://www.mediawiki.org/wiki/Wikibase>

¹⁰<http://wdq.wmflabs.org/>

¹¹<https://github.com/ProjetPP/WikidataQueryApi/>

37

This module works in tree steps:

1. It maps *resource* nodes of the question tree into Wikidata content: the subjects of *triple* nodes are mapped to Wikidata items, predicates to Wikidata properties and objects to the type of value that is the range of the Wikidata property of the predicate. If more than one match are possible, a *list* with all possible matches is used.
2. It performs queries against Wikidata content using the previously done mapping to reduce as much as possible trees. When we have a *triple* node where the object is missing the module gets the Wikidata item of the subject, looks for values for the predicate property and replaces the *triple* node with the *list* of *resource* nodes for each values of the property. When there is a *triple* node with a missing subject the module uses the WikidataQuery API.
3. It adds clean representation of *resource* nodes added by the previous phase. When Wikidata resources are returns it keeps their IDs in order to allow the ui to create nice display of them.

We have also added specific support for "name", "definition" and "identity" predicates in order to be able to output a nice display of the "Barack Obama" item when the query is "Who is Barack Obama?", with normal form (*Barack Obama, identity, ?*): the module maps the subject of the triple to Wikidata items and returns all the possible results. We do the same thing if the module input is a sentence like "Barack Obama".

One of the difficulties met was to avoid meaningless results because Wikidata contains items about Wikipedia organizational content like "disambiguation" or "category" pages. So we use a small filter that ignore these pages when the conversion of strings to Wikidata identifiers is done.

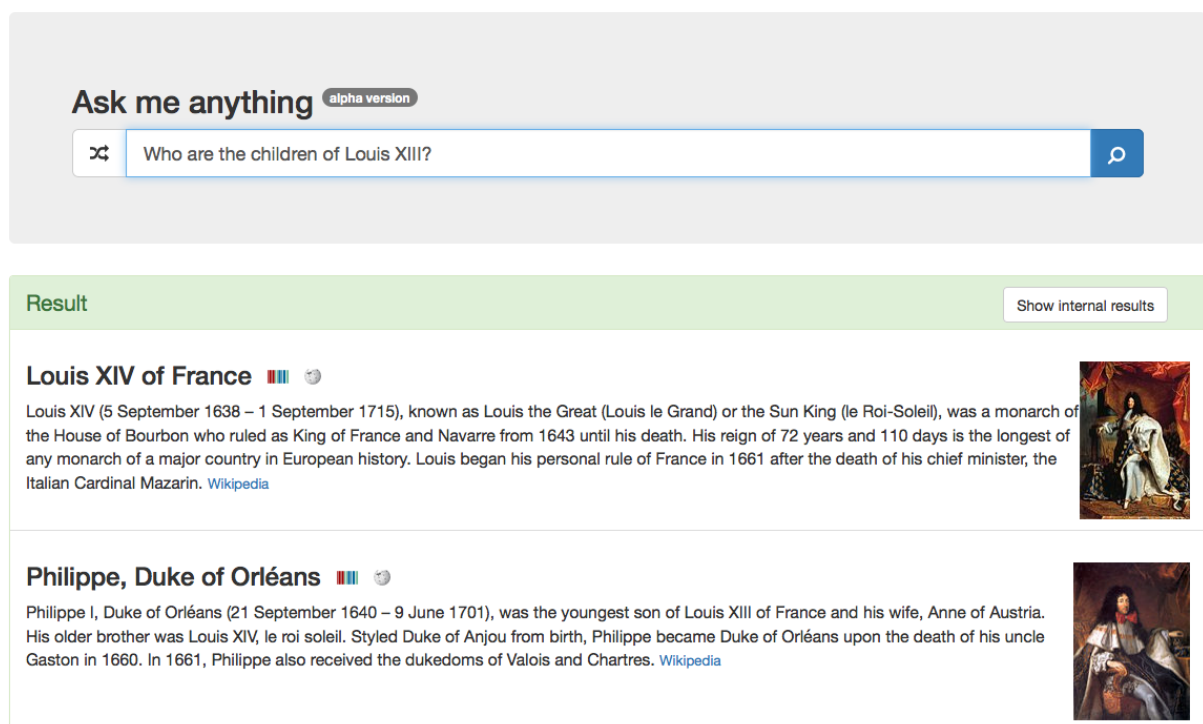


Figure 6.2: An example output from the Wikidata module

6.1.4 Future work

This module is only at its first stage and a lot of nice to have features remains:

