# Formal Ontologies for Semantic Text Processing

Bartłomiej Antoni Szymczak

Supervised by prof. Jørgen Fischer Nilsson

Kongens Lyngby 2007
IMM-MSC-2007-79

Technical University of Denmark
Informatics and Mathematical Modelling
Building 321, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@imm.dtu.dk
www.imm.dtu.dk

# Summary

This project is concerned with designing and applying formal ontologies intended for semantic search and retrieval in biomedical texts.

The project begins with a study of techniques for engineering of formal ontologies. The project considers various formalisms and type systems for representation of so–called generative ontologies. The nodes in generative ontologies take form of concept descriptors for describing the various ontological types and relations.

Next a text analyzer is to be designed and implemented, which utilizes the formal ontology together with a simplified grammar for natural language in order to parse the text and generate a proper descriptors for concepts appearing in the text.

This project takes place within the framework of SIABO, which is a project addressing semantic querying of bio–medical text sources in cooperation with a major medical company.

The above also summary serves as an official project description.

# Contents

**Summary** i

**Contents** iii

**1 Foreword** 1

**2 Motivation** 3

2.1 Importance of fast information retrieval . . . . . . . . 3

2.2 Keyword–based search and its deficiencies . . . . . . 4

2.2.1 Definition of keyword–based search . . . . . . 5

2.2.2 Problems with keyword–based search . . . . . 6

2.2.2.1 Scattered keywords . . . . . . . . . . 6

2.2.2.2 Multiple queries . . . . . . . . . . . 7
2.2.2.3 Lack of underlying ontology . . . . . 8
2.2.2.4 Language dependency . . . . . . . . 8

**3 Formal ontologies 9**

3.1 What is an ontology? . . . . . . . . . . . . . . . . . . 9
3.2 Formalisms used to represent formal ontologies . . . . 10
3.2.1 First Order Predicate Logic . . . . . . . . . . 10
3.2.1.1 Binary relations . . . . . . . . . . . . 12
3.2.1.2 Algebraic operations . . . . . . . . . 13
3.2.1.3 Distinction between class and instance 14
3.2.2 Description Logics . . . . . . . . . . . . . . . . 15
3.2.2.1 Restricting FOL . . . . . . . . . . . 15
3.2.2.2 TBox . . . . . . . . . . . . . . . . . 16
3.2.2.3 ABox . . . . . . . . . . . . . . . . . 17
3.2.2.4 Translating from DL to FOL . . . . 18
3.2.3 OWL . . . . . . . . . . . . . . . . . . . . . . . 19
3.2.4 Peirce algebras . . . . . . . . . . . . . . . . . 19
3.2.5 Context–free grammar . . . . . . . . . . . . . 19

3.3 Types of formal ontologies . . . . . . . . . . . . . . . 20
3.3.1 Top–level ontologies . . . . . . . . . . . . . . 20
3.3.2 Domain ontologies . . . . . . . . . . . . . . . 22
3.3.3 Merging top–level and domain ontologies . . . 22

**4 Linguistic analysis 25**

4.1 Human language understanding . . . . . . . . . . . . 25
4.2 Language as a protocol . . . . . . . . . . . . . . . . 26
4.2.1 Parts of speech . . . . . . . . . . . . . . . . . 27
4.2.2 Necessity of part–of–speech tagging . . . . . . 28
4.3 Grammar as the structure of English . . . . . . . . . 29
4.3.1 Rules and structure . . . . . . . . . . . . . . . 29
4.3.2 Rules for natural languages . . . . . . . . . . 30
4.3.3 Shallow context–free grammar for English . . 31
4.4 Lexical resources . . . . . . . . . . . . . . . . . . . . 33
4.4.1 WordNet . . . . . . . . . . . . . . . . . . . . . 33
4.4.2 FrameNet . . . . . . . . . . . . . . . . . . . . 35

**5 Ontological semantics 37**

5.1 Peirce–algebraic ontological semantics for English . . 37
5.2 Semantical incompleteness . . . . . . . . . . . . . . . . 41
5.3 Rephrasing . . . . . . . . . . . . . . . . . . . . . . . . 42
5.4 Incorrect sentences . . . . . . . . . . . . . . . . . . . 44
5.5 Correcting the author . . . . . . . . . . . . . . . . . . 44
5.6 Relations vs. classes . . . . . . . . . . . . . . . . . . 46

**6 Implementation 49**

6.1 Choice of programming language . . . . . . . . . . . 49
6.1.1 StandardML . . . . . . . . . . . . . . . . . . . . . 50
6.1.2 Prolog . . . . . . . . . . . . . . . . . . . . . . . . 52
6.1.3 Mercury . . . . . . . . . . . . . . . . . . . . . . . 53
6.1.3.1 Type system . . . . . . . . . . . . . . 54
6.1.3.2 Determinism declaration . . . . . . . 54
6.1.3.3 Mode system . . . . . . . . . . . . . . 55
6.1.3.4 Pure declarativeness . . . . . . . . . 57
6.1.4 Java . . . . . . . . . . . . . . . . . . . . . . . . . 58
6.2 Ontological natural language analyzer . . . . . . . . . 58
6.2.1 Module `aux_io` . . . . . . . . . . . . . . . . . . . 58

6.2.2 Module `createOntology` . . . . . . . . . . . . . 60
6.2.3 Module `grabber` . . . . . . . . . . . . . . . . . 63
6.2.3.1 Types and interface . . . . . . . . . . 63
6.2.3.2 Syntactic hinting . . . . . . . . . . . . 67
6.2.3.3 Part–of–speech tagging . . . . . . . . . 68
6.2.3.4 Recognizing relations . . . . . . . . . . 69
6.2.3.5 Ontological restrictions . . . . . . . . . 70
6.2.4 Module `html` . . . . . . . . . . . . . . . . . . . 71
6.2.5 Module `main` . . . . . . . . . . . . . . . . . . . 72
6.2.6 Module `lexicon` . . . . . . . . . . . . . . . . . 73
6.2.7 Module `obo` . . . . . . . . . . . . . . . . . . . . 75
6.2.8 Module `ontology` . . . . . . . . . . . . . . . . . 77
6.2.9 Module `pipe` . . . . . . . . . . . . . . . . . . . 78
6.2.10 Module `wn` . . . . . . . . . . . . . . . . . . . . 81
6.3 Web interface . . . . . . . . . . . . . . . . . . . . . . 82
6.4 Exemplary run of the system . . . . . . . . . . . . . . 84
6.5 Further extensions . . . . . . . . . . . . . . . . . . . 88
6.5.1 Ontological search engine . . . . . . . . . . . . . 88

6.5.2 Ad–hoc concepts . . . . . . . . . . . . . . . . 90

7 Conclusions 93

A Source code and auxiliary files 95

A.1 File ontologies.txt . . . . . . . . . . . . . . . . . . 95

A.2 File index.jsp . . . . . . . . . . . . . . . . . . . . . 97

A.3 File disambiguate.jsp . . . . . . . . . . . . . . . . . 98

A.4 File M2PL.java . . . . . . . . . . . . . . . . . . . . . 103

A.5 File aux_io.m . . . . . . . . . . . . . . . . . . . . . 104

A.6 File pipe.m . . . . . . . . . . . . . . . . . . . . . . . 110

A.7 File wn.m . . . . . . . . . . . . . . . . . . . . . . . 112

A.8 File ontology.m . . . . . . . . . . . . . . . . . . . . 119

A.9 File main.m . . . . . . . . . . . . . . . . . . . . . . 123

A.10 File lexicon.m . . . . . . . . . . . . . . . . . . . . 126

A.11 File obo.m . . . . . . . . . . . . . . . . . . . . . . . 132

A.12 File startDaemon.sh . . . . . . . . . . . . . . . . . . 149

A.13 File grabber.m . . . . . . . . . . . . . . . . . . . . 150

A.14 File createOntology.m . . . . . . . . . . . . . . . . 156

A.15 File html.m . . . . . . . . . . . . . . . . . . . . . . . 165
A.16 File parse.m . . . . . . . . . . . . . . . . . . . . . . . 181
A.17 File compile.sh . . . . . . . . . . . . . . . . . . . . . 183
A.18 File types.sml . . . . . . . . . . . . . . . . . . . . . . 183
A.19 File analyze.sml . . . . . . . . . . . . . . . . . . . . . 184
A.20 File Earley.sig . . . . . . . . . . . . . . . . . . . . . 190
A.21 File Earley.sml . . . . . . . . . . . . . . . . . . . . . 191
A.22 File Insulin.txt . . . . . . . . . . . . . . . . . . . . . 199
A.23 File compile.sh . . . . . . . . . . . . . . . . . . . . . 201
A.24 File parse.c . . . . . . . . . . . . . . . . . . . . . . . 202
A.25 File insulin_OBO_concepts.txt . . . . . . . . . . . . . 204
A.26 File FindConcepts.java . . . . . . . . . . . . . . . . . 210
A.27 File createLists.sh . . . . . . . . . . . . . . . . . . . 211
A.28 File hand-grabbed_insulin_concepts.txt . . . . . . 211
A.29 File getConceptNames.sh . . . . . . . . . . . . . . . . 216
A.30 File increase.owl . . . . . . . . . . . . . . . . . . . . 217

**B Introduction to lattices 219**

B.1 Posets . . . . . . . . . . . . . . . . . . . . . . . . . . . 219

B.2 Top and bottom . . . . . . . . . . . . . . . . . . . . . 220

B.3 Upper and lower bounds . . . . . . . . . . . . . . . . 221

B.4 Lattice as poset . . . . . . . . . . . . . . . . . . . . . 222

B.5 Hasse diagrams . . . . . . . . . . . . . . . . . . . . . 222

B.6 *isa* as partial order . . . . . . . . . . . . . . . . . . . 222

B.7 Lattices as algebras . . . . . . . . . . . . . . . . . . . 223

B.8 Dual nature of lattices . . . . . . . . . . . . . . . . . 223

B.9 Atoms . . . . . . . . . . . . . . . . . . . . . . . . . . 224

**Bibliography** **225**

Chapter 1

# Foreword

This text is intended for a fellow student with a computer science background. I will assume that the reader has a common theoretical computing knowledge. On the other hand, while commencing on this project, I have encountered many obstacles, which have never been an issue during my studies:

- The domain of biomedicine has been at first completely unknown to me. I had to spend a substantial amount of time on understanding some of the most important sentences in the biomedical text, which I have been working with. Obviously, it is impossible to design a system, which should understand the text, if the designer himself doesn't understand it. It is advisable for the Reader to have a look at the excerpt from Wikipedia entry on insulin, which has been the main analyzed text in the project. The text is presented in Appendix A.22 on page 201.
- The area of linguistics has also been completely new to me.

Computer Science students get a lot of insight into formal languages, but no courses are given on natural languages, including English.

- The fact that I am not a native English speaker also posed a small obstacle.
- The Mercury programming language has been chosen by me for the implementation of the system. The language itself isn't taught in any of the courses given at DTU, so I had to learn it from basics.

I am aware that the Reader may face similar problems. Therefore I have decided to illustrate the discussion with a lot of examples. I have also put an extra effort into the explanation of things, which I myself have found to be difficult to understand. I have decided to keep the language of this thesis on a less formal level, in order to ease the reading of the text.

Overcoming all of the problems and finishing the project in the limited five months period would be impossible without great help from my supervisor, prof. Jørgen Fischer Nilsson. I would like to thank him for his support and time spent on helping me.

Chapter 2

# Motivation

## 2.1 Importance of fast information retrieval

In todays world, people increasingly depend on fast access to information. We have entered the era of "information society", where Internet search engines are the primary tool for each computer user.

Subsequently, any improvement in the way people search for information constitutes a big leap forward for humanity. Simple calculation shows that if by improving information retrieval techniques we could save on average one second of time per person daily, this

amounts to:

$$\begin{aligned}
& (\textit{time saved}) \cdot (\textit{Earth population}) \\
= \;& 1\text{s} \cdot 6.5 \cdot 10^9 \\
= \;& 6.5 \cdot 10^9\text{s} \\
= \;& 6.5 \cdot 10^9\text{s} \cdot \frac{1\text{min}}{60\text{s}} \\
= \;& 1.08(3) \cdot 10^8\text{min} \cdot \frac{1\text{h}}{60\text{min}} \\
= \;& 1.80(5) \cdot 10^6\text{h} \cdot \frac{1\text{d}}{24\text{h}} \\
\approx \;& 75231\text{d} \cdot \frac{1\text{y}}{365\text{d}} \\
\approx \;& 206\text{y}
\end{aligned}$$

Thus, each day, we could save two centuries of humanity's time by one–second–per–person improvement. [1]

In section 2.2 I explain why most people unnecessarily waste not one second, but much more time on machine–aided searching using today's state–of–the–art technology.

## 2.2 Keyword–based search and its deficiencies

The big Internet boom of 1990s was followed by a huge increase in search engine research. Many successful companies emerged, most notably Google, whose search engine [7] has revolutionized the Internet.

[1] Similar calculation shows that at the same time we save $100\text{W} \cdot 1.8 \cdot 10^6\text{h} \approx 1.8 \cdot 10^5\text{kWh}$ of electrical energy.

### 2.2.1 Definition of keyword–based search

All of the current search engines in wide use are keyword–based. Formally we can describe the behavior of keyword search using first order predicate logic[2] as follows.

We can represent an empty list using the constant *nil*. A non–empty list will be constructed using binary functor $l$, whose first argument is a head of the list, and second argument is the tail of the list, being itself a list.

We assume that documents are stored in *document* unary factual clause, which holds a list of keywords. An exemplary database of documents $D$ could look as follows:

$$document(l(insulin, l(forces, l(storage, nil)))) \tag{2.1}$$

The following set of formulae describes how a keyword–based search engine works:

$$\begin{aligned}
&\forall D, Q(document(D) \wedge \\
&\qquad \forall K(member(K, Q) \rightarrow member(K, D)) \\
&\qquad \rightarrow query(Q, D)) && (2.2)\\
&\forall X, L(member(X, l(X, L))) && (2.3)\\
&\forall X, Y, L(member(X, L) \rightarrow member(X, l(Y, L))) && (2.4)
\end{aligned}$$

As the set of formulae 2.2 – 2.4 show, querying with a list of keywords returns a document only if it contains all the keywords found in the query.

[2] First order predicate logic is described in Section 3.2.1.

Some engines differ in behavior in one or more aspects:

- Returning documents, which contain only subset of the query keywords.
- Returning documents, which contain keywords related to the query keywords, e.g. $forced$ instead of $forces$.

Let us call the set of formulae 2.2 – 2.4 $KBSE$. Coupled together with our exemplary database $D$ from formula 2.1, we can make the following exemplary inference:

$$\begin{aligned} KBSE \cup D \ \models \ & query(l(forces, l(storage, nil)), \\ & \quad l(insulin, l(forces, l(storage, nil)))) \end{aligned} \tag{2.5}$$

### 2.2.2 Problems with keyword–based search

The keyword querying has some advantages. One of them is simplicity of implementation. The search engine presented in formulae 2.2 – 2.4 can be implemented in Prolog in few lines. Unfortunately, such a primitive approach to information retrieval results in a number of disadvantages, which are enumerated below with accompanying examples.

#### 2.2.2.1 Scattered keywords

Let us consider the following query: `insulin causes storage`. The user is interested in the concept of insulin forcing a storage of something.

Most search engines will return an article, in which `insulin` and `storage` are present in one sentence, and `causes` is present in another sentence. This behavior follows from the fact that the search engine does not recognize concepts appearing in the text, but rather entertains the idea of words appearing in the whole document.

#### 2.2.2.2 Multiple queries

Most search engines have a feature, which allows the user to search for a sequence of keywords, which must appear contiguously in the returned document. Usually, such a sequence must be enclosed in double quotes in the query. This allows the user to overcome the problem of scattered keywords to a certain extent. One could, e.g. search for `''insulin causes storage''`, so that documents with those keywords scattered around will not be shown in the result.

Unfortunately, the user is usually forced to fire multiple queries, as the document author might have written the text in a different way. The author could use different wording, phrased the document in a different manner, etc.

As an example, the user might be forced to query with the following sequences of keywords:

- `''insulin causes storage''`
- `''insulin is causing storage''`
- `''storage caused by insulin''`
- ...

For each query different articles will be returned (if any).

An ontological search engine should realize that all of those sequences of keywords have related ontological semantics. It should therefore be necessary to simply fire one query only.

#### 2.2.2.3 Lack of underlying ontology

Without an underlying ontology, the search engine won't be able to realize that "forcing" is in fact a kind of "causing". The user therefore needs to fire up additional queries:

- `''insulin forces storage''`
- `''insulin is forcing storage''`
- `''storage forced by insulin''`
- ...

#### 2.2.2.4 Language dependency

The deficiencies of keyword–based searching are caused by the fact that the engine compares text, not the semantics. If the engine was able to extract semantics from the text and from the query and compare them, instead of comparing the text itself, many problems could be easily overcome.

Comparing the ontological concepts instead of the words that describe them could allow new ways of information retrieval, e.g. one could query in English, but get an article in Danish as a result.

CHAPTER 3

# Formal ontologies

## 3.1 What is an ontology?

A nice introduction and overview of ontologies and their various definitions is provided in [11]. The word “Ontology” appeared thousands of years ago, where ancient philosophers tried to explain the essence of being. In such context, we use the capitalized word “Ontology”. Please consult [11] for an overview of the fascinating problems that philosophers have been dealing with.

However, the current text deals with formal ontologies, or simply ontologies, written with lower–case “o”. Formal ontology can be understood as specification of a conceptualization. It is crucial that the ontology is specified in a formal way, so that it is machine–readable and understandable. A conceptualization can be thought of as a view of the world shared by some people. Of course none of the ontologies can describe a full conceptualization of the whole

world. In most cases an ontology attempts to describe only a specific domain.

## 3.2 Formalisms used to represent formal ontologies

As the ontology must be formal and understandable by the computer, a particular formalism must be used in order to represent it. Some of the most common are presented in the following sections.

### 3.2.1 First Order Predicate Logic

I assume that the reader is familiar with First Order Predicate Logic (FOL), as it is a basic tool for every computer scientist. Otherwise please consult a text on the subject, e.g. [5].

Since the FOL notation differs from one source to the other, I present the syntax used in the current text in Figure 3.1 on page 11.

As can be seen from the Figure, there is a syntactical distinction between predicates/functors and variables, i.e. variable names start with a capital letter. This is the convention adopted from Prolog/Mercury.

Please observe that the syntax is pure, in the sence that it does not allow any mathematical contaminations. In particular, the symbol of equality (=) is not part of FOL. Instead, one can use the following predicate for testing whether two ground terms are identical:

$$equal(X, X) \tag{3.1}$$

$$
\begin{array}{rcl}
\langle\text{formula}\rangle & ::= & \langle\text{predicate}\rangle(\langle\text{termlist}\rangle) \\
& | & \langle\text{predicate}\rangle \\
& | & \neg\langle\text{formula}\rangle \\
& | & \langle\text{formula}\rangle \vee \langle\text{formula}\rangle \\
& | & \langle\text{formula}\rangle \wedge \langle\text{formula}\rangle \\
& | & \langle\text{formula}\rangle \rightarrow \langle\text{formula}\rangle \\
& | & \langle\text{formula}\rangle \leftarrow \langle\text{formula}\rangle \\
& | & \langle\text{formula}\rangle \leftrightarrow \langle\text{formula}\rangle \\
& | & \langle\text{quantifier}\rangle\langle\text{variablelist}\rangle(\langle\text{formula}\rangle) \\
\\
\langle\text{term}\rangle & ::= & \langle\text{variable}\rangle \\
& | & \langle\text{functor}\rangle \\
& | & \langle\text{functor}\rangle(\langle\text{termlist}\rangle) \\
\\
\langle\text{termlist}\rangle & ::= & \langle\text{term}\rangle \\
& | & \langle\text{term}\rangle, \langle\text{termlist}\rangle \\
\langle\text{variablelist}\rangle & ::= & \langle\text{variable}\rangle \\
& | & \langle\text{variable}\rangle, \langle\text{variablelist}\rangle \\
\\
\langle\text{lowercase}\rangle & ::= & a|b|\ldots|x|y|z \\
\langle\text{uppercase}\rangle & ::= & A|B|\ldots|X|Y|Z \\
\langle\text{letters}\rangle & ::= & \epsilon \\
& | & \langle\text{lowercase}\rangle\langle\text{letters}\rangle \\
& | & \langle\text{uppercase}\rangle\langle\text{letters}\rangle \\
\\
\langle\text{variable}\rangle & ::= & \langle\text{uppercase}\rangle\langle\text{letters}\rangle \\
\langle\text{predicate}\rangle & ::= & \langle\text{lowercase}\rangle\langle\text{letters}\rangle \\
\langle\text{functor}\rangle & ::= & \langle\text{lowercase}\rangle\langle\text{letters}\rangle
\end{array}
$$

Figure 3.1: Abstract syntax of FOL used throughout the text.