- Clever filter of not relevant results: people that are looking for the list of the presidents of the United States are usually not looking for fictional ones.
- Handle triple that does not directly match to Wikidata model. If we looks for people born in France we are also looking for people born in Lyon or if we are looking for the American Secretary of State we are looking for the person that has as office "American Secretary of State".
- Do some reasoning on Wikidata content. For example if (Lyon, instance of, city) and (city, subclass of, place) then we should be able to guess and use that (Lyon, instance of, place).

6.2 CAS

A computer algebra system (CAS) module has been added to the PPP. Its work is to compute formally mathematical expressions.

6.2.1 Input-output

The CAS module takes as input a sentence whose string is a mathematical expression and output a resource of typed `math-latex` with two fields: one with a human readable formula and the other written in `LATEX`.

The difficulty is to decide whether a string represents a mathematical formula or not.

We use two tests. The first is a pre-treatment test. It checks whether the formula contains some substring which prove:

- the formula is a mathematical expression, like "sqrt", "\"... ,
- the formula is not a mathematical expression, like "who", "where", accented character....

But that does not eliminate all natural language questions, there still are false positive. Nevertheless, this test works in the larger part of cases.

So there is another test which is more precise. For example, a query like "author of bli-bla" will not be excluded by the previous algorithm. Moreover we see here why we can not consider '-' as a characteristic character of mathematical formula. The method is to verify if the module apply modifications.

In this example, the module has nothing to compute and just rearrange terms. So "author of bli-bla" becomes something like "author bli of - bla" considering 'author', 'bli', 'bla' and 'of' as variable. But we observe there was no modification. To do this test, the module counts each kind of symbol (except space and some particular symbols). If there is the same quantity of each symbol, we consider there is no modifications and the module decides he was useless and returns an empty list.

6.2.2 Structure

To do mathematical computations, we use a specific library: Sympy.¹² But this library is able to analyse only the input written with a syntax which is not intuitive and pretty complex.

To avoid arbitrary long computation, we launch Sympy evaluation in another process which has a limited memory and time.

To be usable, we define two parsers to handle other syntax: the syntax of Mathematica and another syntax we defined, which is designed to be intuitive and we named $C_{At}C_{AS}^H$.

¹²<http://www.sympy.org/en/index.html>

6.2.3 Parsers

Mathematica parser

This parser is quite simple. We use a LALR parser generated by PLY¹³ (Python Lex Yacc). First, the input formula is parsed and transform into a tree, then, the tree is transform into a Sympy expression.

$\mathcal{C}_A\mathcal{C}_S^H$ parser

This syntax is more permissive. It allows arithmetic operations (division, multiplication, addition, subtraction, modulo, factorial, power) and several names for each standard function. For instance, the reciprocal of the hyperbolic sine can be written "argsh", "argsinh", "arcsh" etc.. So function names are matched with regular expression. For instance, the regex for the function argsinh is

$$[aA](r([cg])?)?[sS](in)?h$$

The other feature we implement for flexibility is implicit multiplications. We understand "a b" as "a*b", "(a)b" as "(a)*b" and "(a)(b)" as "(a)*(b)". But "a(b)" have to remain unchanged because "a" can be a function so, "a(b)" is not "a*(b)".

This transformation is executed before the parser. The method is to launch the lexer a first time. Then we add the symbol "*" between the two symbols in one of these three configurations: ")("; a ")" and a identifier; two consecutive identifiers. Thus, we generate a modified formula with explicit multiplications and we can parse this expression.

The last feature implemented to make this syntax more permissive is to extend functions which are defined on a subset of $\mathbb{R}$ like $\mathbb{N}$. For example, the factorial which is defined on $\mathbb{N}^*$ is extended on $\mathbb{C} \setminus \mathbb{Z}^{-*}$ by using the function Γ . Thus, the expression $(-1/2)!$ has a meaning.

This syntax admits notation as defined in the standard ISO 31-11.

6.2.4 Future work

A parser for $\mathcal{L}\mathcal{A}\mathcal{T}\mathcal{E}\mathcal{X}$ expressions can be implemented. We started to write a $\mathcal{L}\mathcal{A}\mathcal{T}\mathcal{E}\mathcal{X}$ parser but it was more complicated than expected and actually it does not work for big operators like sum, product or integral.

We can also cover other mathematical fields. For instance:

- probability,
- statistics,
- regression analysis,
- logic and set theory,
- discrete mathematics,
- sequences,
- plots.

We can imagine a more lenient syntax for functions which need to bound a variable when there is no ambiguities. For instance, we plan to allow "int(sin(x),0,Pi)" for "int(sin(x),x,0,Pi)" because x is the only variable in this expression. If we are able to determine the list of variables, there is easy to create a new variable. That allows to define functions like $(k, n) \mapsto \binom{n}{k}$ on a larger set using a limit.

¹³<http://www.dabeaz.com/ply/>

An important missing feature is the resolution of differential equations. In fact the `dsolve` function of SymPy does not work on every ODE and does not handle initial conditions. If this feature is not implemented soon, we will use the `dsolve` function of SymPy without initial conditions and determine constants with the function `solve`.

The other feature to implement, is the interpretation of mathematical queries expressed in natural language. For example, "integrate $\sin x \, dx$ from $x=0$ to π " instead of `"integrate(sin(x),x,0,pi)"`.

6.3 Spell Checker

The spell checker is a Python module¹⁴ which aims at correcting spelling mistakes in the sentence. Indeed, errors are often made by the user, and we want to avoid her/his query to fail.

This module uses the *GNU Aspell* library,¹⁵ with a Python wrapper.¹⁶ Aspell is an open source software, and is the standard spell checker on GNU/Linux.

Here are the steps of the algorithm.

Input sentence:

Who is the auhtor of "Le Petit Prince"?

Get all the words of the sentence:

[Who, is, the, auhtor, of, Le, Petit, Prince]

Apply the Aspell corrector on each individual word which is not contained on a quotation in the original sentence:

[Who, is, the, author, of, Le, Petit, Prince]

Replace the words in the sentence, taking care of keeping the punctuation as is:

Who is the author of "Le Petit Prince"?

Remark that it is necessary to keep the words of a quotation unchanged, otherwise we would have modified "Le Petit Prince" since it is not correct English, although it is the correct title of the book.

6.4 OEIS

The OEIS, Online Encyclopedia of Integer Sequences,¹⁷ is a database of integer sequences. It contains their first numbers, name, description, etc.

This module¹⁸ allows the PPP to query this database in order to answer simple questions, like "What follows 1, 2, 4, 8, 16, 32?", or "What is 1, 2, 4, 8, 16, 32?". We use a small OEIS library¹⁹ a member of the project wrote last year.

¹⁴<https://github.com/ProjetPP/PPP-Spell-Checker>

¹⁵<http://aspell.net/>

¹⁶<https://github.com/WojciechMula/aspell-python>

¹⁷<https://oeis.org/>

¹⁸<https://github.com/ProjetPP/PPP-OEIS>

¹⁹<https://github.com/ProgVal/Supybot-plugins/blob/master/OEIS/oeis.py>

Chapter 7

Performances

7.1 Overview

Our query answering tool, *Platypus*, is able to answer correctly a wide range of questions. Here is a non-exhaustive list of questions with correct answers.

7.1.1 Keyword questions

- Definition question.

Sherlock Holmes

- Depth-1 question.

author Les Misérables

7.1.2 English questions

- Definition question.

What is ‘P=NP’?

- Location question.

Where is the capital of New Zealand?

- Depth-1 question, with quotation marks.

Who is the author of ‘Le Petit Prince’?

7.1.3 Math questions

- Approximation of π to the 42th digit.

`N[Pi, 42]`

- Exact value of $\lim_{n \rightarrow \infty} \sum_{i=1}^n \frac{1}{i} - \log(n)$.

`Limit[Sum[1/i, {i,1,n}]-Log[n], n->Infinity]`

7.1.4 Weird questions

- Spelling mistake.