#### 3.2.1.1 Binary relations

When reasoning about properties of various binary relations, we shall use the following FOL notation for expressing them:

$$r(\phi, \rho, \psi)$$

$r$ is a distinct predicate, expressing that $\phi$ is related to $\psi$ by relation $\rho$. E.g. we could express that *liver* is *part_of human_body* by

$$r(liver, part_of, human_body)$$

If we would like to obtain the view of the relation as a set of pairs, we can use the following mathematical definition for arbitrary relation $\rho$:

$$\rho = \{(X, Y)|r(X, \rho, Y)\} \tag{3.2}$$

Similarly, we can use the following mathematical definition for arbitrary relation $\rho$ to obtain the set $A_\rho$ of elements on which the relation is defined:

$$A_\rho = \{X|r(X, \rho, Y) \vee r(Y, \rho, X)\} \tag{3.3}$$

An alternative FOL representation is to use new predicate for each relation, as in:

$$part_of(liver, human_body)$$

This has the disadvantage that general reasoning about relations would require higher order predicates. The notation proposed resembles more natural language and mathematical notation, where most relations are used in infix form. Nevertheless for the Mercury implementation I use both notations, depending on which one is more convenient.

In the notation we propose it is easy to define general properties of relations. The following is assumed to hold throughout the text:

$$\begin{aligned} reflexive(R) &\leftrightarrow \forall(r(X,R,Y) \rightarrow r(X,R,X) \wedge r(Y,R,Y)) \\ antisymmetric(R) &\leftrightarrow \forall(r(X,R,Y) \wedge r(Y,R,X) \rightarrow r(X,equal,Y)) \\ transitive(R) &\leftrightarrow \forall(r(X,R,Y) \wedge r(Y,R,Z) \rightarrow r(X,R,Z)) \end{aligned}$$

We can now use these general definitions to express properties of different relations, e.g.:

$$transitive(part_of)$$ [1]

Please observe that the following definition of reflexivity is insufficient:

$$reflexive(R) \leftrightarrow \forall X(r(X,R,X)) \tag{3.4}$$

The problem is that we should only require reflexivity to hold for objects, which in fact engage in the relation, not for all objects in the universe.

#### 3.2.1.2 Algebraic operations

We shall use the following FOL notation for expressing algebraic operations:

$$o(\phi,\omega,\psi,\gamma)$$

$o$ is a distinct predicate, expressing that $\omega$ applied to $\phi$ and $\psi$ results in $\gamma$. E.g. we could express that Peirce product [2] of *has_color* relation and *green* is *green_things* by:

$$o(has_color, :, green, green_things)$$

[1] This particular statement can be disputed. Please refer to [10] for a more detailed discussion of the `part_of` relation.

[2] Peirce product algebraic operation is described in section 3.2.4 on page 19.

We can use the following mathematical definition for arbitrary operation $\omega$ to obtain the set $A_\omega$ of elements on which the operation is defined:

$$A_\omega = \{X | o(X, \rho, Y, Z) \vee o(Y, \omega, X, Z)\} \tag{3.5}$$

Similarly as for binary relations, it is easy to define general properties of algebraic operations. The following is assumed to hold throughout the text:

$$\begin{aligned}
idempotent(O) &\leftrightarrow \forall(o(X, O, Y, Z) \rightarrow \\
&\qquad o(X, O, X, X) \wedge o(Y, O, Y, Y)) \\
commutative(O) &\leftrightarrow \forall(o(X, O, Y, Z) \rightarrow o(Y, o, X, Z)) \\
associative(O) &\leftrightarrow \forall(o(Y, O, Z, W) \wedge o(X, O, W, V) \\
&\qquad \rightarrow o(X, O, Y, U) \wedge o(U, O, Z, V)) \\
absorptive(O_1, O_2) &\leftrightarrow \forall(o(X, O_2, Y, U) \rightarrow o(X, O_1, U, X))
\end{aligned}$$

We can now use these general definitions to express properties of different operations, e.g. commutativity of set intersection:

$$commutative(\cap)$$

#### 3.2.1.3 Distinction between class and instance

In the biomedical domain we do not see a clear need for distinguishing between classes (descriptions of groups of individuals) and instances (individuals). For example, we could have a class *liver*, which represents a set of all possibly imaginable livers, and an instance $John_Smith's_liver$, representing a particular liver.

Since we are concerned with understanding what sentences in biomedical papers are about, we can forget about instances, as articles

rarely talk about some particulars. The general tendency is to describe things more generally. Even if the article talks about a particular patient, her name will not be repeated in every sentence. Since we do not analyze the context, but analyze each sentence individually, the objects will still represent classes more than instances. Consider the following example:

This paper is about John Smith. His liver stores glucose. (3.6)

Even though the paper talks about some specific liver, our system won't be able to notice that, because it analyzes each sentence independently.

Hence in the ontological analysis described later, I do not make any use of instances, or particulars, but I only use classes only. Anyway, a particular object can be conceived as a singleton set comprising that object, so an instance can be represented by a singleton class.

A more thorough discussion of the notion of instances and classes, please refer to [23, 15].

### 3.2.2 Description Logics

#### 3.2.2.1 Restricting FOL

As described in [2] FOL has a few drawbacks when it comes to applying it in realistic ontology–based systems. Many problems are undecidable and most of the decidable ones are intractable. Description Logics (DL) might serve as a useful replacement. DL are very much restricted compared to FOL, they are on the other hand decidable and computationally efficient.

#### 3.2.2.2 TBox

The terminology of domain in question is described in so–called TBox. The terminology consists of concepts and roles. Concepts are equivalent to unary predicates in FOL (they can represent classes), while roles are equivalent to binary predicates (they represent relations). Atomic concepts and roles are common to all DL languages. Various languages of DL family differ by how expressive is the construction of complex concepts and roles.

Typical systems allow to perform various forms of reasoning based on the terminology introduced by the TBox, e.g. whether the specification is satisfiable or whether one description subsumes the other.

Most DL languages are subsets of the $\mathcal{ALCN}$ language. It provides a set of constructors, which allows to describe complex concepts in terms of atomic ones.

The following BNF rules describe the available $\mathcal{ALCN}$ constructors and their syntax (where $\langle$atomic concept$\rangle$ is an arbitrary atomic concept, $\langle$atomic role$\rangle$ is an arbitrary atomic role, $\top$ is the universal

concept, $\bot$ is the bottom concept and $\langle \text{n} \rangle$ is a natural number):

$$
\begin{array}{rcl}
\langle \text{concept} \rangle & ::= & \langle \text{atomic concept} \rangle \\
 & | & \top \\
 & | & \bot \\
 & | & \neg \langle \text{concept} \rangle \\
 & | & \langle \text{concept} \rangle \sqcap \langle \text{concept} \rangle \\
 & | & \langle \text{concept} \rangle \sqcup \langle \text{concept} \rangle \\
 & | & \forall \langle \text{atomic role} \rangle . \langle \text{concept} \rangle \\
 & | & \exists \langle \text{atomic role} \rangle . \langle \text{concept} \rangle \\
 & | & \geqslant \langle \text{n} \rangle \langle \text{atomic role} \rangle \\
 & | & \leqslant \langle \text{n} \rangle \langle \text{atomic role} \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle \text{terminological axiom} \rangle & ::= & \langle \text{concept} \rangle \sqsubseteq \langle \text{concept} \rangle \\
 & | & \langle \text{concept} \rangle \equiv \langle \text{concept} \rangle \\
\langle \text{definition} \rangle & ::= & \langle \text{atomic concept} \rangle \sqsubseteq \langle \text{concept} \rangle \\
 & | & \langle \text{atomic concept} \rangle \equiv \langle \text{concept} \rangle
\end{array}
$$

A set of $\langle \text{definition} \rangle$s is called a TBox if each symbolic name is defined at most once.

#### 3.2.2.3 ABox

The ABox provides set of assertions expressed in terms of terminology defined by the TBox. There are only two types of assertions one can make: concept assertions of the form $C(a)$ and role assertions of the form $R(a, b)$, where $C$ is a concept, $R$ is a role, and $a, b$ are names of individuals.

Existing systems allow checking whether assertions expressed by the ABox are consistent and whether a particular individual is an instance of a given concept.

#### 3.2.2.4 Translating from DL to FOL

Let us define a function $f$, which given a $\mathcal{ALCN}$ formula, computes a FOL formula with equivalent meaning:

$$
\begin{aligned}
f(A) &= A(X) \text{ for } A \text{ being atomic concept} \\
f(\top) &= true \\
f(\bot) &= false \\
f(\neg C) &= \neg f(C) \\
f(C_1 \sqcap C_2) &= f(C_1) \wedge f(C_2) \\
f(C_1 \sqcup C_2) &= f(C_1) \vee f(C_2) \\
f(\forall R.C) &= \forall Y (R(X,Y) \rightarrow \forall X(equal(Y,X) \rightarrow f(C))) \\
f(\exists R.C) &= \exists Y (R(X,Y) \wedge \forall X(equal(Y,X) \rightarrow f(C))) \\
f(\geqslant n\ R) &= \exists Y_1 \cdots \exists Y_n \left( R(X,Y_1) \wedge \ldots \wedge R(X,Y_n) \wedge \bigwedge_{i<j} \neg equal(Y_i,Y_j) \right) \\
f(\leqslant n\ R) &= \forall Y_1 \cdots \forall Y_{n+1} \left( R(X,Y_1) \wedge \ldots \wedge R(X,Y_{n+1}) \rightarrow \bigvee_{i<j} equal(Y_i,Y_j) \right) \\
f(C_1 \sqsubseteq C_2) &= f(C_1) \rightarrow f(C_2) \\
f(C_1 \equiv C_2) &= f(C_1) \leftrightarrow f(C_2)
\end{aligned}
$$

The definitions are not trivial. Let's consider why we need $\forall X(equal(Y,X) \rightarrow f(C))$. $f(C)$ is a FOL formula, having at most one free variable, $X$. If we quantify it universally and add another free variable $Y$, and require that $f(C)$ holds provided $equal(Y,X)$, then we obtain a simple renaming of variables (from $X$ to $Y$). Therefore $\forall X(equal(Y,X) \rightarrow f(C))$ is like $f(C)$, except that the free variable has been renamed from $X$ to $Y$.

As we can transform each $\mathcal{ALCN}$ formula into FOL formula, the

notions of interpretation, model, satisfiability, etc. carry over from FOL to DL.

For more information about semantics of DL, please consult [2].

### 3.2.3 OWL

OWL is a member of the ontology markup languages family. Almost all markup languages share the XML syntax in order to be easily parsable by computer. An exemplary concept definition is presented in Appendix A.30 on page 218.

### 3.2.4 Peirce algebras

Peirce algebra is a two–sorted algebra, which combines operations on sets and relations. Hence, it is very useful for ontological applications, where classes can be conceived as sets, and where relations play major role. A very detailed introduction is available in [8].

Of particular interest for us is the Peirce product:

$$\rho : \phi = \{X | \exists Y ((X, Y) \in \rho \wedge Y \in \phi)\},$$

where $\rho$ is a relation and $\phi$ is a class understood as a set.

### 3.2.5 Context–free grammar

It is possible to use the familiar notion of context–free grammars for representing ontologies, as described in [1]. In such an application classes are represented by nonterminal symbols and class subsumption relationship is replaced by the string derivation.

As an example, consider the following exemplary grammar, which represents an ontology created by combining Basic Formal Ontology with Gene Ontology. The grammar is presented in Figure 3.2 on page 21.

## 3.3 Types of formal ontologies

Formal ontologies can be generally divided into lightweight and heavyweight ontologies. The former are expressed using a very simple formalism, usually variable–free, so that they serve as an overview of the conceptualization. The latter call for more complicated formalism, as they usually model the conceptualization in a very detailed manner. In heavyweight ontologies a large amount of the information is implicit and is left to be inferred by the engine, hence the name.

Another categorization of ontologies focuses on the level of reality, which they describe. This division is presented in the following sections.

### 3.3.1 Top–level ontologies

Top–level ontologies, as the name suggests, are supposed to describe the most general view of the world. For a nice overview of different upper level ontologies, please refer to [11].

Out of many available top–level ontologies, I have decided to focus on Basic Formal Ontology (BFO). Like most top–level ontologies, BFO divides the world into enduring entities, called continuants, and 4–dimensional entities, called occurrents. A particularly good description of BFO can be found in [19]:

$$
\begin{array}{rcl}
\langle\text{Entity}\rangle & ::= & \langle\text{Continuant}\rangle \\
 & | & \langle\text{Occurrent}\rangle \\
 & & \\
\langle\text{Occurrent}\rangle & ::= & \langle\text{Occurrent}\rangle\langle\text{Occurrent role list}\rangle \\
 & & \\
\langle\text{Occurrent role list}\rangle & ::= & \epsilon \\
 & | & [\langle\text{Occurrent role}\rangle]\langle\text{Occurrent role list}\rangle \\
 & & \\
\langle\text{Occurrent role}\rangle & ::= & CBY : \langle\text{Occurrent}\rangle \\
 & | & LOC : \langle\text{Continuant}\rangle \\
 & | & WRT : \langle\text{Entity}\rangle \\
 & | & BMO : \langle\text{Entity}\rangle \\
 & | & POF : \langle\text{Occurrent}\rangle \\
 & | & \ldots \\
 & & \\
\langle\text{Continuant}\rangle & ::= & \langle\text{Continuant}\rangle\langle\text{Continuant role list}\rangle \\
 & & \\
\langle\text{Continuant role list}\rangle & ::= & \epsilon \\
 & | & [\langle\text{Continuant role}\rangle]\langle\text{Continuant role list}\rangle \\
 & & \\
\langle\text{Continuant role}\rangle & ::= & LOC : \langle\text{Continuant}\rangle \\
 & | & POF : \langle\text{Continuant}\rangle \\
 & & \\
\langle\text{Occurrent}\rangle & ::= & \langle\text{ProcessualEntity}\rangle | \ldots \\
\langle\text{ProcessualEntity}\rangle & ::= & \langle\text{Process}\rangle | \ldots \\
\langle\text{Process}\rangle & ::= & \langle\text{biological_process}\rangle \\
 & & \\
\langle\text{Continuant}\rangle & ::= & \ldots
\end{array}
$$

Figure 3.2: Grammatical specification of an exemplary ontology.

> BFO grows out of a philosophical orientation which overlaps with that of DOLCE and SUMO. Unlike these, however, it is narrowly focused on the task of providing a genuine upper ontology which can be used in support of domain ontologies developed for scientific research, as for example in biomedicine within the framework of the OBO Foundry.

### 3.3.2 Domain ontologies

Domain ontologies consist of classes characteristic for a given domain. Since the current text is concerned with biomedical domain, we shall be interested in biomedical ontologies.

There exists a large collection of biomedical ontologies, which are freely available, called Open Biomedical Ontologies Foundry (OBO) [21]. It consists of many domain–specific ontologies, gathered together in a unified format.

It is worth noticing that a domain ontology must be constructed by domain experts, not by software engineers who want to use the ontology. Hence, in the current project such an ontology is considered to be the input to the system, rather than a part of it. Various papers focus on the topic of creating a high–quality ontologies, e.g. [18].

### 3.3.3 Merging top–level and domain ontologies

Merging top–level ontology with a particular domain ontology is not always easy. Top–level ontologies are usually an attempt at a "perfect" categorization of the world around us. They might be very abstract, in contrast to the domain ontologies. The latter usually attempt to be as complete as possible, trying to include all the

concepts found in a particular domain. Therefore they tend to lose the perfectionistic view of the world. It might therefore be difficult to merge the two into one, coherent ontology.

The reason why we would like to merge the two is that both are insufficient if used alone. Top–level ontologies are simply very small, normally having less than a hundred concepts. Therefore they cannot be used for analyzing natural language, where much more concepts are found. The domain ontologies are, on the other hand, relatively big, reaching even hundreds of thousands of concepts. Unfortunately, they are always focused on the domain, so they are missing the inter–domain concepts. Consider the sentence:

$$\text{Insulin forces storage of glycogen.} \tag{3.7}$$

We can find three domain–specific concepts in the sentence: the chemical substances insulin and glycogen, and the biological process of storing a substance. Unfortunately, the concept of action of forcing something cannot be present in the biomedical ontology, as it is more of a inter–domain concept. If we can construct an ontology, which contains all of the above concepts, the ontological analysis of the sentence will be more complete.

An exemplary ontology, created by merging the top–level Basic Formal Ontology with common biomedical concepts is presented in Figure 3.3 on page 24.

A very interesting attempt at merging DOLCE top–level ontology with WordNet ontology[3] is presented in [9].

[3] In the current text, we however use WordNet purely as a lexical database, rather than an ontology.

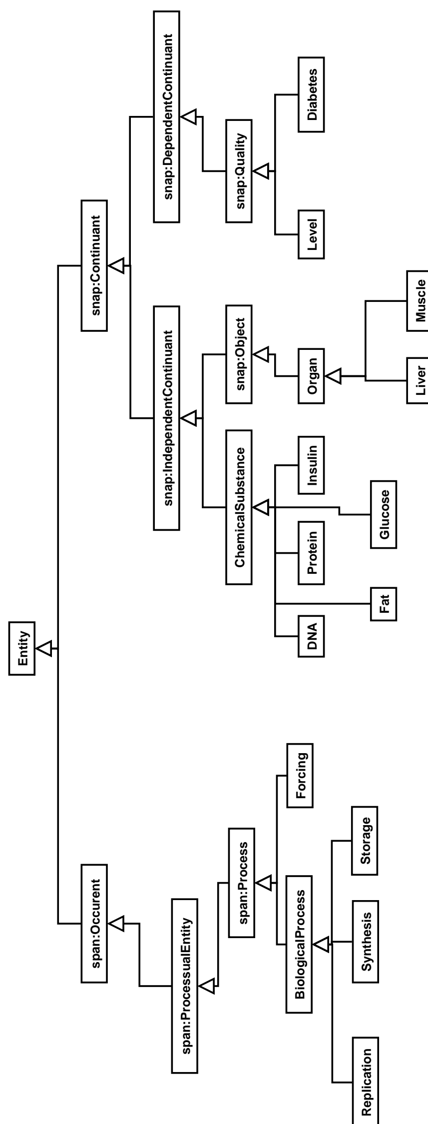



Figure 3.3: An exemplary ontology, created by merging the top–level Basic Formal Ontology with common biomedical concepts.

Chapter 4

# Linguistic analysis

## 4.1 Human language understanding

Human beings are exceptionally good at understanding language. It is by far the most unique way of communicating found in nature, as it is tied to only one species.

Since our goal is to allow computer to understand language, we shall first discuss what makes language understandable for us. Indeed, if we knew how our mind grasps the language, we could try to write a program, which would act in a similar manner.

Unfortunately, this task might be extremely difficult, or maybe even impossible. As a matter of fact we can only understand the brain on two levels:

1. A very low level: we know that impulses are being sent between

cells and we can observe that some regions of the brain are active while performing a particular task, such as speaking.

2. A very high level: we can observe the external behavior produced by our mind.

The first understanding can be compared to how computer hardware functions. The second is similar to the output of a computer program, or it's interface. We do not have any access to the "source code" for our mind.

## 4.2 Language as a protocol

When we want two machines to communicate, we need a protocol, which both will follow, so that they can understand each other. For human beings, the natural language serves as a protocol.

The language, which we use today, and which has been evolving for tens of thousands of years has one particular feature, which distinguishes it from other protocols used today, i.e. it is extremely complicated. This stems from the fact that language evolved to serve purposes of human mind, not that of a computer.

This might be conceived as an advantage in a various contexts, as the ability of describing the same concept in so many different ways allows us to express our emotions, it makes reading novels so pleasurable, etc.

Unfortunately, from the software engineering perspective the complexity of language is a huge problem. No software system is capable of understanding it fully (or even to a large extent).

Luckily, language being a protocol, is not completely random, but rather sentences are built in a specific way. Language has been

studied by linguists and the knowledge about it is extremely useful when one tries to design a software system, which should take natural language sentences as the input.

In Sections 4.2.1 and 4.3 I explain the particularly interesting features of natural language, which might make it computer–understandable to a certain degree.

### 4.2.1 Parts of speech

The English language consists of words, which can be grouped into few classes, called parts of speech or lexical categories. The most common ones are:

- common noun – names object class, e.g. the noun "cell" has the meaning, which covers all imaginable cells, not a particular cell.
- proper noun – names particular object, e.g. "Denmark" is a proper noun referring to a particular country.
- verb – names actions, which are activities or interactions between things, e.g. "stored" is a verb, which names the action of storing something somewhere.
- adjective – names properties ascribed to nouns, e.g. the phrase "hyperglycemic symptoms" ascribes the property of being hyperglycemic to symptoms.
- preposition – explains how different entities are related to each other. Very few prepositions exist. They tend to capture only the most common relations. E.g. the phrase "storage in cell" means that storage takes place inside of a cell.

- adverb – plays similar role as adjective, except that it describes properties of concepts represented by other parts of speech than nouns.

### 4.2.2 Necessity of part–of–speech tagging

Let us consider the following sentence:

insulin forces storage

Let us assume that the underlying ontology understands that the word "insulin" represents the concept c_insulin and the word "storage" represents the concept c_storing. c_insulin is defined as a chemical substance called insulin. c_storing is defined as an action of storing something somewhere. Let's also assume that nothing in our ontology corresponds to the word "forces".

In such a case understanding the whole phrase could be attempted by skipping the word "forces" and analyzing just "insulin storage". Unfortunately, this would result in an incorrect understanding, as the original phrase wasn't really talking about insulin storage. It's not insulin that is in fact stored.

In order to remedy the situation, the algorithm could use part–of–speech tagging. This process assigns a list of lexical categories to each word of the sentence. The coverage of part–of–speech tagging is extensive, due to the existence of large lexical resources, like WordNet. The size of todays lexical resources is far superior to the size of any existing ontology. Almost any word of English language can be assigned a lexical category, but only a small subset can be found in any ontology.

After the application of the part–of–speech tagging, we are faced with the following information:

| sentence | insulin | forces | storage |
|---|---|---|---|
| tagging | noun | verb | noun |
| ontology | c_insulin | ? | c_storing |

At the moment the algorithm understands, that the sentence does not talk about storage of insulin, as words “insulin” and “storage” appear on two sides of a verb. Such a situation disallows the compound concept to be understood as storage of insulin. Rather, insulin could be an agent and storage could be a patient of some action, described by the verb.

Observe that if the middle word is, e.g. a noun, we might be faced with a different situation. For instance: “insulin particles storage” sentence has a different structure, where the second word describes a patient for the action described by third word. In other words, we deal with the concept of storing some particles.

## 4.3 Grammar as the structure of English

### 4.3.1 Rules and structure

All sentences of a language are similar. Even though completely different words are used, the structure of the sentence follows some rules. Recognizing those rules is of utmost importance for understanding the sentence. If the sentence is not constructed according to the rules, it won’t be understood or it will be misunderstood.

With artificial languages (e.g. programming languages), knowing the rules is simple. When language is designed, a set of rules is invented, usually using a variant of Backus–Naur Form if the language is context–free. One might therefore construct all the valid

sentences of the language with the rules provided.

### 4.3.2 Rules for natural languages

The problem with natural languages is that they were not designed with any rules in mind. The language evolved together with human brain, and we somehow know from the very childhood, which sentences are correct and which are not (to some extent). We learn that by listening to exemplary correct sentences, rather than learning some rules and following them.

Hence, we are faced with a problem, which appears only for natural languages. The problem is that we don't know the rules of the language. In a sence the rules are hidden.

Why would we need the rules at all? It's because we would like to teach computers to understand natural language. Unfortunately, inputting all valid sentences into a database is not an option. Such a set would probably be infinite, or in most optimistic estimation extremely huge. So we shall try to construct a relatively small set of rules, which could describe the structure of the language in a satisfactory way.

In order to learn the rules, we might assume that we know which sentences are correct and which are incorrect, and based on that we would like to analyze the language, so that we can recognize the hidden rules.

Such a task, however, turns out to be extremely difficult.

A substantial problem is that whatever set of rules one might come up with, it is always relatively simple to find a counterexample. Such a counterexample could take two forms. It could either be a valid sentence, which cannot be produced using the rules, or it could be

an invalid sentence, which could be produced using the set of rules provided.

For this very reason one cannot prove that English is a context–free language, hence it might not be describable using a context–free grammar.

Many linguistic problems that are faced if one tries to construct a context–free grammar for English are described in [17].

### 4.3.3 Shallow context–free grammar for English

This section describes the idea of a shallow grammar for English language.

The problems described in Section 4.3.2 have to be dealt with somehow if we want the system to understand the language. Fortunately, the system, which is to be designed in the current text, has some important properties. The crucial thing to realize is that the system has to analyze biomedical documents. Such documents are written using correct English language. Therefore, we can assume the absence of incorrect sentences in the analyzed text.

Such assumption has a major influence on the set of rules, which we shall decide upon. Normally the rules should be very restrictive, so that they need to reject incorrect sentences. However, since we won't have any incorrect sentences, we don't need to reject them with the help of our rules.

As an example, consider the following simple rule[1]:

$$\langle \text{sentence} \rangle ::= \langle \text{noun} \rangle \langle \text{verb} \rangle \langle \text{noun} \rangle \tag{4.1}$$

[1] Expressed in BNF.

This rule expresses that a sentence can consist of a noun, followed by a verb, followed by a noun. Let us consider the following two sentences:

1. Insulin forces storage.
2. Insulin force storage.

Sentence 1 is a correct English sentence, while sentence 2 is not. Nevertheless, sentence 1 will be accepted by our rule, as will sentence 2.

If we wanted to reject the second sentence, but accept the first one, we would need to introduce more complicated rules. We can however observe, that the second sentence won't ever appear in the analyzed text, assuming that it's a high–quality material.

Hence, we can use a relatively simple shallow grammar for analyzing English.

As previously noted, English might not be describable by a context–free grammar. However, taking into consideration the aforementioned observation, we can try to formulate a shallow grammar as a context–free grammar, since we don't need to reject incorrect sentences.

Such a grammar would not cover the whole English language, but this is not a problem. We shall note that the text, which is to be analyzed, is a scientific text. As such it normally makes use of a specific subset of English language only.

As an example, we could consider the problem of questions. In a casual conversation, a novel, or a play, questions occur very often. This is however not a case for scientific articles, reports and texts. The Wikipedia entry for "Insulin", which is our playground, does not

contain even one question. That observation allows us to construct a substantially simpler grammar for text analysis, as the order of words in a question is not the same as in a statement.

In case a question appears in the analyzed text, we simply won't be able to analyze such a sentence. Similarly if a sentence formulated in a less obvious way is encountered, which is rejected by the grammar, it won't be analyzed.

Taking all the considerations into account, we might design the a shallow context–free grammar for English, which is presented in Figure 4.3.3 on page 34.

Please observe that the grammar is ambiguous. The same sentence could produce different parse trees.

## 4.4 Lexical resources

### 4.4.1 WordNet

WordNet is a huge and complex lexical resource, containing a lot of information about different words. It's big coverage resulted in using WordNet as a very general ontology, where synsets represent concepts and the hyponym relation represents the class subsumption relation. Nevertheless, in the current project, WordNet is used exclusively for part–of–speech tagging, as described in Section 6.2.10 on page 81.

Figure 4.1: A shallow context-free grammar for English, specified in BNF.

$$
\begin{array}{rcl}
\langle \text{s} \rangle & ::= & \langle \text{d_a_cnp_pp} \rangle \langle \text{vp} \rangle \\
\langle \text{vp} \rangle & ::= & \langle \text{v} \rangle \langle \text{pps} \rangle \\
 & | & \langle \text{v} \rangle \langle \text{d_a_cnp_pp} \rangle \langle \text{pps} \rangle \\
\langle \text{pps} \rangle & ::= & \epsilon \\
 & | & \langle \text{pp} \rangle \langle \text{pps} \rangle \\
\langle \text{pp} \rangle & ::= & \langle \text{p} \rangle \langle \text{d_a_cnp_pp} \rangle \\
\langle \text{d_a_cnp_pp} \rangle & ::= & \langle \text{d_a_cnp} \rangle \langle \text{pps} \rangle \\
\langle \text{d_a_cnp} \rangle & ::= & \langle \text{d} \rangle \langle \text{a_cnp} \rangle \\
 & | & \langle \text{a_cnp} \rangle \\
\langle \text{a_cnp} \rangle & ::= & \langle \text{adj} \rangle \langle \text{a_cnp} \rangle \\
 & | & \langle \text{cnp} \rangle \\
\langle \text{cnp} \rangle & ::= & \langle \text{n} \rangle \langle \text{cnp} \rangle \\
 & | & \langle \text{n} \rangle \\
\langle \text{v} \rangle & ::= & force|forces|cause|causes|\ldots \\
\langle \text{n} \rangle & ::= & insulin|storage|cell|cells|\ldots \\
\langle \text{d} \rangle & ::= & the|this|my|\ldots \\
\langle \text{p} \rangle & ::= & in|with|by|\ldots \\
\langle \text{adj} \rangle & ::= & pancreatic|recombinant|adipose|pineal|\ldots
\end{array}
$$

### 4.4.2 FrameNet

FrameNet [3, 4] is a large linguistic resource, centered around the notion of frame semantics. A succinct description of what a frame is can be found in [24]:

> Semantic frames are schematic representations of situation types (eating, spying, removing, classifying, etc.) together with lists of the kinds of participants, props, and other conceptual roles that are seen as components of such situations. The semantic arguments of a predicating word correspond to what we call the frame elements of the frame associated with that word.

Even though FrameNet could be very helpful in parsing English texts, I have decided not to use it. The reason for that is that FrameNet is a relatively young project, and therefore does not have a very large coverage. It is also cumbersome to merge with a biomedical ontology. Additionally the complications associated with incorporating it into the system would be overwhelming for such a short project. The shallow context–free grammar coupled with an ontology should be enough for the prototypical implementation of the text analyzer.

CHAPTER 5

# Ontological semantics

## 5.1 Peirce–algebraic ontological semantics for English

We are interested in understanding the language. Understanding the language is nothing more than the ability to capture correct semantics from the text.

There are many theories describing formal semantics for English. One example is the Montague semantics, where one focuses on translating various English phrases into quantified formulae in first order predicate logic.

As an example, consider the following article from the Wikipedia

entry on “Insulin”:

$$\text{“Insulin is required for all animal life.”} \tag{5.1}$$

One could capture the semantics of this sentence in the following way:

$$\forall X(life(X) \land animal(X) \rightarrow requires(X, insulin))$$

This semantical representation is not particularly useful for the purpose of creating a search engine. One of the problems is that predicate logic is not decidable. Therefore it would be impossible to decide whether semantics of some sentence in the text subsumes the semantics of the query sentence.

Additionally, such a focus on quantification is not something that the user of the search engine would expect. As a matter of fact the user is mostly concerned with finding articles, which refer to a particular concept. Here the concept could be defined as requiring of insulin by animal life.

Peirce algebra 3.2.4 turns out to contain a very useful operation, called Peirce product, which allows us to express the ontological semantics of the sentence of interest as:

$$requiring \cap agt : insulin \cap pnt : (animal \cap life)$$

Note that Peirce product (:) binds stronger than intersection ($\cap$).

In order to explain this ontological semantics, let us go back to our running example:

$$\text{Insulin forces storage of glycogen.} \tag{5.2}$$

It's semantics can be expressed using Peirce algebra as:

$$forcing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen) \quad (5.3)$$

Recall that Peirce algebra is an algebra of sets and relations between individuals belonging to those sets. In Equation 5.3, the sets, which represent classes in our ontology, are:

- *forcing* – A class (set) comprising all imaginable actions of forcing. E.g. “me forcing my younger brother yesterday to clean the room” is an example of individual belonging to this class.
- *insulin* – This class contains only one individual, being the chemical substance called “insulin”. It's a singleton set.
- *storage* – A class (set) comprising all imaginable actions of storing.
- *glycogen*– This class contains only one individual, being the chemical substance called “glycogen. It's a singleton set.

Similarly, the following ontological relations are present in Equation 5.3:

- *agt* – Agent relation, relating an action to its agent, which is usually initiating the action and carrying it on.
- *pnt* – Patient relation, relating an action to its patient, which is usually influenced by the action, being modified, etc.

The problem of capturing the semantics presented in Equation 5.3 from the exemplary sentence is described in Section 6.2.3.

In order to fully understand the semantics at hand, let us first translate it according to the definition of Peirce product:

$$\begin{aligned} & forcing \\ \cap\ & \{X | \exists Y((X,Y) \in agt \wedge Y \in insulin)\} \\ \cap\ & \{X | \exists Y((X,Y) \in pnt \\ & \qquad \wedge Y \in (storage \cap \{X | \exists Y((X,Y) \in pnt \wedge Y \in glycogen)\}))\} \end{aligned}$$

The resulting formula is quite complex. Let's try to explain part of it:

$$\{X | \exists Y((X,Y) \in agt \wedge Y \in insulin)\} \tag{5.4}$$

Equation 5.4 defines a set of all individuals $X$, which are agent–related to some individual $Y$, such that $Y$ is an instance of *insulin*. The resulting set will be a class, comprising all entities, which have insulin as an agent.

Coming back to the full sentence, if we intersect all imaginable forcing actions with all entities having insulin as an agent, we get a class containing all forcing actions, which have insulin as an agent.

Similarly, we narrow the *forcing* class by intersecting it with all entities, which have storage as a patient. A further, Peirce product restriction is applied to the *storage* in a nested way, as we are only interested in the storage of glycogen.

Hence, by using Peirce algebraic operations we can construct infinitely many compound concepts, using a finite supply of primitive classes and relations. It is worth noticing that such a representation allows us to capture most of the substantial information from the text.

This semantics is chosen as the best alternative for the ontological natural language analyzer, which is described in Section 6.2.

A very useful property of the chosen semantics is that it is variable–free. Therefore it doesn't suffer from the problems of first order predicate logic, like unsatisfiability. It is used in the working implementation of the system, and allows it to work very efficiently.

Towards the very end of my project, my supervisor came across a very interesting article, which utilizes a very similar application of Peirce product. Please refer to [6] for details.

## 5.2 Semantical incompleteness

If we consider sentence 5.2 and its semantics 5.3, we can notice that not all of the information that humans can retrieve from the sentence is captured by the semantics. For instance, the tense of the verb is lost, so the semantics doesn't tell whether the action of forcing happens presently, in the past, etc.

In other words, we have:

$$\begin{aligned} & [\![\text{Insulin forces storage of glycogen.}]\!] \\ = & [\![\text{Insulin forced storage of glycogen.}]\!] \\ = & forcing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen) \end{aligned}$$

We should discuss whether such a behaviour is desired. We have to keep in mind that the semantics presented is supposed to be used by a search engine for ontological querying.

Clearly, the search engine will give one of the sentences as the match for the other, as they have the same semantics. This is a desired behaviour, because the sentences talk about the same concepts, and we would probably like to neglect the tense when we use the search engine.

On the other hand, we need to keep in mind that such a phenomenon cannot be used as the main principle for the search engine's matching. Also sentences with different semantics should be often found as answers to queries. Consider e.g.:

$$\begin{aligned} & [\![\text{Insulin forces storage of glycogen.}]\!] \\ = \quad & forcing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen) \end{aligned}$$

and:

$$\begin{aligned} & [\![\text{Insulin causes storage of glycogen.}]\!] \\ = \quad & causing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen) \end{aligned}$$

Both sentences have different semantics, yet we would like to return one as the match for the other one being the query. We can do it, because in the underlying ontology we have:

$$isa(forcing, causing)$$

Knowing that the action of "forcing" is kind of a "causing", the search engine shall in this case decide to return the sentence in question as a match.

We can conclude by saying that the incompleteness of the captured semantics can in fact improve the end result of the search.

## 5.3 Rephrasing

One sentence might be reformulated in different ways. Usually words can be moved around the sentence, in order to stress something, make the sentence funny, etc. The sentence normally keeps similar meaning.

Rephrasing, however, poses a challenge for the ontological text analyzer. Clearly, we would like to achieve semantics, which is independent of the way the sentence was phrased. This independence comes in most cases for free with the commutativity of the set intersection operation. As an example, consider:

$$
\begin{aligned}
& forcing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen) \\
= \; & forcing \cap pnt : (storage \cap pnt : glycogen) \cap agt : insulin
\end{aligned}
$$

One could imagine that such two meanings are captured from different sentences. Nevertheless, they will be considered equal, as the set intersection ($\cap$) is commutative.

Another type of rephrasing is illustrated by the following example:

$$
\begin{aligned}
& [\![\text{Insulin forces storage of glycogen.}]\!] \\
= \; & [\![\text{Insulin forces glycogen storage.}]\!] \\
= \; & forcing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen)
\end{aligned}
$$

The system is able to construct the same semantics for this example, because it looks for patient of an action both in prepositional phrase and in compond noun phrase. This is the expected behaviour, as both sentences have the same meaning.

Unfortunately, the system being a prototype, is not able to understand all English sentences. Hence some of the rephrasing will not be handled. The situation concerns most notably the passive form of a sentence, which is not recognized by the text analyzing component. It could be added relatively easily, but due to time limitation, it was not implemented.

To conclude, let us mention that in the current implementation, the system is partially handling the problem of rephrasing.

## 5.4 Incorrect sentences

Since in the design of our shallow grammar we have allowed incorrect sentences to belong to the language described by it, we should discuss what result this decision has on the resulting semantics. One may argue that we assumed that the analyzed text will not contain any incorrect sentences. While this assumption is correct, it could happen that sporadically an incorrect sentence is encountered. As a special case, the user of the search engine might enter such an incorrect sentence as the query.

The following example illustrates the problem:

$$\begin{aligned} & [\![\text{Insulin forces storage of glycogen.}]\!] \\ = \; & [\![\text{Insulin force storage of glycogen.}]\!] \\ = \; & forcing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen) \end{aligned}$$

As one can observe, the incorrect sentence is analyzed in the same way as the correct one. This follows from the shallowness of the grammar.

## 5.5 Correcting the author

Even though the following might appear correct at a first glance:

$$\begin{aligned} & [\![\text{Insulin forces storage of glycogen.}]\!] \\ = \; & forcing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen) \end{aligned}$$

we actually have a deep problem which causes the extracted semantics to be in fact incorrect. The problem is that insulin cannot really act as an agent for the action of forcing. When we think of insulin, we realize that it is a chemical substance, represented by some chemical formula. We can talk about some chemical compound even if no

particles of such substance actually exist in our universe. Therefore a mere existence of a chemical formula cannot force anything.

We arrive at a conclusion that it's rather the presence of insulin, which causes some action, not the insulin. So we could attempt to capture the following semantics:

$$\begin{aligned} & [\![\text{Insulin forces storage of glycogen.}]\!] \\ = \quad & forcing \cap agt : (presence \cap agt : insulin) \\ & \cap pnt : (storage \cap pnt : glycogen) \end{aligned}$$

Still, this semantics is flawed. The fact that some chemical substance exists, in the sence that maybe a particle with such a chemical formula was found once, cannot be used as an agent for our forcing action. It's not the presence of a substance, which we are after, but the presence of particles of this substance:

$$\begin{aligned} & [\![\text{Insulin forces storage of glycogen.}]\!] \\ = \quad & forcing \cap agt : (presence \cap agt : (particle \cap wrt : insulin)) \\ & \cap pnt : (storage \cap pnt : glycogen) \end{aligned}$$

The last semantics is finally correct. The problem in recognizing such a correct semantics is that the word representing the concept of presence is not visible in the input sentence. Additionally the same can be said about the concept of a particle.

When humans use language, they create a lot of mental shortcuts, to make sentences shorter. So when the author says "Insulin forces storage of glycogen.", she doesn't probably mean that some chemical formula forces an action, but that the presence of particles with that formula forces some action. The reader is able to decipher the intended meaning, of course.

Unfortunately, similar deciphering, or on–the–fly correction of extracted semantics would be really difficult for the computer to perform. I've decided that trying to implement it would be out of the

scope of my project, and therefore the system extracts a semantics, which might contain the mental shortcuts implanted by the author.

## 5.6 Relations vs. classes

For some sentences we have the choice of extracting semantics in two different ways: using a relation or a class. For our exemplary sentence we could have:

$$\begin{aligned} & [\![\text{Insulin forces storage of glycogen.}]\!] \\ = \; & forcing \cap agt : insulin \cap pnt : (storage \cap pnt : glycogen) \end{aligned}$$

But we could also use the `CBY` relation, which relates an action to its cause. In such a case we would obtain:

$$\begin{aligned} & [\![\text{Insulin forces storage of glycogen.}]\!] \\ = \; & storage \cap pnt : glycogen \cap cby : insulin \end{aligned}$$

Please observe how in one version the notion of forcing is represented by a class and in the other by relation.

It should be decided which of the two semantics is preferable. In my opinion the former is a better choice for the following reasons:

- By representing the action of forcing by the *cby* role we actually loose some useful information. This is because few other verbs could trigger this role, e.g. “causes”.
- Since we did not introduce a subsumption relation between relations, but only between classes, representing forcing as a class allows for some more ontology–based reasoning. E.g. we can easily see that “forcing” is below “causing” in the ontology, which is in turn below “action”, etc.

- One could argue that if we introduce a relation for representing the action of causing, we should be consistent and do the same for other actions as well. In my opinion it is preferable to keep the number of relations low.

- Relations also have the problem of requiring the inverse relations to be taken into consideration. Consider the following two semantics:

  $$storage \cap pnt : glycogen \cap cby : insulin$$
  $$insulin \cap cau : (storage \cap pnt : glycogen)$$

  Both semantics are equivalent, as the relation *cau*[1] is the inverse relation of *cby*[2]. For some reasoning engine, it must be explicitly defined that those relations are inverse, so that it could infer that the two presented semantics are the same.

[1] $r(\phi, cau, \rho)$ means that *phi* causes $\rho$.
[2] $r(\phi, cby, \rho)$ means that *phi* is caused by $\rho$.

Chapter 6

# Implementation

## 6.1 Choice of programming language

I have deceded to use a declarative programming language for the implementation of the system. There are few key features that had most impact on my decision:

- Most declarative languages have a formal or semiformal semantics, which allow for proving the correctness of the program.
- Declarative languages have a syntax, which is very close to the standard mathematical notation. It is therefore easy to express formal ideas as a program.
- Because a declarative program describes *what* the solution is instead of *how* to compute it, the source code for the pro-

gram is around 10 times more concise than one expressed in a imperative language.

- Declarative languages have strong type systems, unlike imperative languages. Consider e.g. the Java imperative language, where `null` is a member of any user–defined type, which is mostly undesirable.

- Some declarative languages have support for nondeterministic computation (returning multiple results), which turned out to be very helpful.