`What is the cappyttal of Franse?`

- Nested question.

`Who are the daughters of the wife of the husband of the wife of the president of the United States?`

7.2 Comparison with existing tools

We study the ability of our tool to successfully answer questions, in comparison to *WolframAlpha*,¹ one of the best question answering engine.

7.2.1 Nested questions

A nested question is a question with several level of knowledge. For instance, the question "What is the birth date of the first person to walk on the moon?" requires firstly to know that Neil ARMSTRONG was the first person to walk on the moon and then to know that he was born on August 5, 1930.

Our search engine tends to be more successful than *WolframAlpha* on such questions. We explain this by the fact that nested questions are generally well structured and therefore good candidates for a grammatical analysis.

Some examples:

Who is the mayor of the capital of Czech Republic?

WolframAlpha	Prague
Platypus	Tomáš Hudeček

When was born the author of "Sherlock Holmes"?

WolframAlpha	no result
Platypus	Sunday, May 22, 1859

What is the citizenship of the daughters of the wife of the president of the United States?

WolframAlpha	Barack Obama
Platypus	United States of America

¹<https://www.wolframalpha.com/>

7.2.2 Conjunction questions

A conjunction is a question which requires to do either an union of sets or an intersection of sets. For instance, the question "Who has been a Jedi and a Sith?" requires to compute the intersection of the members of the Jedi organization and the Sith organization, to find out all the persons that have been in both organizations.

Thank to the union and the intersection operators of our data model, our search engine also provides better answers for such questions.

An example:

Who is the actress of "Pulp Fiction" and "Kill Bill"?	
WolframAlpha	the whole casting of this two films
Platypus	Uma Thurman

7.3 Limitations

Platypus is still limited to the knowledge of *Wikidata* and to some grammatical structures.

Wikidata does not contain any information about measures. We are unable to answer questions like "How far is the Sun?", "How small is an atom?" or "What is the speed of Ariane 5?".

We are also unable to parse correctly sentences with a more complex grammatical structure, like "From which country does the jazz come from?".

Conclusion

The *Projet Pensées Profondes* resulted in the creation of a query answering tool: *Platypus*.²

This engine uses innovative techniques based on the analysis of the grammatical rules existing in natural languages. It enables it to answer correctly to non-trivial questions and to outperform the cutting-edge tools of the domain such as *WolframAlpha* or *Google Knowledge Graph* on some questions.

There still remains a huge amount of work to make it as good as it should be:

- A better database. Contribute to the development of *Wikidata* (for example, to add measures like size, speed...). Add a module to retrieve information from other databases (maybe domain specific ones like *IMdb*³).
- Support more languages like French. Our data model and the *Wikidata* module is already ready to do so, the only strong blocker is the NLP part.
- A better question parsing. Improve the grammatical approach, by training the *Stanford* library and refining our dependency analysis. Improve the machine learning approach, by increasing our dataset.
- Add new modules: cooking recipes, music knowledge, sport statistics, etc.
- Improve the user interface with rich and well designed display of results.
- Advertise *Platypus*, maybe at first through the *Wikidata* community.

We produced a strong basis with a well structured architecture and a well defined data model. We are ready to welcome other developers in the *Projet Pensées Profondes* to enhance *Platypus*.

²<http://askplatyp.us/>

³<http://www.imdb.com/>

Bibliography

- [CWB⁺11] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12:2493–2537, 2011.
- [Def09] Aaron Defazio. Natural language question answering over triple knowledge bases, 2009. <http://users.cecs.anu.edu.au/~adefazio/TripleQuestionAnswering-adefazio.pdf>.
- [dMM13] Marie-Catherine de Marneffe and Christopher D. Manning. Stanford typed dependencies manual, 2013. http://nlp.stanford.edu/software/dependencies_manual.pdf.
- [FI12] Asja Fischer and Christian Igel. An introduction to restricted boltzmann machines. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*, pages 14–36. Springer, 2012.
- [HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.
- [RDF⁺07] Delia Rusu, Lorand Dali, Blaž Fortuna, Marko Grobelnik, and Dunja Mladenić. Triplet extraction from sentences, 2007. http://ailab.ijs.si/delia_rusu/Papers/is_2007.pdf.
- [VK14] Denny Vrandečić and Markus Krötzsch. Wikidata: A free collaborative knowledge base. *Communications of the ACM*, 57:78–85, 2014. <http://cacm.acm.org/magazines/2014/10/178785-wikidata/fulltext>.
- [ZNF06] Amal Zouaq, Roger Nkambou, and Claude Frasson. The knowledge puzzle: An integrated approach of intelligent tutoring systems and knowledge management. In *Proceedings of the 18th IEEE International Conference on Tools with Artificial Intelligence, ICTAI '06*, pages 575–582, Washington, DC, USA, 2006. IEEE Computer Society.
- [ZNF07] Amal Zouaq, Roger Nkambou, and Claude Frasson. Building domain ontologies from text for educational purposes. In *Proceedings of the Second European Conference on Technology Enhanced Learning: Creating New Learning Experiences on a Global Scale, EC-TEL'07*, pages 393–407, Berlin, Heidelberg, 2007. Springer-Verlag.

Appendix A

Question parsing – Triples tree

```
{
  "type": "triple",
  "subject": {
    "type": "triple",
    "subject": {
      "type": "resource",
      "value": "United States"
    },
    "object": {
      "type": "missing"
    },
    "predicate": {
      "type": "resource",
      "value": "president"
    }
  },
  "object": {
    "type": "missing"
  },
  "predicate": {
    "type": "resource",
    "value": "birth place"
  }
}
```

Figure A.1: Triples tree produced by the question parsing grammatical module on *Where was the president of the United States born?*

Appendix B

Question parsing – Grammatical rules analysis

```

undef      : R0
root       : R0
dep        : R1
  aux      : remove
  auxpass  : remove
  cop      : impossible
  arg      : impossible
  agent    : R5
  comp     : R3
    acomp  : R3
    ccomp  : R5
    xcomp  : R3
    pcomp  : R3
    obj     : impossible
      dobj  : R5
      iobj  : R3
      pobj  : R3
  subj     : impossible
    nsubj  : R2 (if the output node is a leaf)
              R5 otherwise
      nsubjpass : R5
      csubj     : impossible
      csubjpass : impossible
cc          : impossible
conj       : apply global transformation (conjunction) and produce Rconj
expl       : remove
mod        : R4
  amod      : merge (if the tag of the output node is neither ORDINAL nor JJS)
                otherwise apply global transformation and produce Rspl
  appos     : R4
  advcl     : R4
  det       : remove
  predet    : remove
  preconj   : remove
  vmod      : R3
  mwe       : merge
    mark    : remove
  advmod    : merge
    neg     : apply global transformation (conjunction) and produce Rconj
  rcmod     : R4
    quantmod : remove
  nn        : merge
  npadvmod  : merge
    tmod     : R3
  num       : merge
  number    : merge
  prep      : R5
  prepc     : R5
  poss      : R5
  possessive : impossible
  prt       : merge
parataxis  : remove
punct      : impossible
ref        : impossible
sdep       : impossible
  xsubj     : R3
goeswith   : merge
discourse  : remove

```

Figure B.1: Replacement rules used by the question parsing grammatical module (impossible means that the relation is not supposed to happen, or is not supported yet by our algorithm)

Appendix C

Wikidata API examples

C.1 Retrieve entities

Goal: get the item Q42 (Douglas Adams).

Request: <http://www.wikidata.org/w/api.php?action=wbgetentities&ids=Q42&format=json&languages=en>

Partial response:

```
{
  "entities": {
    "Q42": {
      "id": "Q42",
      "type": "item",
      "aliases": {
        "en": [{"language": "en", "value": "Douglas␣Noel␣Adams"}]
      },
      "labels": {
        "en": {"language": "en", "value": "Douglas␣Adams"}
      },
      "descriptions": {
        "en": {"language": "en", "value": "English␣writer␣and␣humorist"}
      },
      "claims": {
        "P31": [
          {
            "id": "Q42$F078E5B3-F9A8-480E-B7AC-D97778CBBEF9",
            "mainsnak": {
              "snaktype": "value",
              "property": "P31",
              "datatype": "wikibase-item",
              "datavalue": {
                "value": "Q5",
                "type": "wikibase-entityid"
              }
            },
            "type": "statement"
          }
        ]
      }
    }
  },
}
```

```

        "sitelinks": {
            "enwiki": {"site": "enwiki", "title": "Douglas Adams"}
        }
    }
}

```

C.2 Retrieve entities

Gloal: get items for the label "Douglas Adams".