### 6.1.1 StandardML

The following description [13] of StandardML [14] is a particularly concise description of the language:

> Standard ML is a safe, modular, strict, functional, polymorphic programming language with compile–time type checking and type inference, garbage collection, exception handling, immutable data types and updatable references, abstract data types, and parametric modules. It has efficient implementations and a formal definition with a proof of soundness.

For those reasons I have initially chosen StandardML as the implementation language. In addition I know it very well from a few DTU courses, so it was easy for me to write the code.

Unfortunately, I have stumbled upon a very big obstacle along the way. It turned out that some of the operations, which are necessary, would benefit greatly from nondeterministic implementation. Unfortunately, Standard ML does not provide language support for nondeterminism.

Consider, e.g. the problem of recognizing the ontological relations, based on prepositions. Let's say that we would like to have the following exemplary mappings:

| Preposition | Relation | Example |
|---|---|---|
| on | TMP | on Monday |
| to | DST | to school |
| in | TMP | in winter |
| in | LOC | in cells |

As we can see, preposition "in" can describe two different relations, temporal placement and locational placement. Unfortunately, one cannot write in StandardML:

```
1 datatype role
2   = TMP (* temporal aspects (generic role) *)
3   | LOC (* location, position *)
4   | DST (* destination of moving process *)
5
6 fun p2r "on" = TMP
7   | p2r "to" = DST
8   | p2r "in" = TMP
9   | p2r "in" = LOC
```

This program will not compile, because of the way pattern matching works in functional languages. Only the first matching definition is used, so the pattern

```
1   | p2r "in" = LOC
```

is unreachable.

Of course workarounds exist for that problem, e.g. the function p2r instead of having the type `fn : string -> role`, could have type `fn : string -> role list`, so instead of nondeterministically returning a role, it could return deterministically a list of roles:

```
1 datatype role
2   = TMP (* temporal aspects (generic role) *)
3   | LOC (* location, position *)
4   | DST (* destination of moving process *)
5
6 fun p2r "on" = [TMP]
7   | p2r "to" = [DST]
8   | p2r "in" = [TMP,LOC]
```

In this way, one could implement a behaviour that is a substitute for nondeterminism, using exception throwing for backtracking. I have followed that path, however at some point the code became very complicated. When I realized that I'm using lists of lists of lists, I have decided that I need a language with support for nondeterminism, so I have dropped the StandardML as an implementation language.

### 6.1.2 Prolog

Prolog is the most popular programming language supporting nondeterminism. It's based on SLD–resolution and unification with negation as failure and a closed world assumption. Therefore we can simply write:

```
1 p2r(on,tmp).
2 p2r(to,dst).
3 p2r(in,tmp).
4 p2r(in,loc).
```

If we now run the program with the goal ←`p2r(in,X)`, we will get two answers for `X`, i.e. `X=tmp` and `X=loc`.

This is a great feature allowing very easy formulation of the problem at hand. Unfortunately, Prolog does not come with a type system. The programmer needs to verify by hand that the terms used will unify iff they are supposed to. In particular, the heavy use of lists introduces some problems.

I have moved a prototype implementation of the system to Prolog, but once I've made a type mistake (e.g. term had arity of three, not four), Prolog couldn't find it. In such a case the only thing that happens is that Prolog is saying `NO` and not giving any answers, because the unification fails at some stage.

Since this sort of behaviour is not considered erroneous by Prolog, the only way of detecting such a simple typing error is tracing the execution of the program step–by–step until one realizes where the problem lies. I have spent countless hours on such unnecessary debugging, and therefore I've decided that I need to find a language with embedded type system.

### 6.1.3 Mercury

The following description [20] of Mercury is a concise description of the language:

> Mercury is a new logic/functional programming language, which combines the clarity and expressiveness of declarative programming with advanced static analysis and error detection features. Its highly optimized execution algorithm delivers efficiency far in excess of existing logic programming systems, and close to conventional programming systems. Mercury addresses the problems of large-scale program development, allowing modularity, separate compilation, and numerous optimization/-time trade-offs.

#### 6.1.3.1 Type system

The most important feature of Mercury, which has decided of its choice as the language of implementation, is its strong type system. The type system is very similar to the one known from functional languages, like Haskell or ML. It includes discriminating union, recursive and polymorphic types. A type definition for a well–known append predicate has the following form:

```
1 :- pred append(list(T), list(T), list(T)).
```

#### 6.1.3.2 Determinism declaration

Mercury allows for creating the following kinds of predicates and functions:

- Deterministic – predicate can succeed exactly once.
- Semi–deterministic – predicate can succeed at most once.
- Multi – predicate always succeeds more than once.
- Nondeterministic – predicate can succeed any number of times.
- Erroneous – predicate cannot succeed.

In addition there are "commited choice" nondeterministic predicates.

The determinism of a predicate has to be defined by the programmer and is strictly checked by the compiler. The compiler can infer the correct nondeterminism declaration for most predicates.

If the program is not written clearly, the compiler may overestimate the behaviour of a predicate, e.g. it can infer that a predicate is

nondeterministic, while in fact it is semi–deterministic. This can usually be easily fixed by a small re–write of the code. Usually one should use more functional programming style than logic programming style for semi–deterministic and deterministic predicates. For instance, using if–then–else constructs helps a great deal, as in such a case compiler knows that either "then" branch will be taken, or "else", but not both, and not neither.

While the necessity of declaring the determinism of a predicate might seem like a waste of time, it is not. There are certain programming errors that are detected by the compiler in this way.

The knowledge that compiler gains from the determinism declarations allows it to generate very efficient code. Deterministic and semi–deterministic predicates, for instance, do not have a need for backtracking. Mercury programs tend to work around 50–200 times faster than Prolog programs.

#### 6.1.3.3 Mode system

The Mercury language has the notion of instantiatedness of a variable. It corresponds to a specific instantiation of the type constructors, which form a value. Consider the following simple example:

```
1 :- inst non_empty_list
2         == bound([ground|ground]).
```

Here we can see that the value is `bound` to the list constructor (`[|]`) and both head and tail are `ground`. Hence, the list must be non–empty. Two most common instantiatednesses are `ground`, where the term does not contain any variables, and `free`, meaning that variable is not instantiated at all.

Those definitions are in turn used in so–called modes. The mode of a predicate declares what happens with the parameters of a predicate during its execution. Two most common modes are:

```
1 :- mode in == ground >> ground.
2 :- mode out == free >> ground.
```

In other words, a variable is an input to a predicate, if its `ground` both when the predicate is called, and when its finished. A variable is an output of a predicate, if its `free` when the predicate is called, and it's `ground` when its finished.

The mode system is immensely helpful for avoiding common mistakes in logic programming. Consider for instance the well-known list appending predicate, where type, determinism, and mode declarations look as follows:

```
1 :- pred append(list(T), list(T), list(T)).
2 :- mode append(in, in, out) is det.
3 :- mode append(out, out, in) is nondet.
```

One needs to declare that all three arguments are lists of the same type of elements. Additionally, we want to use `append` in two ways. First, given two input lists, we want to concatenate them to deterministically produce a resulting list. Secondly, given a list, we want to nondeterministically produce two lists, which when concatenated, give us the supplied list.

Apart from serving as a nice overview of how the predicate works, such a declaration allows the compiler to check for the following exemplary mistakes:

- Type errors – if the predicate tries to unify one of the arguments with a list of something else than the other arguments.
- Determinism errors – if the mode which we declared to work

deterministically, is in fact nondeterministic.

- Mode errors – If the `out` variable is not instantiated to be ground by the predicate.

#### 6.1.3.4 Pure declarativeness

In most declarative languages there is a need for impure functions or predicates, which introduce side–effects. This is very often the case with e.g. input/output operations. In Mercury input and output is performed with preservation of purely declarative semantics. This can only be achieved thanks to the mode declarations for predicates. Consider, e.g. the predicate for loading WordNet database:

```
1 :-pred load_wn(string::in,
2               wn::out,
3               io::di,
4               io::uo) is det.
```

The two last parameters represent input/output state before and after the predicate was called. They have special modes. `di` means "destructive input" – the variable is dead after being passed to `load_wn`, cannot be used again. `uo` means "unique output" – the variable can be used only once after being output from the predicate. Please observe that the predicate cannot be nondeterministic, as it is impossible to backtrack over input/output operations. One cannot "undisplay text" or "unread from a file". This is different than in Prolog, where input/output is performed in impure way, so one can perform input/output operations in backtracking predicates.

### 6.1.4 Java

I've used Java Server Pages technology for the web interface to the ontological language analyzer daemon. It allows for creation of a nice graphical user interface, which cannot be done in Mercury.

## 6.2 Ontological natural language analyzer

### 6.2.1 Module aux_io

This module contains many useful input/output predicates, which were not found in the standard Mercury library. Of particular interest might be the following predicate:

```
1 :- pred save_concept_graphs(list({concept, list(
      word)})::in,
2                                string::in,
3                                io::di,
4                                io::uo) is det.
```

It is used for creating graphical diagrams for compound concepts. Such a representation is a much more user–friendly representation than the mathematical notation involving Peirce product.

Drawing diagrams is a task, which is best accomplished with the use of GraphViz. GraphViz is an open–source diagram drawing software package. It accepts description of a graph written in the "dot language". The predicate `save_concept_graphs` will convert a semantical sentence description to a graph, and write a file, containing a "dot language" description of the graph. Later, GraphViz software will transform such a description into a PNG image file, containing a drawing representing the graph.

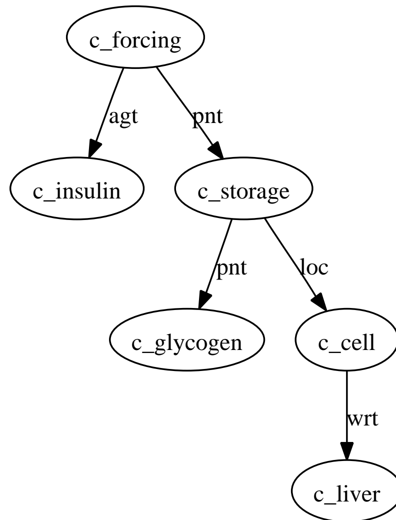
Consider for instance the sentence:

$$\text{Insulin forces storage of glycogen in liver cells.} \tag{6.1}$$

It's ontological semantics is:

$$\begin{aligned} &forcing \cap agt : insulin \\ &\cap pnt : (storage \cap pnt : glycogen \\ &\qquad \cap\, loc : (cell \cap wrt : liver)) \end{aligned}$$

Since the Peirce product is not recognizable by average users, we might represent it by an arrow in the following manner: whenever we encounter a formula of the form $a \cap b : c$, we replace it by $a \xrightarrow{b} c$. We shall repeat such procedure until no more Peirce products are found in our formula. We achieve the following graph:




Such a graph is described by the following "dot language" file:

```
1  digraph concept {
2  label="insulin[n] forces[v] storage[n] of[prep] 
      glycogen[n] in[prep] liver[n] cells[n]"
3  n_1 [label="c_forcing"]
4  n_2 [label="c_insulin"]
5  n_1 -> n_2 [label="agt"]
6  n_3 [label="c_storage"]
7  n_1 -> n_3 [label="pnt"]
8  n_4 [label="c_glycogen"]
9  n_3 -> n_4 [label="pnt"]
10 n_5 [label="c_cell"]
11 n_3 -> n_5 [label="loc"]
12 n_6 [label="c_liver"]
13 n_5 -> n_6 [label="wrt"]
14 }
```

### 6.2.2 Module `createOntology`

Normally an ontology is stored in a database, and read when the program starts, or accessed during the runtime frequently. This is in fact the treatment, which I use for the `obo` module. However, here I have chosen a completely different approach. The ontology is read and turned into a Mercury source code file, so that it can be compiled together with the rest of the system.

This module contains the `main` predicate, so that it becomes a separate executable program. The module creates a Mercury source code for another module, `ontology`. The `ontology` module is generated automatically in the following way:

- The ontology definition file is read, parsed, and analyzed. The definition file is a text file, where each line defines a class.

The first entry on a given line is the name of the class, the remaining entries are the names of all immediate superclasses of the given class, or its parents. The excerpt of the ontology definition file looks as follows:

```
continuant entity
conversion biological_process
entity
excretion biological_process
forcing causing
glucose substance
glycogen substance
insulin substance
```

- The so–called ISA–table is created, which is used later for fast ISA–lookup.
- The preamble of the resulting Mercury source file is written.
- All the ontological concepts are encapsulated in the `simpleConcept` type, and written to the resulting Mercury source file.
- The class subsumption relation is appended to the resulting file. It is represented as a set of facts. Given a class $c$, all superclasses of a given class are found, not just immediate ones. ISA relation is then written down in the following way:

  $$isa(c, a_1).\, isa(c, a_2).\, \ldots isa(c, a_n).$$

  where the set $\{a_1, a_2, \ldots, a_n\}$ represents all the concepts, which are ISA–related to concept $c$. Please note that ISA is reflexive, so $c$ is always member of such a set.

There are a few advantages of compiling the ontology together with the rest of the system:

- The resulting concepts are represented in a very efficient way, e.g. if the C language is set as a target, they will be represented by `enum`s.
- The system does not need to access a database dynamically, which reduces the overhead of using the ontology to the minimum.
- Certain ontology errors will be detected by the compiler, while compiling the automatically generated Mercury source file. E.g. if the class name is misspelled in one place in the ontology definition file, it will be detected as a type error.
- The ISA inferences are performed not when the system is running, but when the `ontology` module is generated. Since the system uses the `isa` predicate very often, doing the inferences only once is a big performance improvement. The resulting factual clauses are compiled by Mercury compiler in a very efficient way. The compiler associates a hash table with the inputs and outputs of factual clauses, so that calling the generated `isa` predicate takes almost a constant time, assuming that few hash collisions occur. In a more traditional setup, one would rather call a backtrackable predicate, which does the ISA inference each time, which would take a time many orders in magnitude longer. There is a small drawback of the chosen representation for ISA relation, i.e. its memory requirement. Remembering just the parents in a relation and then inferring all ancestors requires less memory than remembering all the ancestors. However, we should keep in mind that the biomedical ontology is quite “flat”. We can come from almost each concept to the top concept in just a few steps. Additionally, the taxonomy is almost a tree, so that each ontological concept will typically have 1 parent and from 5 to 7 ancestors. Hence, the memory requirement is practically around 6 times larger than for the slow program.

The speed improvements are quite important, mostly due to the

way that a commercial version of such a system would be used. In a realistic scenario, the system would need to process huge amounts of input data, e.g. all Wikipedia articles in the area of biomedicine.

The only disadvantage is that the source must be regenerated, and the module recompiled, whenever the ontology changes. This is not a big problem, as the ontology is not a database in a classical sence. It is a form of a static knowledge base, so it is not modified dynamically. New versions of the ontology become available few times per month or so, but once the available ontologies mature, they will not be updated as often.

### 6.2.3 Module grabber

This module is the core of the system. It extracts the ontological semantics from the text. Its name comes from the fact that it "grabs" ontological concepts from the text.

#### 6.2.3.1 Types and interface

The following roles are defined in the module:

```
11 :- type role --->
12       tmp % temporal aspects (generic role)
13     ; loc % location, position
14     ; prp % purpose, function
15     ; wrt % with respect to
16     ; chr % characteristic (property ascription)
17     ; cum % cum (i.e., with accompanying)
18     ; bmo % by means of, instrument, via
19     ; cby % caused by
20     ; cau % causes
21     ; cmp % comprising, has part
```

```
22      ; pof % part of
23      ; agt % agent of act or process
24      ; pnt % patient of act or process
25      ; src % source of act or process
26      ; rst % result of act or process
27      ; dst % destination of moving process
28      .
```

Not all of the roles are actually used. The following type is used to represent the Peirce product:

```
30 %Peirce product representation
31 :-type rolePair --->
32             r(role,concept).
```

Whenever we have a formula of the form:

$$rel : c$$

it is represented in Mercury as:

```
r(rel,c)
```

The semantic concepts captured from the text are represented by the following type:

```
34 %A class with a list of Peirce product
       restrictions
35 :-type concept --->
36             c(simpleConcept,            %Name of the
                   concept
37                 list(rolePair)          %Associated
                       attributes
38               ).
```

This type contains a main ontological concept and the list of roles, which are formed by a list Peirce product restrictions. There is an implicit conjunction between the elements of the list. Consider our earlier example:

$$\begin{aligned}&forcing \cap agt : insulin\\ &\cap pnt : (storage \cap pnt : glycogen\\ &\qquad \cap loc : (cell \cap wrt : liver))\end{aligned}$$

It can be represented in Mercury as:

```
c(forcing,[r(agt,c(insulin,[])),
           r(pnt,c(storage,[r(pnt,c(glycogen,[])),
                            r(loc,c(cell,[r(wrt,c(liver,[]))]))
                           ]
                    )
              )
          ]
 )
```

The resulting term is of a `concept` type. Notice that the `rolePair` and `concept` types are mutually recursive.

The module has only one predicate exported in its interface:

```
40 %Wrapper for the grab predicate
41 :-pred grab_main(wn::in,
42                  list(html.pstring)::in,
43                  {concept,list(word)}::out) is
                      nondet.
```

The predicate takes two inputs: the WordNet database and the sentence to be analyzed, represented as a list of words. The ontology does not need to be passed as an argument, because unlike WordNet it is compiled into the program in the form of `ontology` module.

Each nondeterministic result is returned as a pair, comprising the concept found and the list of words which were used to capture the concept. The list of words is returned, because not always the concept grabbed corresponds to the whole sentence. In case some part of the sentence cannot be understood, the concept might correspond only to the part of the input sentence. The list returned in the result indicates which part of the sentence was actually understood.

Internally, the module uses another predicate for analyzing sentences:

```
74 %%This predicate performs the ontological
      sentence analysis
75 :-pred grab(simpleConcept::in,
76              syntacticHinting::in,
77              list(word)::in,
78              concept::out) is nondet.
```

Since this predicate is of great importance, let us have a look at its arguments:

1. The first input argument of type `simpleConcept` tells the predicate what it should look for. In some cases we don't care what should be found. In such a case it should be set to `c_entity`, which is the top concept of the ontology, because every concept is also a top concept. However, in certain cases, we would prefer to look for something else. As an example imagine, that a concept of storage has been found and we are now analyzing a phrase, which determines the patient of the action of storage. Since we may have the additional information that only substances can actually be stored, we might analyze the given phrase searching only for a substance. In this case this argument shall be set to `c_substance`.

2. The second input argument of type `syntacticHinting` is used as a hint for the grabber containing the information at what

nonterminal symbol of the shallow context–free grammar for English we actually are. This is necessary due to the fact that the sentence is not parsed beforehand. Instead the ontological analysis is performed on the non–parsed sentence with the help of the hinting mechanism provided by this argument.

3. The third input argument of type `list(word)` represents the list of words, which should be analyzed. At the top level the list represents the whole sentence, but it will correspond to shorter phrases when the predicate is called recursively during the analysis.

4. The only output argument of type `concept` represents the concept, which has been captured from the supplied text.

#### 6.2.3.2 Syntactic hinting

Perhaps the syntactic hinting mechanism requires more insight. In the initial version of the system, the sentence was actually parsed according to the shallow grammar described in Section 4.3.3 on page 31. The parsing returned nondeterministically multiple parse trees, which were used to capture ontological semantics from the text. Due to the ambiguity of the shallow grammar, the number of such parse trees could be large (reaching around 100 for longer sentences). At the same time almost all of such parse trees did not produce a valid ontological semantics. This is due to the fact that there are certain limitations to how the concepts in the ontology can be combined together.

For that reason parsing the sentence and analyzing all the parse trees is not a very efficient approach. It is much more efficient to incorporate the parsing into the ontological analysis. In such a case the possible outcome is being constricted by two factors simultaneously: the shallow context–free grammar for English and the ontological restrictions. In such a solution the system performs much faster, as

all the useless parse trees do not need to be examined.

As an example, consider the following two calls of the `grab` predicate:

```
grab(c_entity,s,...).
```

Such a call means that we are looking for arbitrary entity, and the list of words passed as third argument represents a whole sentence. On the other hand somewhere during the recursive execution we might encounter the following call:

```
grab(c_substance,d_a_cnp_pp,...).
```

This means that we are only looking for some chemical substance[1], and the list of words passed in the third argument corresponds to the `d_a_cnp_pp` nonterminal symbol of the shallow grammar for English. In other words some chemical substance is expected to be described by a phrase consisting of a possible determiner, possible list of adjectives, a possibly compound noun phrase and a possible prepositional phrase, in that order. Any violation of the expected syntax will result in a failure. But also any violation of ontological expectations will result in a failure, e.g. if the "main" noun of the phrase is not placed under `c_substance` in the ontology.

Hence, a lot of fruitless searching is eliminated, making the system perform much faster.

#### 6.2.3.3 Part–of–speech tagging

Another improvement concerns the way that the part–of–speech tagging is performed. The sentence is tagged before being analyzed,

[1] Perhaps as a patient for previously recognized action of storage.

what is visible in the following definition:

```
50 grab_main(WN,S,{C,TS}):-
51         lexicon.tag(WN,S,tagged(TS)),
52         grab(c_entity,s,TS,C).
```

This approach is intended for improving the speed of the system. Words could be also tagged on the fly, but that would unnecessarily slow down the system. It is worth noticing that the same word may be visited several times during the semantic analysis process. This comes from the fact that our shallow grammar is ambiguous, prepositions correspond to many ontological relations, and words can have different ontological meanings. Hence tagging the word once beforehand and remembering the tag reduces the need of repeatedly checking the WordNet database for corresponding tags.