Request: <http://www.wikidata.org/w/api.php?action=wbgetentities&ids=Q42&format=json&languages=en>

Partial response:

```

{
  "searchinfo":{"search": "Douglas Adams"},
  "search":[
    {
      "id": "Q42",
      "url": "//www.wikidata.org/wiki/Q42",
      "description": "English writer and humorist",
      "label": "Douglas Adams"
    }
  ]
}

```

C.3 Do search with WikidataQuery

Gloal: get items that have "position held" (P39) "president of the united States of America" (Q11696).

Documentation: http://wdq.wmflabs.org/api_documentation.html

Request: [http://wdq.wmflabs.org/api?q=CLAIM\[39:11696\]](http://wdq.wmflabs.org/api?q=CLAIM[39:11696])

Partial response:

```

{
  "items": [23, 76, 91, 207]
}

```

Appendix D

CAS – C_{AL}C^H_{AS} functions

We give here the functions and constants known by C_{AL}C^H_{AS}. As previously, we give regex in some cases. All variable name, function identifier or constant have to match [a-zA-Z_][a-zA-Z0-9_]*.

D.1 Infix operators

Table D.1: List of handled infix operators

Regex	Operator
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo
(^ **)	Power
!	Factorial

D.2 Functions

Table D.2: List of handled functions

Regex	Function
[aA]bs(a)	$\ a\ $
[mM]od(a)	$\ a\ $
[sS]ig(n)?(a)	$\begin{cases} \frac{a}{\ a\ } & \text{if } a \neq 0 \\ 0 & \text{otherwise} \end{cases}$
[sS]ignum(a)	$\begin{cases} \frac{a}{\ a\ } & \text{if } a \neq 0 \\ 0 & \text{otherwise} \end{cases}$
[pP]ow(er)?(a,b)	a^b
[sS]qrt(a)	$\sqrt{a}$
[rR]oot(a,n)	$\sqrt[n]{a}$
[eE]xp(a)	e^a
[lL](n og)(a)	$\ln(a)$
[lL](og10 g)(a)	$\frac{\ln(a)}{\ln(10)}$