#### 6.2.3.4 Recognizing relations

The following predicate aids in translating prepositions to roles:

```
64 %%Translate preposition to role
65 :-pred p2r(string::in,role::out) is nondet.
```

This predicate is defined using factual clauses of the form:

```
70 p2r("in",loc).
71 p2r("of",pnt).
72 p2r("of",wrt).
```

While some relations are recognized based on the prepositions present [12], it is also necessary to make use of our shallow grammar for English for that purpose. It is often necessary to infer the relation based on the position of the word in the sentence. This is done, e.g. for the `agt` relation, which is triggered if we have a noun phrase followed

by a verb phrase. In such a case the concept described by the noun phrase is analyzed as a possible agent for the action described by the verb phrase. Similar situation arises with the `pnt`, `wrt` and `chr` relations.

#### 6.2.3.5 Ontological restrictions

Consider the following sentence:

Insulin forces storage of conversion in liver cells. (6.2)

Clearly, the sentence is flawed. It is impossible to understand the sentence, because conversion cannot really be stored. We usually expect some chemical substance to be the patient of the action of storing. The ontological analyzer makes use of the following predicates to impose such restrictions:

```
179 %%Valid role definitions
180 :- pred validRole_d(simpleConcept::in,
181                      role::in,
182                      simpleConcept::out) is
                            semidet.
```

```
192 %% Inferred valid role
193 :- pred validRole(simpleConcept::in,
194                    role::in,
195                    simpleConcept::out) is nondet.
```

We can for example define:

```
186 validRole_d(c_biological_process,pnt,c_substance
        ).
```

By such a definition we would like to ensure that all biological processes operate on chemical substances. Since `storage` action is in the ontology under `biological_process`, we express that only chemical substances can be stored. Other biological processes, like excretion, should also have chemical substances as their patients. The compound concepts, which do not adhere to the restrictions defined, will be rejected, according to the closed world assumption.

### 6.2.4 Module `html`

This module's main function is to analyze documents in HTML and/or XML syntax, and extract the sentences out of them in Mercury–readable way. Each extracted word is stored in the following datatype:

```
13 %%Positioned string. Contains the begin index
       and end
14 %%index of a word, in STL convention.
15 :-type pstring --->
16           pstring(int,int,string).
```

Remembering just the words is not enough, we need the position as well. It is used later on if we want to modify the HTML document, e.g. highlight some words. This module allows direct reading of HTML/XML files, which is necessary for analysis of vast amount of resources found on the Internet, including Wikipedia entries.

Reading the sentences from HTML/XML file is accomplished by the following predicate:

```
18 :-pred read_sentences(
19           list(list(pstring))::out,
20           io::di,
21           io::uo) is det.
```

Another useful predicate is:

```
23 :-pred tag_and_write_sentences(
24          wn::in,
25          obo::in,
26          list(list(pstring))::in,
27          io::di,
28          io::uo) is det.
```

This predicate is used for "highlighting" the coverage of WordNet and OBO for a given HTML file. All words in all sentences are tagged using WordNet, and printed with part–of–speech tags. The words, which represent ontological concepts according to OBO, will be marked using boldface font.

### 6.2.5 Module `main`

This module is the entry point of the program. After reading the input data necessary for system operation, the program enters the `main_loop`:

```
66 %%Main program loop
67 :-pred main_loop(wn::in,
68                  io::di, io::uo) is det.
69
70 main_loop(WN,!IO):-
71         pipe.read_command_from_pipe(Command,!IO)
           ,
72         interpret(WN,Command,!IO),
73         main_loop(WN,!IO).
```

The program repeatedly:

- Reads a command from pipe, as described in Section 6.2.9 on

page 78. The command is normally sent by the Java Server Pages graphical user web interface.

- Interprets the command. At this point the ontological semantics will be extracted from the query supplied by the user. The system will also generate all necessary graphs that need to be shown to the user as the response to the query. A response will be sent to the GUI website by means of the pipe.
- Calls the `main_loop` predicate recursively. Observe that the predicate is tail–recursive, so that the compiler can generate a code as efficient as `for` or `while` loop in imperative languages.

### 6.2.6 Module `lexicon`

The lexicon module is the bridge between a particular language and the rest of the system. It provides predicates for language–to–ontology translation and part–of–speech tagging.

The important datatype defined in this module, which represents the part–of–speech classes used throughout the system is presented below:

```
11 :- type part_of_speech --->
12         adj;                   %adjective (nice
               , fast)
13         adv;                   %adverb (nicely,
               quickly)
14         cnj;                   %conjunction (
               and, or)
15         d;                     %d (the)
16         n;                     %n (bike, tree)
17         prep;                  %prep (in, on)
18         prn;                   %prn (it)
19         v.                     %verb (increase,
               excrete)
```

Basically, if we have a word, the lexicon may contain two pieces of information associated with it. One is the part–of–speech information, another is the ontological concept represented by this word. Such a tripple is represented by the following datatype:

```
21 :-type word ---> word(html.pstring,
22                        lexicon.part_of_speech,
23                        maybe(simpleConcept)).
```

The following predicate may be used to access the lexical information from outside of the `lexicon` module:

```
27 :-pred getLexicalInfo(wn::in,
28                       html.pstring::in,
29                       word::out) is nondet.
```

It will tag the word and try to assign an ontological concept corresponding to the word.

The `lexicon` module accesses the `wn` module, in order to make use of the WordNet database. This is achieved with help of the predicate:

```
45 :-pred ask_wn(wn::in,
46              string::in,         %Input word, e.g
                   . catched
47              part_of_speech::out,
48              string::out)        %Base form, e.g.
                  catch
49 is nondet.
```

Since WordNet contains only words in base form, a given word might not be found in WordNet. Hence the predicate in addition to looking up the word in unchanged form, performs some suffix manipulation, in the following manner:

- If the word ends with "ies" suffix, it is replaced with "y". It is useful for verbs and nouns, like: "therapies" $\mapsto$ "therapy", "cries" $\mapsto$ "cry".
- If the word ends with "s" suffix, it is simply removed. It is useful for verbs and nouns, like: "creates" $\mapsto$ "create", "articles" $\mapsto$ "article".
- If the word ends with "ied" suffix, it is replaced with "y". It is useful for verbs, like: "supplied" $\mapsto$ "supply".
- If the word ends with "d" suffix, it is simply removed. It is useful for verbs, like: "released" $\mapsto$ "release".
- If the word ends with "ed" suffix, it is simply removed. It is useful for verbs, like: "worked" $\mapsto$ "work".

For the moment, only the English language is supported. Nevertheless it is actually possible to add more languages by adding lexicons for them. It is possible, because the underlying ontology is not language–dependent. In other words, concepts present in sentences of different languages could be the same. One problem would be that different languages might have different grammars. Therefore we shall also introduce a new grammar if a new language should be supported, unless the shallow grammar for that language intersects to a large extent with the shallow grammar for English.

### 6.2.7 Module `obo`

This module is designed for integrating Open Biomedical Ontology resources into the system. OBO is a large biomedical ontology, which can be incorporated into large ontology–based systems. The following types are of crucial importance for the `obo` module:

```
14 :- type obo_class_name == string.
15 :- type obo_class_label == string.
```

```
16 :- type obo_class == int.
```

As we can see, both the class name and label are represented as strings. However, the class itself is represented as a machine integer. This is because the system can perform much better if the common operations are executed using integers. Comparing two integers takes one CPU cycle, while comparing two strings requires a loop, which takes multiple CPU cycles. Hence, the operations, which are performed very often, like checking class subsumption relation, do not take too long time.

A OBO database, when loaded to memory, is represented by the following datatype:

```
17 :- type obo --->
18         obo(map(obo_class_name,
19                 obo_class),        %'GO:0153' -> 34
20             map(obo_class_label,
21                 obo_class),        %"insulin" -> 21
22             map(obo_class,
23                 set(obo_class)), %subclass
24              int).                 %Least unused ID
                 .
```

Basically, it consists of four components:

- A map, which assigns a class to each class name found in the OBO files. This is actually a one–to–one relation, as a class name uniquely determines the class. It's provided for performance reasons mainly, as explained earlier.
- A map, which maps class labels to classes. Class labels represent many–to–one correspondence in OBO. Any class can have one or more labels, which usually correspond to all synonyms describing the class. In a more comprehensive ontology, such

a relation would actually be many–to–many. Consider, for instance the word “force”, which could be a label for a class representing the action of forcing and for a class representing a physical notion of a force. Nevertheless, OBO is a biomedical ontology and I could not find any label that would describe two classes, and therefore I have decided to treat it as a map.

- A map, which stores the class subsumption relation, so–called ISA. It maps each class to a set of its immediate superclasses, or parents. If a class is at the top of the ontology, it will map to an empty set.
- An integer, which represents a count of all the classes present in the whole datastructure. It is very useful during the creation of the datastructure from file. Whenever a new class is encountered, we don't need to check what is the smallest unused integer value representing it. Instead, this value will be used and incremented. Hence, the first class read from the file will be represented by 0, the second one by 1, etc.

The following predicate is used for reading the OBO database:

```
26 :- pred load_obo(string::in, %directory
27                  obo::out,
28                  io::di,
29                  io::uo) is det.
```

Only the directory name shall be provided, where the OBO files reside. The files are parsed and useful data is extracted into the `obo` datastructure.

### 6.2.8 Module `ontology`

This module is automatically generated by module `createOntology`, described in Section 6.2.2 on page 60. The excerpt from the `simpleConcept`

datatype definition presented below:

```
9  :-type simpleConcept --->
10         c_process;
11         c_biological_process;
12         c_animate_process;
13         c_temporal_interval;
14         c_occurrent;
15         c_blood;
```

The ISA relation is defined by means of the following predicate:

```
5 :-pred isa(simpleConcept,simpleConcept).
6 :-mode isa(in,in) is semidet.
7 :-mode isa(in,out) is nondet.
```

The predicate consists of many factual clauses of the form:

```
47 isa(c_liver,c_continuant). isa(c_liver,c_entity)
      . isa(c_liver,c_liver).
48 isa(c_storage,c_biological_process). isa(
      c_storage,c_entity). isa(c_storage,c_occurrent
      ). isa(c_storage,c_process). isa(c_storage,
      c_storage).
```

### 6.2.9 Module pipe

This module is the bridge between the Java Server Pages graphical user interface and Mercury daemon. Its name comes from the way that both systems communicate, i.e. by means of named Unix pipes.

A pipe is a interprocess communication facility provided by the Unix/Linux operating system. Once a pipe is created, processes may use it in order to communicate easily. The pipe is seen by the

processes as a regular file. There are two types of pipes: unidirectional and bidirectional. Since the Linux kernel implements only the unidirectional pipes, I have used those.

The Mercury system works as a daemon – after it starts, it works forever, awaiting a command from the pipe. When the command arrives, it is read, parsed, interpreted, and the response is sent via another pipe.

I have decided to use pipes, because they have a tremendous advantage – they allow two processes to communicate without any language–level support for inter–language communication. Normally if two programs written in different languages want to communicate, they either have to use a specific protocol, like RMI[2], or communicate using network sockets.

Since Mercury is a very young language, it does not come with socket input/output operations. Also using some remote protocol, e.g. SOAP or XMLRPC would be very difficult, as I would need to implement them myself. Fortunately, Mercury supports file input/output, so that it can use pipes, which are implemented in the operating system.

Pipes are relatively primitive communication facility. On one hand programs written in arbitrary languages can communicate through them, but on the other hand, there is no protocol defined, so that types must be converted by the programmer appropriately.

I have decided to use a very helpful facility of the Mercury language. The `read` predicate of the standard library `io` module is able to read input, which is formatted in Mercury syntax and dynamically instantiate a variable to a ground term represented by the input. Using this facility, the Java Server Pages front–end can simply write a valid Mercury term to the pipe, and Mercury will understand

[2]Remote Method Invokation – a protocol used widely for Java interprocess communication, both local and remote.

it. This way I have eliminated the need of creating a protocol for information exchange and implementing a parser for it.

The protocol and syntax is therefore simply described by the following Mercury type:

```
12 %% Commands that the server could send:
13 :- type command --->
14         disambiguate(list(html.pstring), %
               sentence
15                     string).   %directory to
                        save graphs
```

Currently only one command is necessary, but it's very easy to extend the system with additional commands. The command itself is read by means of the predicate:

```
17 :- pred read_command_from_pipe(command::out,
18                                io::di,
19                                io::uo) is det.
```

The query sentence to be analyzed is later analyzed by the ontological natural language analyzer, and the result is presented in the form of semantical graphs[3]. Since Java does not support nondeterminism, all the results are gathered into one list and returned via pipe using the predicate:

```
21 :- pred save_ok_to_pipe(list(concept)::in,
22                         io::di,
23                         io::uo) is det.
```

In case the query contains a word, which cannot be tagged using WordNet, an error response is sent via a pipe, using the predicate:

[3]Which are generated by the aux_io module, described in Section 6.2.1 on page 58.

```
25 :-pred save_tag_error_to_pipe(pstring::in,
26                               io::di,
27                               io::uo) is det.
```

### 6.2.10 Module wn

This module is responsible for integration with WordNet lexical database. The main type defined is:

```
11 :-type wn --->
12         wn(map(string,set(lexicon.part_of_speech
              ))).
```

It is constructed as a map, which assigns a set of part–of–speech classes to each string defined in WordNet. Clearly, the same string can have more than one part–of–speech class assigned. Consider, for instance the word “force”, which is defined in WordNet as both a verb and a noun.

Since WordNet provides only four lexical classes[4], it was necessary to manually create corresponding files for the remaining classes of interest, e.g. prepositions.

The WordNet database is loaded from files[5] using the following predicate:

```
14 :-pred load_wn(string::in,
15                wn::out,
16                io::di,
17                io::uo) is det.
```

[4] Nouns, verbs, adjectives and adverbs.

[5] For the description of the WordNet file formats, please see [22].

After WordNet has been loaded to memory, one can access the stored information using the predicate:

```
33 :- pred get_wn_info(wn::in,
34                     string::in,
35                     lexicon.part_of_speech::out)
                          is nondet.
```

This predicate for a given string, returns nondeterministically all lexical categories to which the string belongs.

## 6.3 Web interface

The overall architecture of the system is presented in Figure 6.1 on page 83.

The system is built using almost exclusively free[6] software, except for the Java Development Environment, which is still proprietary[7]. Therefore it should be relatively easy to replicate the whole architecture on any GNU/Linux or Unix machine.

The web interface consists of Java Server Pages, deployed in Tomcat JSP container. The user is presented with a simple HTML form, which allows her to enter a query in form of a sentence. The query is sent to the Mercury system using unidirectional Unix pipes. The Mercury system performs the ontological analysis of the sentence supplied and for each semantics extracted it generates a graph in the "dot language", as described in Section 6.2.1 on page 58. Java Server Pages upon receiving the output through the return pipe, executes the GraphViz software in order to transform graph descriptions into

[6] As in "free speech", not "free beer".

[7] However, Sun Microsystems has announced that it is doing everything in its power in order to make all its Java packages free.

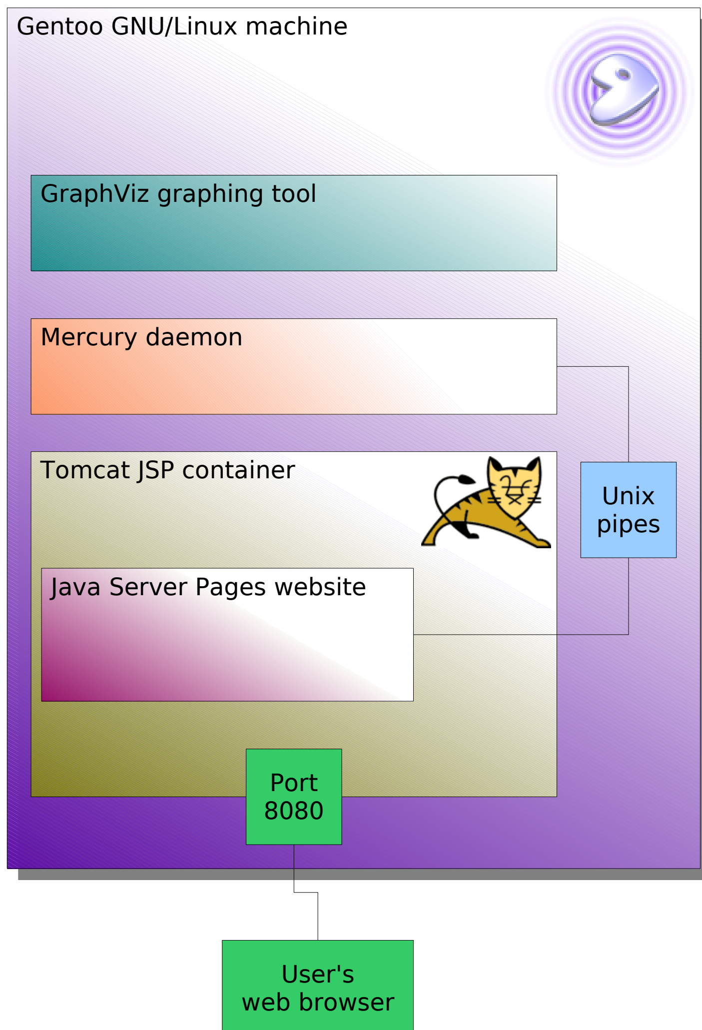



Figure 6.1: The overall architecture of the system.

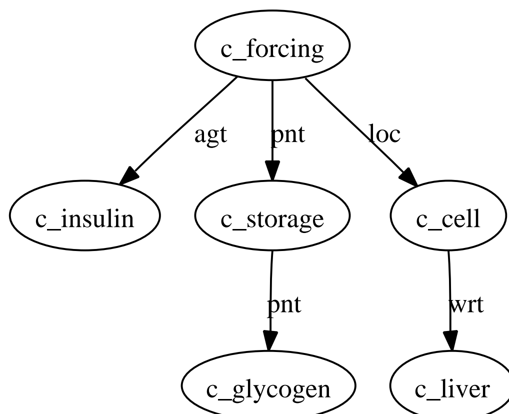



Figure 6.2: First semantics captured from the exemplary sentence.

Portable Network Graphics pictures, which are then embedded in a HTML document and presented to the user as the result of the execution.

## 6.4 Exemplary run of the system

For the exemplary input:

$$\text{Insulin forces storage of glycogen in liver cells.} \tag{6.3}$$

we get three graphs, which are presented in Figures 6.2, 6.3 and 6.4 on pages 84, 85 and 86, respectively.

Notice that the system produces three different ontological semantics for the given sentence, as the sentence itself is ambiguous. The ambiguity arises from the way our shallow grammar has been constructed. Different prepositional phrases can be attached to different noun phrases.

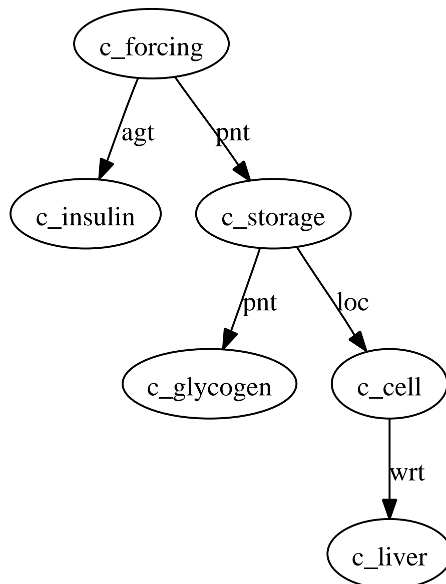



Figure 6.3: Socond semantics captured from the exemplary sentence.

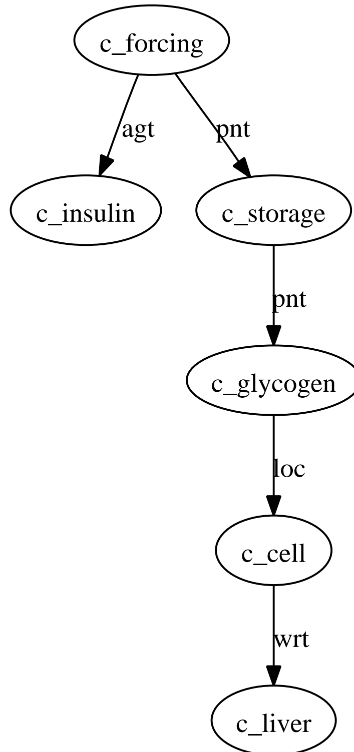



Figure 6.4: Third semantics captured from the exemplary sentence.

As a matter of fact the sentence is also ambiguous from the ontological perspective, as the preposition "in" can describe not only the `LOC` role, but also `TMP`. The meaning which would arise from using the latter one is however ruled out by the application of the rules of validity, described in Section 6.2.3.5 on page 70.

If we analyze the three meanings produced by the system, we observe that:

- In the first meaning, presented in Figure 6.2, we deal with the action of forcing. The agent of the action is insulin. The patient, or the thing being forced, is the action of storage. The patient of the storage action, or the thing being stored, is glycogen. The action of forcing takes place in a liver cell.