– Continued on next page –

Table D.2 – List of handled functions

Regex	Function
[lL](og2 b)(a)	$\frac{\ln(a)}{\ln(2)}$
[lL]og(a,b)	$\log_b(a)$
G(amma AMMA)(a)	$\Gamma(a)$ (EULER's Γ function)
[fF]act(orial)?(a)	$\Gamma(a + 1)$
[cC]os(a)	$\cos(a)$
[sS]in(a)	$\sin(a)$
[tT]an(a)	$\tan(a)$
[cC](osec sc)(a)	$\operatorname{cosec}(a)$ (cosecant)
[sS]ec(a)	$\sec(a)$ (secant)
[cC]ot(an)?	$\cotan(a)$ (cotangent)
[aA](rc)?[cC]os(a)	$\arccos(a)$ (reciprocal of cosine)
[aA](rc)?[sS]in(a)	$\arcsin(a)$ (reciprocal of sine)
[aA](rc)?[tT]an(a)	$\arctan(a)$ (reciprocal of tangent)
[aA](rc)?[cC](sc osec)(a)	$\operatorname{arccosec}(a)$ (reciprocal of cosecant)
[aA](rc)?[sS]ec(a)	$\operatorname{arcsec}(a)$ (reciprocal of secant)
[aA](rc)?[cC]ot(an)?(a)	$\operatorname{arccotan}(a)$ (reciprocal of cotangent)
[cC](os)?h(a)	$\cosh(a)$ (hyperbolic cosine)
[sS](in)?h(a)	$\sinh(a)$ (hyperbolic sine)
[tT](an)?h(a)	$\tanh(a)$ (hyperbolic tangent)
[cC](osec sc)h(a)	$\operatorname{cosech}(a)$ (hyperbolic cosecant)
[sS]ech(a)	$\operatorname{cosech}(a)$ (hyperbolic secant)
[cC]ot(an)?h(a)	$\cotan(a)$ (hyperbolic cotangent)
[aA](r([cg])?)?[cC](os)?h(a)	$\operatorname{acosh}(a)$ (reciprocal of hyperbolic cosine)
[aA](r([cg])?)?[sS](in)?h(a)	$\operatorname{asinh}(a)$ (reciprocal of hyperbolic sine)
[aA](r([cg])?)?[tT](an)?h(a)	$\operatorname{atanh}(a)$ (reciprocal of hyperbolic tangent)
[aA](r([cg])?)?[cC](osec sc)h(a)	$\operatorname{acosech}(a)$ (reciprocal of hyperbolic cosecant)
[aA](r([cg])?)?[sS]ech(a)	$\operatorname{asech}(a)$ (reciprocal of hyperbolic secant)
[aA](r([cg])?)?[cC]ot(an)?h(a)	$\operatorname{acoth}(a)$ (reciprocal of hyperbolic cotangent)
[fF]loor(a)	$\lfloor a \rfloor$
[cC]eil(a)	$\lceil a \rceil$
[bB]inomial(a,b)	$\frac{\Gamma(a+1)}{\Gamma(b+1)\Gamma(a-b+1)}$
[cC]omb(ination)?(a,b)	$\frac{\Gamma(a+1)}{\Gamma(b+1)\Gamma(a-b+1)}$
C(a,b)	$\frac{\Gamma(a+1)}{\Gamma(b+1)\Gamma(a-b+1)}$
[pP]artial[pP]ermutation(a,b)	$\frac{\Gamma(a+1)}{\Gamma(a-b+1)}$
A(a,b)	$\frac{\Gamma(a+1)}{\Gamma(a-b+1)}$
[dD]igamma(a)	$\psi(a)$ (Digamma function)
[bB]eta(a,b)	$B(a, b)$ (Beta function)
[gG]c[dm](a,b)	$a \wedge b$ (gcd)
[hH]cf(a,b)	$a \wedge b$ (gcd)
[lL]cm(a,b)	$a \vee b$ (lcm)
[dD](iff eriv(e at(e ive)))(f,x)	$\frac{\partial f}{\partial x}$ (Derivative)
[iI]nt(egra(te l))(f,x)	$\int f dx$ (Antiderivative)
[iI]nt(egra(te l))(f,x,a,b)	$\int_a^b f dx$ (Tntegral)
[aA]ntiderivative(f,x)	$\int f dx$ (Antiderivative)
[aA]ntiderivative(f,x,a,b)	$\int_a^b f dx$ (Tntegral)
[sS]um(mation)?(e,n,a,b)	$\sum_{n=a}^b e$
[aA]pprox(imation)?(a)	Numeric value with 15 decimals
[aA]pprox(imation)?(a,b)	Numeric value with b decimals
[nN]umeric(a)	Numeric value with 15 decimals

– Continued on next page –

Table D.2 – List of handled functions

Regex	Function
[nN]umeric(a,b)	Numeric value with b decimals
N(a)	Numeric value with 15 decimals
N(a,b)	Numeric value with b decimals
[eE]val(f)?(a)	Numeric value with 15 decimals
[eE]val(f)?(a,b)	Numeric value with b decimals
[sS]impl(if(y ication))?(a)	Simplification of the expression a
[sS]ol(ve ution(s))?(a, x)	Solutions for x of the equation a
[sS]ol(ve ution(s))?(a, x)	Solutions for x of the equation a
[LL]im(it)?(f, x, a)	$\lim_{x \rightarrow a} f$
[LL]im(it)?[LL](f, x, a)	$\lim_{x \rightarrow a^-} f$
[LL]im(it)?[rR](f, x, a)	$\lim_{x \rightarrow a^+} f$

D.3 Constants

Table D.3: List of handled constants

Regex	Constant
[eE]	NAPIER's constant
[pP]i	π
([eE]uler[gG]amma gamma)	EULER-MASCHERONI constant
[iI]nf(ini)?ty	∞
oo	∞
I	i
(GoldenRatio [pP]hi)	φ