- In the second meaning, presented in Figure 6.3, everything remains as in the first meaning, except that now it is not the action of forcing, which takes place in the liver cell, but the action of storage is located there.

- In the third meaning, presented in Figure 6.4, neither forcing nor storage takes place in the liver cell, but rather the glycogen, which is being stored, is located in liver cell.

If we have a look at the sentence and at all three produced meanings, we can conclude that the author of the sentence probably had the second meaning in mind. It is also possible that the author had the first meaning in mind when writing the sentence, but that's less probable. The least probable meaning is the third one. If we wanted to turn the third meaning into English, we would probably structure the sentence in a slightly different way, but it is extremely difficult to teach computer subtle differences like that.

## 6.5 Further extensions

### 6.5.1 Ontological search engine

The semantics extracted from the text could be used in a search engine. As a matter of fact, the design was guided with such a possibility in mind. The semantics has been tailored, so that the search can be performed efficiently and easily.

Consider the following semantics extracted from the text:

$$\begin{aligned}&forcing \cap agt : insulin\\ &\cap pnt : (storage \cap pnt : glycogen\\ &\qquad \cap\, loc : (cell \cap wrt : liver))\end{aligned}$$

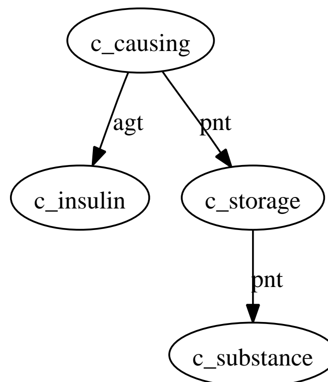
Semantics of this form could be pre–extracted from multiple articles and stored in a database, which could be accessed by the search engine. If the user issues a query, we would simply perform the semantical analysis on it, and extract the query semantics. Let say that the user is interested in the fact that insulin causes the storage of some substance and issues the following query:

$$\text{Insulin caused storage of substance.} \tag{6.4}$$

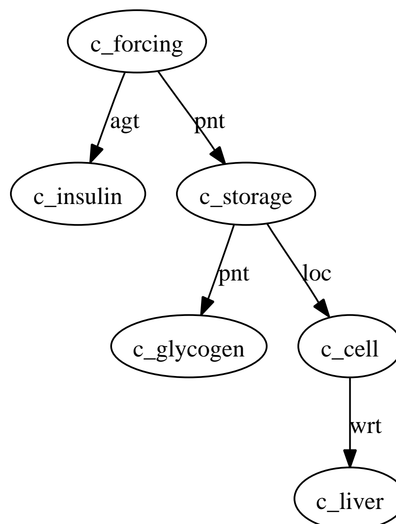
This query would get the following semantics captured:

$$\begin{aligned}&causing \cap agt : insulin\\ &\cap pnt : (storage \cap pnt : substance)\end{aligned}$$

How would the search engine identify the sentence of interest as the answer to the query? The simplest procedure can use a graph matching algorithm, where the graphs are constructed in the way described in Section 6.2.1 on page 58. In our case, we would need to match the query graph:




against the article sentence graph:




Please observe that the underlying ontology with its class subsumption plays a vital role in such a matching, as we need to identify that `forcing` is a subclass of `causing` and `glycogen` is a subclass of `substance` for the graphs to match.

The graph matching has the advantage of respecting the commutativity of set intersection. Additionally it can very easily deal with inverse relations by simply honoring the direction of the arc in the graph.

The need of comparing the query graph to all graphs in the database can be easily eliminated by indexing the database graphs by the primitive classes appearing in them, which are conceived in the current context as graph nodes. The query semantics would only need to be matched against those graphs, which contain all the concepts appearing in the query according to the index.

Unfortunately, the time constraints did not allow me to implement the search engine. Even though the core of the engine would be relatively easy to implement, the amount of work would be substantial due to the need of downloading many articles, setting up a database for them, indexing them, etc.

### 6.5.2 Ad–hoc concepts

A keyword–based search has a certain advantage over an ontological semantic search. If a given word does not correspond to any class in the ontology, the keyword search will still be able to perform some search involving such a word. It is possible to add such a functionality to the ontological search engine quite easily.

This can be achieved by introducing ad–hoc concepts. If a word does not correspond to any class in the ontology, we can generate a new class on the fly. Such a class would be identified by the given word. Obviously, we would not know where this class should be placed in the ontology. We may simply put it under the top concept. The ad–hoc class created in such a way allows the ontological search engine to work as if it was a keyword, i.e. only exact match will be considered as a response to the query. While this approach

is relatively primitive, it allows at least some search results to be returned in case that the ontology does not contain what was queried about.

Chapter 7

# Conclusions

The project has given me a great opportunity for getting acquainted with the area of ontological engineering through the lecture of many interesting books and articles. I believe I have managed to select some of the most interesting, yet simple, ideas from the worlds of semantical natural language processing and ontological engineering. Those ideas turned out to be particularly suited for a declarative, nondeterministic implementation of semantical language analyzer in Mercury. I have also had a pleasure of learning this magnificent programming language, which turns out to be very efficient and safe, yet remaining fully–declarative. The implementation, even though intended to be a prototype, has been optimized in many respects in order to perform very efficiently.

The variable–free semantics used is particularly useful for utilization in a search engine. Semantical search based on the techniques described in this text can improve the user experience dramatically compared with a keyword–based search, because the search engine can understand the text to a certain extent.

I regret that I have applied for five–month project only. With one additional month the time balance between research and programming would be more equilibrial, what would allow me to implement the search engine as well. Nevertheless, I find the project to be interesting to such an extent, where I am willing to devote a month of my personal time for implementing the search engine after the project has been finished.

Appendix A

# Source code and auxiliary files

## A.1 File ontologies.txt

This file contains a list of links to many biomedical ontologies found in OBO collection. It was created for easy ontology downloading procedure with the command "`wget -i ontologies.txt`".

```
1 http://www.berkeleybop.org/ontologies/obo-all/
    arabidopsis_development/
    arabidopsis_development.pro
2 http://www.berkeleybop.org/ontologies/obo-all/
    biological_process/biological_process.pro
3 http://www.berkeleybop.org/ontologies/obo-all/
    brenda/brenda.pro
4 http://www.berkeleybop.org/ontologies/obo-all/
    caro/caro.pro
5 http://www.berkeleybop.org/ontologies/obo-all/
    cell/cell.pro
```

```
6  http://www.berkeleybop.org/ontologies/obo-all/
       cellular_component/cellular_component.pro
7  http://www.berkeleybop.org/ontologies/obo-all/
       chebi/chebi.pro
8  http://www.berkeleybop.org/ontologies/obo-all/
       disease_ontology/disease_ontology.pro
9  http://www.berkeleybop.org/ontologies/obo-all/
       event/event.pro
10 http://www.berkeleybop.org/ontologies/obo-all/
       evidence_code/evidence_code.pro
11 http://www.berkeleybop.org/ontologies/obo-all/
       evoc/evoc.pro
12 http://www.berkeleybop.org/ontologies/obo-all/
       fix/fix.pro
13 http://www.berkeleybop.org/ontologies/obo-all/
       fma_lite/fma_lite.pro
14 http://www.berkeleybop.org/ontologies/obo-all/
       human-dev-anat-abstract/human-dev-anat-
       abstract.pro
15 http://www.berkeleybop.org/ontologies/obo-all/
       human-dev-anat-staged/human-dev-anat-staged.
       pro
16 http://www.berkeleybop.org/ontologies/obo-all/
       image/image.pro
17 http://www.berkeleybop.org/ontologies/obo-all/
       mammalian_phenotype/mammalian_phenotype.pro
18 http://www.berkeleybop.org/ontologies/obo-all/
       mao/mao.pro
19 http://www.berkeleybop.org/ontologies/obo-all/
       mesh/mesh.pro
20 http://www.berkeleybop.org/ontologies/obo-all/
       mged/mged.pro
21 http://www.berkeleybop.org/ontologies/obo-all/
       molecular_function/molecular_function.pro
22 http://www.berkeleybop.org/ontologies/obo-all/
       molecule_role/molecule_role.pro
```

```
23 http://www.berkeleybop.org/ontologies/obo-all/
       ncbi_taxonomy/ncbi_taxonomy.pro
24 http://www.berkeleybop.org/ontologies/obo-all/
       obi/obi.pro
25 http://www.berkeleybop.org/ontologies/obo-all/
       pathway/pathway.pro
26 http://www.berkeleybop.org/ontologies/obo-all/
       plant_environment/plant_environment.pro
27 http://www.berkeleybop.org/ontologies/obo-all/
       plant_trait/plant_trait.pro
28 http://www.berkeleybop.org/ontologies/obo-all/
       psi-mi/psi-mi.pro
29 http://www.berkeleybop.org/ontologies/obo-all/
       psi-mod/psi-mod.pro
30 http://www.berkeleybop.org/ontologies/obo-all/
       quality/quality.pro
31 http://www.berkeleybop.org/ontologies/obo-all/
       rex/rex.pro
32 http://www.berkeleybop.org/ontologies/obo-all/
       sep/sep.pro
33 http://www.berkeleybop.org/ontologies/obo-all/
       sequence/sequence.pro
34 http://www.berkeleybop.org/ontologies/obo-all/
       systems_biology/systems_biology.pro
```

## A.2 File index.jsp

The Java Server Pages website consisting the query input box.

```
1 <HTML>
2 <BODY>
3 <FORM METHOD=POST ACTION="disambiguate.jsp">
4 Enter your query: <INPUT TYPE=TEXT NAME=query
      SIZE=50>
```

```
5 <P><INPUT TYPE=SUBMIT>
6 </FORM>
7 </BODY>
8 </HTML>
```

## A.3 File disambiguate.jsp

The Java Server Pages website, which is run after the query has been submitted.

```
1  <%!
2    public static String stringsToString(String[]
       strings){
3      StringBuffer b = new StringBuffer();
4      for(int i=0;i<strings.length;i++){
5        if(i>0)
6          b.append(' ');
7        b.append(strings[i]);
8      }
9      return b.toString();
10   }
11
12   public static synchronized void toMercury(
       String message)
13     throws Exception{
14     java.io.FileWriter writer
15      = new java.io.FileWriter("/tmp/
          serverToMercury");
16     writer.write(message);
17     writer.close();
18   }
19
20   public static synchronized String[]
       fromMercury() throws Exception{
```

```
21      final java.io.BufferedReader r
22     = new java.io.BufferedReader(new java.io.
          FileReader("/tmp/mercuryToServer"));
23
24      final java.util.Vector<String> lines
25       = new java.util.Vector<String>();
26
27      String line;
28
29      while((line=r.readLine())!=null){
30        lines.add(line);
31      }
32
33      r.close();
34
35      String[] result = new String[0];
36      return lines.toArray(result);
37    }
38
39    /**
40    Since multiple users will be handled by
        threads within the same JVM, there is no
        need of
41    file-locking on the pipe connecting with
        Mercury. It's just important to make sure
        that
42    threads will not try to use the pipe
        concurrently, and that's what the
        synchronized keyword
43    is for in this method.
44    */
45    public static synchronized void disambiguate(
        StringBuffer body, String[] strings, javax.
        servlet.http.HttpSession session)
46      throws Exception{
47
```

```
48        //Construct message:
49        StringBuffer b = new StringBuffer();
50        b.append("disambiguate([");
51        for(int i=0;i<strings.length;i++){
52          if(i>0)
53            b.append(',');
54          b.append("\""+strings[i]+"\"");
55        }
56        b.append("],\"/tmp/graphs/\").␣");
57
58        //Send message:
59        toMercury(b.toString());
60
61        //Analyze response:
62
63        final String[] response = fromMercury();
64
65        if(response[0].equals("tag_error")){
66
67          //Tag error occurred.
68          body.append("Sorry,␣we␣don't␣have␣the␣word
               ␣\""
69            +response[1]
70            +"\"␣in␣our␣lexicon.");
71        }else if(response[0].equals("ok")){
72
73          //It worked!
74          final int count
75           = Integer.parseInt(response[1]);
76
77          if(count>1)
78            body.append("Your␣query␣is␣ambiguous.");
79
80          if(count==1)
81            body.append("There's␣one␣ontological␣
                meaning.");
```

```
82          else
83            body.append("There are "
84              +count
85              +" possible ontological meanings.");
86
87          final String[] concepts = new String[count
              ];
88
89          Runtime runtime = Runtime.getRuntime();
90
91          for(int i=0;i<count;i++){
92            concepts[i]=response[i];
93
94            final String uniqueId
95             = "g_" + session.getId()
96             + "_" + new java.util.Date().getTime()
97             + "_" +i;
98
99            final String[] args
100            = {"dot",
101             "-Tpng",
102             "-o",
103             "/var/lib/tomcat-5.5/webapps/oq/images
                 /" + uniqueId + ".png",
104             "/tmp/graphs/g_" + i + ".dot"};
105
106           final int returnCode
107            = runtime.exec(args).waitFor();
108
109           if(returnCode==0)
110             body.append("<p><img src=\"images/"
111               + uniqueId + ".png\"/>");
112           else
113             throw new Exception("dot returned "+
                 returnCode);
114         }
```

```
115       }else{
116         throw new Exception("Unknown mercury 
            response: "
117             +response[0]);
118       }
119     }
120 
121 %>
122 
123 <HTML>
124 <BODY>
125 <%
126 //Disable caching:
127 response.setHeader("Cache-Control", "no-cache, 
      post-check=0, pre-check=0");
128 response.setHeader("Pragma", "no-cache");
129 response.setHeader("Expires", "Thu, 01 Dec 1994 
      16:00:00 GMT");
130 
131 try{
132   StringBuffer body = new StringBuffer();
133 
134   String query = request.getParameter( "query" )
        ;
135 
136   //If no query in request, use one from session
        :
137   if(query==null)
138     query = (String) session.getAttribute("query
          ");
139 
140   //Save current query in session:
141   if(query!=null)
142     session.setAttribute( "query", query );
143 
144   final String[] words
```

```
145       = (query==null
146       ?new String[0]
147       :query.toLowerCase().split("\\W+"));
148
149    if(words.length>0)
150       disambiguate(body,words,session);
151    else
152       body.append("No query.");
153    out.println(body.toString());
154 }catch(Exception e){
155    out.println("Problem occurred: "+e);
156    e.printStackTrace();
157 }
158 %>
159 </BODY>
160 </HTML>
```

## A.4 File M2PL.java

The following program was written for transforming Mercury code into Prolog, in order to use Prolog's top–level loop.

```
1  /* Copyright 2007 Bartlomiej Antoni Szymczak */
2
3  import java.util.*;
4  import java.io.*;
5
6  public class M2PL{
7
8      public static void main(String[] args)
           throws Exception{
9
10         final BufferedReader r
```

```
11                  = new BufferedReader(new
                        InputStreamReader(System.in));
12
13          String line;
14
15          boolean skip=false;
16
17          while((line=r.readLine())!=null){
18                if(skip){
19                      if(line.endsWith("."))
20                            skip=false;
21                      continue;
22                }
23                if(line.startsWith(":-")
24                     || line.startsWith("main(")){
25                      if(!line.endsWith("."))
26                            skip=true;
27                      continue;
28                }
29                System.out.println(line.replace('"',
                        '\''));
30          }
31    }
32 }
```

## A.5 File aux_io.m

The aux_io Mercury module.

```
1 :-module aux_io.
2
3 :-interface.
4
5 :-import_module
```

```
6          string ,
7          list ,
8          lexicon ,
9          io ,
10         grabber .
11
12 :-pred expect_see(string::in,
13                   io::di,
14                   io::uo) is det.
15
16 :-pred expect_tell(string::in,
17                    io::di,
18                    io::uo) is det.
19
20 :-pred read_line(io.result(string)::out,
21                  io::di,
22                  io::uo) is det.
23
24 :-pred expect_eof(io::di,
25                   io::uo) is det.
26
27 :-pred write_concept_graph(concept::in,
28                            list(word)::in,
29                            io::di,
30                            io::uo) is det.
31
32 :-pred save_concept_graphs(list({concept,list(
       word)})::in,
33                            string::in,
34                            io::di,
35                            io::uo) is det.
36
37 :-implementation.
38
39
40 :-import_module
```

```
41            html,
42            main,
43            exception,
44            ontology,
45            int.
46
47  :-pred succeed is det.
48  succeed.
49
50  write_concept_graph(C,T_S,!IO):-
51            write_string("digraph␣concept␣{\n",!IO),
52            write_string("label=\"",!IO),
53            write_list(T_S,"␣",write_word,!IO),
54            write_string("\"\n",!IO),
55            concept_to_graph_helper(C,0,_,!IO),
56            write_string("}\n",!IO).
57
58
59  :-pred concept_to_graph_helper(concept::in,
60                                  int::in,
61                                  int::out,
62                                  io::di,
63                                  io::uo) is det.
64
65  concept_to_graph_helper(c(C,Rs),LastN,NewN,!IO)
        :-
66            io.write_string("n_",!IO),
67            MyN=int.plus(LastN,1),
68            io.write_int(MyN,!IO),
69            io.write_string("␣[label=\"",!IO),
70            io.write(C,!IO),
71            io.write_string("\"]\n",!IO),
72            roles_to_graph(Rs,MyN,MyN,NewN,!IO).
73
74  :-pred roles_to_graph(list(rolePair)::in,
75                        int::in,
```

```
76                               int::in,
77                               int::out,
78                               io::di,
79                               io::uo) is det.
80
81  roles_to_graph([], _ParentN, LastN, LastN, !IO).
82
83  roles_to_graph([r(R,c(C,Rs))|T], ParentN, LastN,
        NewLastN_2, !IO):-
84          MyN=int.plus(LastN,1),
85          io.write_string("n_",!IO),
86          io.write_int(MyN,!IO),
87          io.write_string(" [label=\"",!IO),
88          io.write(C,!IO),
89          io.write_string("\"]\n",!IO),
90
91          io.write_string("n_",!IO),
92          io.write_int(ParentN,!IO),
93          io.write_string(" -> n_",!IO),
94          io.write_int(MyN,!IO),
95          io.write_string(" [label=\"",!IO),
96          io.write(R,!IO),
97          io.write_string("\"]\n",!IO),
98          roles_to_graph(Rs,MyN,MyN,NewLastN_1,!IO
                ),
99          roles_to_graph(T,ParentN,NewLastN_1,
                NewLastN_2,!IO).
100
101 save_concept_graphs(Cs_T_Ss,DotDirectory,!IO):-
102         save_concept_graphs_count(list.length(
                Cs_T_Ss),
103                                   DotDirectory,
104                                   !IO),
105         save_concept_graphs_helper(Cs_T_Ss,0,
                DotDirectory,!IO).
106
```

```
107 :-pred save_concept_graphs_count(int::in,
108                                   string::in,
109                                   io::di,
110                                   io::uo) is det.
111
112 save_concept_graphs_count(N,DotDirectory,!IO):-
113         string.append(DotDirectory,"count.txt",
               FileName),
114         io.tell(FileName,Result,!IO),
115         (if Result=ok
116         then
117         io.write_int(N,!IO),
118          io.nl(!IO),
119          io.told(!IO)
120         else
121         throw("Cannot tell graphs/count.txt")).
122
123 :-pred save_concept_graphs_helper(list({concept,
       list(word)})::in,
124                                    int::in,
125                                    string::in,
126                                    io::di,
127                                    io::uo) is det
                                          .
128
129 save_concept_graphs_helper([],_,_,!_).
130
131 save_concept_graphs_helper([{C,T_S}|Rest],N,
       DotDirectory,!IO):-
132         string.int_to_string(N,NString),
133         string.append(DotDirectory,"g_",
               FileName0),
134         string.append(FileName0,NString,
               BaseFileName),
135         string.append(BaseFileName,".dot",
               FileName),
```

```
136         io.tell(FileName, Result, !IO),
137         (if Result=ok
138         then
139         write_concept_graph(C, T_S, !IO),
140          io.told(!IO)
141         else
142         throw(Result)),
143         save_concept_graphs_helper(Rest,
144                                           int.plus(N,1)
                                                ,
145                                           DotDirectory,
146                                           !IO).
147
148 :- pred write_word(word::in,
149                    io::di,
150                    io::uo) is det.
151
152 write_word(word(html.pstring(_P0, _P1, W),
153                  Category,
154                  _),
155             !IO):-
156         io.write_string(W, !IO),
157         io.write_string("[", !IO),
158         io.write(Category, !IO),
159         io.write_string("]", !IO).
160
161 read_line(Result, !IO):-
162         io.read_line_as_string(Line_Option, !IO),
163         (
164          Line_Option=ok(Line),
165          Result=ok(string.chomp(Line))
166         ;
167          Line_Option=eof,
168          Result=eof
169         ;
170          Line_Option=error(Error),
```

```
171             Result=error(Error)
172            ).
173
174 expect_see(FileName,!IO):-
175            io.see(FileName,Result,!IO),
176            (if Result=ok
177            then succeed
178            else
179            string.append("Cannot see ",FileName,
                   ErrorMessage),
180             throw(ErrorMessage)).
181
182 expect_eof(!IO):-
183            io.ignore_whitespace(Char_Option,!IO),
184            (if Char_Option=eof
185            then succeed
186            else throw(Char_Option)).
187
188 expect_tell(FileName,!IO):-
189            io.tell(FileName,Result,!IO),
190            (if Result=ok
191            then succeed
192            else
193            string.append("Cannot tell ",FileName,
                   ErrorMessage),
194             throw(ErrorMessage)).
```

## A.6 File pipe.m

The pipe Mercury module.

```
1 :-module pipe.
2
3 :-interface.
```

```
4
5  :-import_module
6          html,
7          list ,
8          io ,
9          grabber ,
10         string .
11
12 %% Commands that the server could send:
13 :-type command --->
14         disambiguate(list(html.pstring), %
               sentence
15                         string).   %directory to
                              save graphs
16
17 :-pred read_command_from_pipe(command::out,
18                                io::di,
19                                io::uo) is det.
20
21 :-pred save_ok_to_pipe(list(concept)::in,
22                          io::di,
23                          io::uo) is det.
24
25 :-pred save_tag_error_to_pipe(pstring::in,
26                                io::di,
27                                io::uo) is det.
28
29 :-implementation.
30
31 :-import_module
32         ontology ,
33         lexicon ,
34         aux_io ,
35         exception .
36
37 read_command_from_pipe(Result ,!IO):-
```

```
38            expect_see("/tmp/serverToMercury",!IO),
39            io.read(Command_Option,!IO),
40            (if Command_Option=ok(Command)
41            then Result=Command
42            else throw(Command_Option)),
43            expect_eof(!IO),
44            io.seen(!IO).
45
46 save_ok_to_pipe(Cs,!IO):-
47            expect_tell("/tmp/mercuryToServer",!IO),
48            io.write_string("ok",!IO),
49            io.nl(!IO),
50            list.length(Cs,Length),
51            io.write_int(Length,!IO),
52            io.nl(!IO),
53            io.write_list(Cs,"\n",write,!IO),
54            io.told(!IO).
55
56 save_tag_error_to_pipe(html.pstring(_P0,_P1,Word
       ),
57                                   !IO):-
58            expect_tell("/tmp/mercuryToServer",!IO),
59            io.write_string("tag_error",!IO),
60            io.nl(!IO),
61            io.write_string(Word,!IO),
62            io.nl(!IO),
63            io.told(!IO).
```

## A.7 File wn.m

The wn Mercury module.

```
1 :-module wn.
2
```

```
3  :-interface.
4
5  :-import_module set,
6          map,
7          string,
8          io,
9          lexicon.
10
11 :-type wn --->
12         wn(map(string,set(lexicon.part_of_speech
               ))).
13
14 :-pred load_wn(string::in,
15                wn::out,
16                io::di,
17                io::uo) is det.
18
19 :-pred save_wn(string::in,
20                wn::in,
21                io::di,
22                io::uo) is det.
23
24 :-pred valid(wn::in,
25              string::in,
26              lexicon.part_of_speech::in) is
                   semidet.
27
28 :-pred write(wn::in,
29              lexicon.part_of_speech::in,
30              io::di,
31              io::uo) is det.
32
33 :-pred get_wn_info(wn::in,
34                    string::in,
35                    lexicon.part_of_speech::out)
                        is nondet.
```

```
36
37 :-implementation.
38
39 :-import_module io,
40         aux_io,
41         exception,
42         assoc_list,
43         pair,
44         list,
45         term_io.
46
47 :-pred load(string::in,
48             wn::in,
49             lexicon.part_of_speech::in,
50             string::in, %POS as string
51             wn::out,
52             io::di,
53             io::uo) is det.
54
55 load(Directory,
56      WN_0,
57      P_O_S,
58      P_O_S_String,
59      WN_2,
60      !IO):-
61         string.append(Directory,
62                       P_O_S_String,
63                       Base_File_Name),
64         string.append(Base_File_Name,
65                       ".txt",
66                       Main_File_Name),
67         aux_io.expect_see(Main_File_Name,!IO),
68         read_entries(P_O_S,
69                      WN_0,
70                      WN_1,
71                      !IO),
```

```
72             io.seen(!IO),
73             string.append(Base_File_Name,
74                           "_manual.txt",
75                           Manual_File_Name),
76             aux_io.expect_see(Manual_File_Name,!IO),
77             read_entries(P_O_S,
78                          WN_1,
79                          WN_2,
80                          !IO),
81             io.seen(!IO).
82
83 load_wn(Directory,
84             wn(Optimized_Map),
85             !IO):-
86             map.init(Empty_Map),
87             load(Directory,
88                  wn(Empty_Map),
89                  n,
90                  "n",
91                  WN_1,
92                  !IO),
93             load(Directory,
94                  WN_1,
95                  v,
96                  "v",
97                  WN_2,
98                  !IO),
99             load(Directory,
100                 WN_2,
101                 adj,
102                 "adj",
103                 WN_3,
104                 !IO),
105            load(Directory,
106                 WN_3,
107                 adv,
```

```
108                 "adv",
109                 WN_4,
110                 !IO),
111         load(Directory,
112                 WN_4,
113                 cnj,
114                 "cnj",
115                 WN_5,
116                 !IO),
117         load(Directory,
118                 WN_5,
119                 d,
120                 "d",
121                 WN_6,
122                 !IO),
123         load(Directory,
124                 WN_6,
125                 prep,
126                 "prep",
127                 WN_7,
128                 !IO),
129         load(Directory,
130                 WN_7,
131                 prn,
132                 "prn",
133                 wn(Final_Map),
134                 !IO),
135         map.optimize(Final_Map,Optimized_Map).
136 
137 save_wn(Directory,
138         WN,
139         !IO):-
140         string.append(Directory,
141                       "nouns.txt",
142                       Nouns_File_Name),
143         aux_io.expect_tell(Nouns_File_Name,!IO),
```

```
144            write(WN,n,!IO),
145            io.told(!IO),
146            string.append(Directory,
147                          "verbs.txt",
148                          Verbs_File_Name),
149            aux_io.expect_tell(Verbs_File_Name,!IO),
150            write(WN,v,!IO),
151            io.told(!IO),
152            string.append(Directory,
153                          "adjectives.txt",
154                          Adjectives_File_Name),
155            aux_io.expect_tell(Adjectives_File_Name
                   ,!IO),
156            write(WN,adj,!IO),
157            io.told(!IO),
158            string.append(Directory,
159                          "adverbs.txt",
160                          Adverbs_File_Name),
161            aux_io.expect_tell(Adverbs_File_Name,!IO
                   ),
162            write(WN,adv,!IO),
163            io.told(!IO).
164
165 :-pred read_entries(lexicon.part_of_speech::in,
166                     wn::in,
167                     wn::out,
168                     io::di,
169                     io::uo) is det.
170
171 read_entries(P_O_S,
172              wn(Input_Map),
173              Output_WN,
174              !IO):-
175            aux_io.read_line(Line_Option,!IO),
176            (
177             Line_Option=ok(Line),
```

```
178             (if map.search(Input_Map,
179                            Line,
180                            Previous_Set)
181             then (set.insert(Previous_Set,
182                              P_O_S,
183                              New_Set),
184                   map.set(Input_Map,
185                           Line,
186                           New_Set,
187                           New_Map),
188                   read_entries(P_O_S,
189                                wn(New_Map),
190                                Output_WN,
191                                !IO)
192                  )
193             else (set.singleton_set(New_Set,P_O_S),
194                   map.set(Input_Map,
195                           Line,
196                           New_Set,
197                           New_Map),
198                   read_entries(P_O_S,
199                                wn(New_Map),
200                                Output_WN,
201                                !IO)
202                  )
203             )
204            ;
205             Line_Option=eof,
206             Output_WN=wn(Input_Map)
207            ;
208             Line_Option=error(Error),
209             throw(Error)
210            ).
211
212 valid(wn(Map),W,P_O_S):-
213            map.search(Map,W,Set),
```

```
214          set.contains(Set,P_O_S).
215
216  write(wn(Map),P_O_S,!IO):-
217          map.to_assoc_list(Map,List),
218          list.filter(pos_pred(P_O_S),List,
                 Filtered),
219          list.map(pair.fst,Filtered,Words),
220          io.write_list(Words,
221                          "\n",
222                          io.write_string,
223                          !IO).
224
225  :-pred pos_pred(lexicon.part_of_speech::in,
226                  pair(string,set(lexicon.
                         part_of_speech))::in)
227  is semidet.
228
229  pos_pred(P_O_S,Pair):-
230          pair.snd(Pair,Set),
231          set.contains(Set,P_O_S).
232
233  get_wn_info(wn(Map),
234              Word,
235              P_O_S):-
236          map.search(Map,Word,Set),
237          set.member(P_O_S,Set).
```

## A.8 File ontology.m

The ontology Mercury module.

```
1  :-module ontology.
2
3  :-interface.
```

```
4
5  :-pred isa(simpleConcept,simpleConcept).
6  :-mode isa(in,in) is semidet.
7  :-mode isa(in,out) is nondet.
8
9  :-type simpleConcept --->
10         c_process;
11         c_biological_process;
12         c_animate_process;
13         c_temporal_interval;
14         c_occurrent;
15         c_blood;
16         c_causing;
17         c_cell;
18         c_continuant;
19         c_conversion;
20         c_entity;
21         c_excretion;
22         c_forcing;
23         c_glucose;
24         c_glycogen;
25         c_insulin;
26         c_john;
27         c_level;
28         c_liver;
29         c_pool;
30         c_storage;
31         c_substance;
32         c_swimming;
33         c_winter;
34         c_catching;
35         c_lowered;
36         c_quality;
37         c_cause;
38         c_form.
39
```

```
40 :-implementation.
41 isa(c_pool,c_continuant). isa(c_pool,c_entity).
       isa(c_pool,c_pool).
42 isa(c_continuant,c_continuant). isa(c_continuant
       ,c_entity).
43 isa(c_level,c_continuant). isa(c_level,c_entity)
       . isa(c_level,c_level).
44 isa(c_temporal_interval,c_entity). isa(
       c_temporal_interval,c_occurrent). isa(
       c_temporal_interval,c_temporal_interval).
45 isa(c_substance,c_continuant). isa(c_substance,
       c_entity). isa(c_substance,c_substance).
46 isa(c_cell,c_cell). isa(c_cell,c_continuant).
       isa(c_cell,c_entity).
47 isa(c_liver,c_continuant). isa(c_liver,c_entity)
       . isa(c_liver,c_liver).
48 isa(c_storage,c_biological_process). isa(
       c_storage,c_entity). isa(c_storage,c_occurrent
       ). isa(c_storage,c_process). isa(c_storage,
       c_storage).
49 isa(c_excretion,c_biological_process). isa(
       c_excretion,c_entity). isa(c_excretion,
       c_excretion). isa(c_excretion,c_occurrent).
       isa(c_excretion,c_process).
50 isa(c_quality,c_continuant). isa(c_quality,
       c_entity). isa(c_quality,c_quality).
51 isa(c_glucose,c_continuant). isa(c_glucose,
       c_entity). isa(c_glucose,c_glucose). isa(
       c_glucose,c_substance).
52 isa(c_john,c_continuant). isa(c_john,c_entity).
       isa(c_john,c_john).
53 isa(c_biological_process,c_biological_process).
       isa(c_biological_process,c_entity). isa(
       c_biological_process,c_occurrent). isa(
       c_biological_process,c_process).
54 isa(c_forcing,c_causing). isa(c_forcing,c_entity
```

```
   ). isa(c_forcing,c_forcing). isa(c_forcing,
   c_occurrent). isa(c_forcing,c_process).
55 isa(c_entity,c_entity).
56 isa(c_causing,c_causing). isa(c_causing,c_entity
   ). isa(c_causing,c_occurrent). isa(c_causing,
   c_process).
57 isa(c_catching,c_animate_process). isa(
   c_catching,c_catching). isa(c_catching,
   c_entity). isa(c_catching,c_occurrent). isa(
   c_catching,c_process).
58 isa(c_insulin,c_continuant). isa(c_insulin,
   c_entity). isa(c_insulin,c_insulin). isa(
   c_insulin,c_substance).
59 isa(c_occurrent,c_entity). isa(c_occurrent,
   c_occurrent).
60 isa(c_process,c_entity). isa(c_process,
   c_occurrent). isa(c_process,c_process).
61 isa(c_swimming,c_animate_process). isa(
   c_swimming,c_entity). isa(c_swimming,
   c_occurrent). isa(c_swimming,c_process). isa(
   c_swimming,c_swimming).
62 isa(c_cause,c_cause). isa(c_cause,c_continuant).
    isa(c_cause,c_entity).
63 isa(c_conversion,c_biological_process). isa(
   c_conversion,c_conversion). isa(c_conversion,
   c_entity). isa(c_conversion,c_occurrent). isa(
   c_conversion,c_process).
64 isa(c_glycogen,c_continuant). isa(c_glycogen,
   c_entity). isa(c_glycogen,c_glycogen). isa(
   c_glycogen,c_substance).
65 isa(c_winter,c_entity). isa(c_winter,c_occurrent
   ). isa(c_winter,c_temporal_interval). isa(
   c_winter,c_winter).
66 isa(c_form,c_continuant). isa(c_form,c_entity).
   isa(c_form,c_form).
67 isa(c_blood,c_blood). isa(c_blood,c_continuant).
```

```
    isa(c_blood,c_entity).
68 isa(c_lowered,c_continuant). isa(c_lowered,
      c_entity). isa(c_lowered,c_lowered). isa(
      c_lowered,c_quality).
69 isa(c_animate_process,c_animate_process). isa(
      c_animate_process,c_entity). isa(
      c_animate_process,c_occurrent). isa(
      c_animate_process,c_process).
```

## A.9 File main.m

The main Mercury module.

```
1  :-module main.
2
3  :-interface.
4
5  :-import_module io.
6
7  :-pred main(io::di,io::uo) is det.
8
9  :-implementation.
10
11 :-import_module
12         string,
13         list,
14         hash_table,
15         assoc_list,
16         maybe,
17         ontology,
18         lexicon,
19         int,
20         exception,
21         grabber,
```

```
22          solutions,
23          aux_io,
24          pipe,
25          html,
26          wn,
27          obo,
28          map.
29
30  main(!IO):-
31          io.write_string("Loading WordNet...\n",!
               IO),
32          wn.load_wn("../WordNet/dict/",WN,!IO),
33          io.write_string("WordNet loaded.\n",!IO)
               ,
34
35          io.write_string("Saving WordNet copy...\
               n",!IO),
36          wn.save_wn("../WordNet/dict/saved/",WN,!
               IO),
37          io.write_string("WordNet copy saved.\n"
               ,!IO),
38
39          %%Load OBO
40          io.write_string("Loading OBO...\n",!IO),
41          obo.load_obo("../obo/",
42                      OBO,
43                      !IO),
44          io.write_string("OBO loaded.\n",!IO),
45          obo.write_stats(OBO,!IO),
46
47          %%Load HTML
48          io.write_string("Loading HTML...\n",!IO)
               ,
49          aux_io.expect_see("../Wikipedia/Insulin.
               html",!IO),
50          html.read_sentences(Sentences,!IO),
```

```
51          io.seen(!IO),
52          io.write_string("HTML loaded.\n",!IO),
53
54          io.write_string("Writing tag errors HTML
                .\n",!IO),
55          aux_io.expect_tell("tags.html",!IO),
56          html.tag_and_write_sentences(WN,
57                                      OBO,
58                                      Sentences,
59                                      !IO),
60          io.told(!IO),
61          io.write_string("Tag errors HTML written
                .\n",!IO),
62
63          %Enter infinite loop
64          main_loop(WN,!IO).
65
66  %%Main program loop
67  :-pred main_loop(wn::in,
68                   io::di,io::uo) is det.
69
70  main_loop(WN,!IO):-
71          pipe.read_command_from_pipe(Command,!IO)
                ,
72          interpret(WN,Command,!IO),
73          main_loop(WN,!IO).
74
75  :-pred interpret(wn::in,
76                   command::in,
77                   io::di,
78                   io::uo) is det.
79
80  interpret(WN,disambiguate(S,DotDirectory),!IO):-
81          (if has_tag_error(WN,S,TagError)
82          then save_tag_error_to_pipe(TagError,!IO
                )
```

```
83            else  (solutions(grab_main(WN,S),
84                                  Solutions),
85                     %%io.write_list(Solutions,"\n",io.
                           write,!IO),
86                     save_concept_graphs(Solutions,
                           DotDirectory,!IO),
87                     first_from_each_pair(Solutions,Cs)
                           ,
88                     save_ok_to_pipe(Cs,!IO))).
89
90
91  :-pred  has_tag_error(wn::in,
92                             list(pstring)::in,
93                             pstring::out) is semidet.
94
95  has_tag_error(WN,[W|S],Error_W):-
96            (if  lexicon.getLexicalInfo(WN,W,_)
97            then  has_tag_error(WN,S,Error_W)
98            else  Error_W=W).
99
100 :-pred  first_from_each_pair(list({T1,T2})::in,
101                                        list(T1)::out) is
                                              det.
102
103 first_from_each_pair([],[]).
104
105 first_from_each_pair([{X,_}|Rest],[X|Result]):-
106           first_from_each_pair(Rest,Result).
```

## A.10 File `lexicon.m`

The `lexicon` Mercury module.

```
1  :-module lexicon.
2
3  :-interface.
4
5  :-import_module html.
6  :-import_module list.
7  :-import_module ontology.
8  :-import_module maybe.
9  :-import_module wn.
10
11 :-type part_of_speech --->
12         adj;                    %adjective (nice
               , fast)
13         adv;                    %adverb (nicely,
               quickly)
14         cnj;                    %conjunction (
               and, or)
15         d;                      %d (the)
16         n;                      %n (bike, tree)
17         prep;                   %prep (in, on)
18         prn;                    %prn (it)
19         v.                      %verb (increase,
               excrete)
20
21 :-type word ---> word(html.pstring,
22                       lexicon.part_of_speech,
23                       maybe(simpleConcept)).
24
25 %%Part of speech tagging
26 %%Word to concept conversion
27 :-pred getLexicalInfo(wn::in,
28                       html.pstring::in,
29                       word::out) is nondet.
30
31 :-type taggedSentence --->
32         tagged(list(word));
```

```
33          tagError(html.pstring).
34
35 :-pred tag(wn::in,
36              list(html.pstring)::in,
37              taggedSentence::out) is multi.
38
39 :-implementation.
40
41 :-import_module
42          exception,
43          string.
44
45 :-pred ask_wn(wn::in,
46                   string::in,          %Input word, e.g
                          . catched
47                   part_of_speech::out,
48                   string::out)         %Base form, e.g.
                         catch
49 is nondet.
50
51 %%Get info from internal database:
52 getLexicalInfo(WN,
53                    pstring(P0,P1,Word),
54                    word(pstring(P0,P1,Word),
55                           P_O_S,
56                           Concept_Option)):-
57          string.to_lower(Word,Lower_Word),
58          ask_wn(WN,
59                    Lower_Word,
60                    P_O_S,
61                    Base_Form),
62          (if linfo(Base_Form,SimpleConcept)
63          then Concept_Option=yes(SimpleConcept)
64          else Concept_Option=no).
65
66 %%Word already in base form:
```

```
67  ask_wn(WN,
68          Word,
69          PartOfSpeech,
70          Word):-
71           wn.get_wn_info(WN,
72                          Word,
73                          PartOfSpeech).
74
75  %%therapies -> therapy , cries -> cry
76  ask_wn(WN,
77          Therapies,
78          P_O_S,
79          Therapy):-
80           string.remove_suffix(Therapies,"ies",
               Therap),
81           string.append(Therap,"y",Therapy),
82           wn.get_wn_info(WN,
83                          Therapy,
84                          P_O_S).
85
86  %%articles -> article creates -> create
87  ask_wn(WN,
88          Articles,
89          P_O_S,
90          Article):-
91           string.remove_suffix(Articles,"s",
               Article),
92           wn.get_wn_info(WN,
93                          Article,
94                          P_O_S).
95
96  %%supplied -> supply
97  ask_wn(WN,
98          Supplied,
99          v,
100         Supply):-
```

```
101             string.remove_suffix(Supplied,"ied",
                    Suppl),
102             string.append(Suppl,"y",Supply),
103             wn.get_wn_info(WN,
104                            Supply,
105                            v).
106
107 %%released -> release
108 ask_wn(WN,
109        Released,
110        v,
111        Release):-
112             string.remove_suffix(Released,"d",
                    Release),
113             wn.get_wn_info(WN,
114                            Release,
115                            v).
116
117 %%worked -> work
118 ask_wn(WN,
119        Worked,
120        v,
121        Work):-
122             string.remove_suffix(Worked,"ed",Work),
123             wn.get_wn_info(WN,
124                            Work,
125                            v).
126
127 :-pred linfo(string::in,
128              simpleConcept::out) is nondet.
129
130 linfo("catched",c_catching).
131 linfo("catches",c_catching).
132 linfo("cause",c_causing).
133 linfo("convert",c_conversion).
134 linfo("excrete",c_excretion).
```

```
135 linfo("forces",c_forcing).
136 %%linfo("form",c_forming).
137
138 linfo("lowered",c_lowered).
139
140 linfo("swimmed",c_swimming).
141
142 linfo("swims",c_swimming).
143
144 linfo("blood",c_blood).
145 linfo("cause",c_cause).
146 linfo("cells",c_cell).
147 linfo("form",c_form).
148 linfo("glucose",c_glucose).
149 linfo("glycogen",c_glycogen).
150 linfo("insulin",c_insulin).
151 linfo("john",c_john).
152 linfo("levels",c_level).
153 linfo("liver",c_liver).
154 linfo("pool",c_pool).
155 linfo("storage",c_storage).
156 linfo("winter",c_winter).
157
158 % :-pred uselessWord(string::in) is semidet.
159
160 % uselessWord("a").
161 % uselessWord("her").
162 % uselessWord("his").
163 % uselessWord("its").
164 % uselessWord("mine").
165 % uselessWord("my").
166 % uselessWord("our").
167 % uselessWord("ours").
168 % uselessWord("the").
169 % uselessWord("their").
170 % uselessWord("theirs").
```

```
171 % uselessWord("your").
172 % uselessWord("yours").
173
174 % Prunes a list of words by eliminating useless
      words.
175 % Preserves the order.
176 %:-pred prune(list(string)::in, list(string)::out
      ) is det.
177
178 %prune([],[]).
179
180 %prune([H|T],RES):-prune(T,PT),
181 %           (uselessWord(H) -> RES=PT ; RES=[H|PT]).
182
183 tag(_WN,[],tagged([])).
184
185 tag(WN,[W|Ws],Result):-
186             (if lexicon.getLexicalInfo(WN,W,Tagged_W
                  )
187             then (tag(WN,Ws,T_Ws_Option),
188                   (if T_Ws_Option=tagged(T_Ws)
189                   then Result=tagged([Tagged_W|T_Ws
                          ])
190                   else Result=T_Ws_Option))
191             else Result=tagError(W)).
```

## A.11 File obo.m

The obo Mercury module.

```
1 :-module obo.
2
3 :-interface.
4
```

```
5  :- import_module
6          io,
7          map,
8          string,
9          int,
10         set.
11
12 %:- pred main(io::di, io::uo) is det.
13
14 :- type obo_class_name == string.
15 :- type obo_class_label == string.
16 :- type obo_class == int.
17 :- type obo --->
18         obo(map(obo_class_name,
19                 obo_class),      %'GO:0153' -> 34
20             map(obo_class_label,
21                 obo_class),      %"insulin" -> 21
22             map(obo_class,
23                 set(obo_class)), %subclass
24             int).                %Least unused ID
                .
25
26 :- pred load_obo(string::in, %directory
27                 obo::out,
28                 io::di,
29                 io::uo) is det.
30
31 :- pred write_stats(obo::in,
32                     io::di,
33                     io::uo) is det.
34
35 :- pred label(obo::in,
36               string::in,
37               obo_class::out) is semidet.
38
39 :- implementation.
```

```
40
41 :-import_module
42           aux_io,
43           exception.
44
45 label(obo(_,Label_Map,_,_),
46        Label,
47        Class):-
48          map.search(Label_Map,
49                     Label,
50                     Class).
51
52 write_stats(obo(Input_Class_Map,
53                 Input_Label_Map,
54                 Input_Subclass_Map,
55                 _Next),!IO):-
56           io.write_string("OBO #classes: ",!IO),
57           io.write_int(map.count(Input_Class_Map)
                 ,!IO),
58           io.nl(!IO),
59           io.write_string("OBO #labels: ",!IO),
60           io.write_int(map.count(Input_Label_Map)
                 ,!IO),
61           io.nl(!IO),
62           io.write_string("OBO #subclass: ",!IO),
63           io.write_int(map.count(
                Input_Subclass_Map),!IO),
64           io.nl(!IO).
65
66 :-pred init(obo::out) is det.
67
68 init(obo(Empty_Classes,
69          Empty_Labels,
70          Empty_Subclass,0)):-
71          map.init(Empty_Classes),
72          map.init(Empty_Labels),
```

```
73            map.init(Empty_Subclass).
74
75
76  load_obo(Directory,
77           OBO,
78           !IO):-
79           init(OBO_0),
80           load(Directory,
81                "arabidopsis_development.pro",
82                OBO_0,
83                OBO_1,
84                !IO),
85           load(Directory,
86                "biological_process.pro",
87                OBO_1,
88                OBO_2,
89                !IO),
90           load(Directory,
91                "brenda.pro",
92                OBO_2,
93                OBO_3,
94                !IO),
95           load(Directory,
96                "caro.pro",
97                OBO_3,
98                OBO_4,
99                !IO),
100          load(Directory,
101               "cell.pro",
102               OBO_4,
103               OBO_5,
104               !IO),
105          load(Directory,
106               "cellular_component.pro",
107               OBO_5,
108               OBO_6,
```

```
109                 !IO),
110         load(Directory,
111                 "chebi.pro",
112                 OBO_6,
113                 OBO_7,
114                 !IO),
115         load(Directory,
116                 "disease_ontology.pro",
117                 OBO_7,
118                 OBO_8,
119                 !IO),
120         load(Directory,
121                 "event.pro",
122                 OBO_8,
123                 OBO_9,
124                 !IO),
125         load(Directory,
126                 "evidence_code.pro",
127                 OBO_9,
128                 OBO_10,
129                 !IO),
130         load(Directory,
131                 "evoc.pro",
132                 OBO_10,
133                 OBO_11,
134                 !IO),
135         load(Directory,
136                 "fix.pro",
137                 OBO_11,
138                 OBO_12,
139                 !IO),
140         load(Directory,
141                 "fma_lite.pro",
142                 OBO_12,
143                 OBO_13,
144                 !IO),
```

```
145             load(Directory,
146                 "human-dev-anat-abstract.pro",
147                 OBO_13,
148                 OBO_14,
149                 !IO),
150             load(Directory,
151                 "human-dev-anat-staged.pro",
152                 OBO_14,
153                 OBO_15,
154                 !IO),
155             load(Directory,
156                 "image.pro",
157                 OBO_15,
158                 OBO_16,
159                 !IO),
160             load(Directory,
161                 "mammalian_phenotype.pro",
162                 OBO_16,
163                 OBO_17,
164                 !IO),
165             load(Directory,
166                 "mao.pro",
167                 OBO_17,
168                 OBO_18,
169                 !IO),
170             load(Directory,
171                 "mesh.pro",
172                 OBO_18,
173                 OBO_19,
174                 !IO),
175             load(Directory,
176                 "mged.pro",
177                 OBO_19,
178                 OBO_20,
179                 !IO),
180             load(Directory,
```

```
181                "molecular_function.pro",
182                OBO_20,
183                OBO_21,
184                !IO),
185          load(Directory,
186                "molecule_role.pro",
187                OBO_21,
188                OBO_22,
189                !IO),
190          load(Directory,
191                "ncbi_taxonomy.pro",
192                OBO_22,
193                OBO_23,
194                !IO),
195          load(Directory,
196                "obi.pro",
197                OBO_23,
198                OBO_24,
199                !IO),
200          load(Directory,
201                "pathway.pro",
202                OBO_24,
203                OBO_25,
204                !IO),
205          load(Directory,
206                "plant_environment.pro",
207                OBO_25,
208                OBO_26,
209                !IO),
210          load(Directory,
211                "plant_trait.pro",
212                OBO_26,
213                OBO_27,
214                !IO),
215          load(Directory,
216                "psi-mi.pro",
```

```
217                         OBO_27,
218                         OBO_28,
219                         !IO),
220                 load(Directory,
221                         "psi-mod.pro",
222                         OBO_28,
223                         OBO_29,
224                         !IO),
225                 load(Directory,
226                         "quality.pro",
227                         OBO_29,
228                         OBO_30,
229                         !IO),
230                 load(Directory,
231                         "rex.pro",
232                         OBO_30,
233                         OBO_31,
234                         !IO),
235                 load(Directory,
236                         "sep.pro",
237                         OBO_31,
238                         OBO_32,
239                         !IO),
240                 load(Directory,
241                         "sequence.pro",
242                         OBO_32,
243                         OBO_33,
244                         !IO),
245                 load(Directory,
246                         "systems_biology.pro",
247                         OBO_33,
248                         OBO_34,
249                         !IO),
250                 OBO_34=obo(Input_Class_Map,
251                            Input_Label_Map,
252                            Input_Subclass_Map,
```

```
253                         Next),
254          map.optimize(Input_Class_Map,
255                          Optimized_Class_Map),
256          map.optimize(Input_Label_Map,
257                          Optimized_Label_Map),
258          map.optimize(Input_Subclass_Map,
259                          Optimized_Subclass_Map),
260          OBO=obo(Optimized_Class_Map,
261                  Optimized_Label_Map,
262                  Optimized_Subclass_Map,
263                  Next).
264
265
266 %%Load single file.
267 :-pred load(string::in,          %Directory
268             string::in,          %Filename
269             obo::in,             %Read so far
270             obo::out,            %Result
271             io::di,
272             io::uo) is det.
273
274 load(Directory,
275      FileName,
276      OBO_0,
277      OBO_1,
278      !IO):-
279          string.append(Directory,
280                        FileName,
281                        Path),
282          aux_io.expect_see(Path,!IO),
283          read_entries(OBO_0,
284                       OBO_1,
285                       !IO),
286          io.seen(!IO).
287
288 %%Add new label for OBO class.
```

```
289 :-pred add_label(obo::in,          %Input OBO
290                  obo_class_name::in, %Class name
291                  obo_class_label::in, %Label
292                  obo::out)          %New OBO
293 is det.
294
295 add_label(obo(Input_Class_Map,
296               Input_Label_Map,
297               Input_Subclass_Map,
298               Next),
299           Class_Name,
300           Label,
301           obo(New_Class_Map,
302               New_Label_Map,
303               Input_Subclass_Map,
304               Output_Next)):-
305         string.to_lower(Label,Lower_Label),
306         (if is_good_label(Lower_Label)
307         then (if map.contains(Input_Class_Map,
308                                Class_Name)
309               then (map.lookup(Input_Class_Map,
310                                Class_Name,
311                                Class),
312                     map.set(Input_Label_Map,
313                             Lower_Label,
314                             Class,
315                             New_Label_Map),
316                     New_Class_Map=Input_Class_Map
                         ,
317                     Output_Next=Next
318                    )
319               else (Output_Next=int.plus(Next,1),
320                     map.set(Input_Class_Map,
321                             Class_Name,
322                             Next,
323                             New_Class_Map),
```

```
324                         map.set(Input_Label_Map,
325                                 Lower_Label,
326                                 Next,
327                                 New_Label_Map)
328                        )
329                   )
330         else (New_Class_Map=Input_Class_Map,
331               New_Label_Map=Input_Label_Map,
332               Output_Next=Next)
333         ).
334
335 :-pred is_good_label(string::in) is semidet.
336
337 is_good_label(Label):-
338         string.is_alpha(Label).
339
340 %%Add new subclass information for OBO classes.
341 :-pred add_subclass(obo::in,      %Input OBO
342                     obo_class_name::in, %
                            Subclass name
343                     obo_class_name::in, %
                            Superclass name
344                     obo::out)     %New OBO
345 is det.
346
347 add_subclass(Input_OBO,
348              Subclass_Name,
349              Superclass_Name,
350              obo(Class_Map,
351                  Label_Map,
352                  Output_Subclass_Map,
353                  Next)):-
354         insert_class(Input_OBO,
355                      Subclass_Name,
356                      OBO_1),
357         insert_class(OBO_1,
```

```
358                             Superclass_Name,
359                             obo(Class_Map,
360                                 Label_Map,
361                                 Subclass_Map,
362                                 Next)),
363         map.lookup(Class_Map,
364                    Subclass_Name,
365                    Subclass),
366         map.lookup(Class_Map,
367                    Superclass_Name,
368                    Superclass),
369         (if map.contains(Subclass_Map,
370                          Subclass)
371         then (map.lookup(Subclass_Map,
372                          Subclass,
373                          Superclasses),
374               set.insert(Superclasses,
375                          Superclass,
376                          New_Superclasses),
377               map.set(Subclass_Map,
378                       Subclass,
379                       New_Superclasses,
380                       Output_Subclass_Map)
381              )
382         else (set.singleton_set(Superclasses,
              Superclass),
383               set(Subclass_Map,
384                   Subclass,
385                   Superclasses,
386                   Output_Subclass_Map)
387              )
388         ).
389
390 :-pred insert_class(obo::in,
391                     obo_class_name::in,
392                     obo::out) is det.
```

```
393
394 insert_class(obo(Input_Class_Map,
395                  Label_Map,
396                  Subclass_Map,
397                  Input_Next),
398              Class_Name,
399              obo(Output_Class_Map,
400                  Label_Map,
401                  Subclass_Map,
402                  Output_Next)):-
403         (if map.contains(Input_Class_Map,
404                          Class_Name)
405         then (Output_Class_Map=Input_Class_Map,
406               Output_Next=Input_Next
407              )
408         else (Output_Next=int.plus(Input_Next,1)
             ,
409               map.set(Input_Class_Map,
410                       Class_Name,
411                       Input_Next,
412                       Output_Class_Map)
413              )
414         ).
415
416
417
418 %%parses binary predicate.
419 :-pred parse_binary(string::in,
420                     string::out,
421                     string::out) is semidet.
422
423 %%Form:('first', 'second'). %comment
424 parse_binary(All,First,Second):-
425         string.sub_string_search(All,
426                                  "', '",
427                                  Middle_Position
```

```
                                     ),
428           string.sub_string_search(All,
429                                    " ').",
430                                    End_Position),
431           string.left(All,
432                       Middle_Position,
433                       First),
434           string.substring(All,
435                            Middle_Position+4,
436                            End_Position-
                                 Middle_Position-4,
437                            Second).
438
439 :-pred read_entries(obo::in,
440                     obo::out,
441                     io::di,
442                     io::uo) is det.
443
444 read_entries(obo(Input_Class_Map,
445                  Input_Label_Map,
446                  Input_Subclass_Map,
447                  Next),
448              Output_OBO,
449              !IO):-
450           aux_io.read_line(Line_Option,!IO),
451           (
452            Line_Option=ok(Line),
453            (if (string.append("class('",
454                               Rest,
455                               Line),
456                 string.remove_suffix(Rest,
457                                      " ').",
458                                      Class_Name)
459                )
460            then (if map.contains(Input_Class_Map,
461                                  Class_Name)
```

```
462                then  read_entries(obo(
                      Input_Class_Map,
463                                   Input_Label_Map
                                        ,
464                                   Input_Subclass_Map
                                        ,
465                                   Next),
466                               Output_OBO,
467                               !IO)
468                else  (map.set(Input_Class_Map,
469                           Class_Name,
470                           Next,
471                           New_Class_Map),
472                     read_entries(obo(
                          New_Class_Map,
473                                    Input_Label_Map
                                         ,
474                                    Input_Subclass_Map
                                         ,
475                                    Next+1),
476                                Output_OBO,
477                                !IO)
478                     )
479                )
480           else if (string.append("metadata_db:
                  entity_label('",
481                               Rest,
482                               Line),
483                  parse_binary(Rest,
484                              Class_Name,
485                              Label)
486               )
487           then  (add_label(obo(Input_Class_Map,
488                                Input_Label_Map,
489                                Input_Subclass_Map,
490                                Next),
```

```
491                                    Class_Name,
492                                    Label,
493                                    New_OBO),
494                         read_entries(New_OBO,
495                                      Output_OBO,
496                                      !IO)
497                     )
498                 else if (string.append("metadata_db:
                        entity_synonym('",
499                                           Rest,
500                                           Line),
501                              parse_binary(Rest,
502                                           Class_Name,
503                                           Synonym)
504                         )
505                 then (add_label(obo(Input_Class_Map,
506                                     Input_Label_Map,
507                                     Input_Subclass_Map,
508                                     Next),
509                                 Class_Name,
510                                 Synonym,
511                                 New_OBO),
512                         read_entries(New_OBO,
513                                      Output_OBO,
514                                      !IO)
515                     )
516                 else if (string.append("subclass('",
517                                           Rest,
518                                           Line),
519                              parse_binary(Rest,
520                                           Subclass_Name,
521                                           Superclass_Name)
522                         )
523                 then (add_subclass(obo(Input_Class_Map,
524                                        Input_Label_Map,
525                                        Input_Subclass_Map
```

```
                                                         ,
526                                                  Next),
527                                             Subclass_Name,
528                                             Superclass_Name,
529                                             New_OBO),
530                      read_entries(New_OBO,
531                                   Output_OBO,
532                                   !IO)
533                     )
534              else   (read_entries(obo(Input_Class_Map,
535                                       Input_Label_Map,
536                                       Input_Subclass_Map
                                                         ,
537                                       Next),
538                                   Output_OBO,
539                                   !IO)
540                     )
541              )
542
543
544 %            (if  map.search(Input_Map,
545 %                            Line,
546 %                            Previous_Set)
547 %             then  (set.insert(Previous_Set,
548 %                               P_O_S,
549 %                               New_Set),
550 %                   map.set(Input_Map,
551 %                           Line,
552 %                           New_Set,
553 %                           New_Map),
554 %                   read_entries(P_O_S,
555 %                                wn(New_Map),
556 %                                Output_WN,
557 %                                !IO)
558 %                  )
559 %             else  (set.singleton_set(New_Set,P_O_S),
```

```
560 %                       map.set(Input_Map,
561 %                               Line,
562 %                               New_Set,
563 %                               New_Map),
564 %                       read_entries(P_O_S,
565 %                                    wn(New_Map),
566 %                                    Output_WN,
567 %                                    !IO)
568 %                   )
569 %           )
570             ;
571              Line_Option=eof,
572              Output_OBO=obo(Input_Class_Map,
573                             Input_Label_Map,
574                             Input_Subclass_Map,
575                             Next)
576             ;
577              Line_Option=error(Error),
578              throw(Error)
579             ).
```

## A.12 File startDaemon.sh

A script for starting the Mercury daemon. It creates the necessary unidirectional pipes for communication with the GUI.

```
1 #!/bin/bash
2
3  mkfifo /tmp/serverToMercury
4  mkfifo /tmp/mercuryToServer
5
6 chmod a+w /tmp/serverToMercury
7 chmod a+w /tmp/mercuryToServer
8
```

```
mkdir -p /tmp/graphs/
chmod 777 /tmp/graphs/
```

## A.13 File grabber.m

The grabber Mercury module.

```
:- module grabber.

:- interface.

:- import_module html.
:- import_module list.
:- import_module lexicon.
:- import_module ontology.
:- import_module wn.

:- type role --->
      tmp % temporal aspects (generic role)
    ; loc % location, position
    ; prp % purpose, function
    ; wrt % with respect to
    ; chr % characteristic (property ascription)
    ; cum % cum (i.e., with accompanying)
    ; bmo % by means of, instrument, via
    ; cby % caused by
    ; cau % causes
    ; cmp % comprising, has part
    ; pof % part of
    ; agt % agent of act or process
    ; pnt % patient of act or process
    ; src % source of act or process
    ; rst % result of act or process
    ; dst % destination of moving process
```

```
28      .
29
30  %Peirce product representation
31  :-type rolePair --->
32          r(role,concept).
33
34  %A class with a list of Peirce product
        restrictions
35  :-type concept --->
36          c(simpleConcept,            %Name of the
               concept
37            list(rolePair)            %Associated
                 attributes
38           ).
39
40  %Wrapper for the grab predicate
41  :-pred grab_main(wn::in,
42                   list(html.pstring)::in,
43                   {concept,list(word)}::out) is
                       nondet.
44
45  :-implementation.
46
47  :-import_module
48          maybe.
49
50  grab_main(WN,S,{C,TS}):-
51          lexicon.tag(WN,S,tagged(TS)),
52          grab(c_entity,s,TS,C).
53
54  %%Hinting based on CFG grammar for English
55  :-type syntacticHinting --->
56          s;
57          vp;
58          d_a_cnp_pp;
59          d_a_cnp;
```

```
60            a_cnp;
61            cnp;
62            pp.
63
64  %%Translate preposition to role
65  :-pred p2r(string::in,role::out) is nondet.
66  p2r("on",tmp).
67  p2r("on",loc).
68  p2r("to",dst).
69  p2r("in",tmp).
70  p2r("in",loc).
71  p2r("of",pnt).
72  p2r("of",wrt).
73
74  %%This predicate performs the ontological
        sentence analysis
75  :-pred grab(simpleConcept::in,
76                syntacticHinting::in,
77                list(word)::in,
78                concept::out) is nondet.
79
80  %C - Concept
81  %CN - ConceptName
82  %V - Verb
83  %VP - VerbPhrase
84  %NP - NounPhrase
85  %Rs - Roles
86  %S - Sentence
87  %H - Head
88  %T - Tail
89  grab(C,s,S,c(VP_C,[r(agt,NP_C)|VP_Rs])):-
90            append([NP_H|NP_T],[word(V,v,C_O)|VP_T],
                S),
91            grab(C,vp,[word(V,v,C_O)|VP_T],c(VP_C,
                VP_Rs)),
92            grab(c_entity,d_a_cnp_pp,[NP_H|NP_T],
```

```
                NP_C).
 93
 94  grab(CN,vp,[word(_,v,yes(V_C)),word(P,prep,no)|
         PPs_T],c(V_C,Rs)):-
 95          isa(V_C,CN),
 96          grab_rs(V_C,[word(P,prep,no)|PPs_T],Rs).
 97
 98  grab(CN,vp,[word(_,v,yes(V_C))|NP_PPs],c(V_C,[r(
         pnt,NP_C)|PP_Rs])):-
 99          isa(V_C,CN),
100          append([NP_H|NP_T],[word(P,prep,no)|
                PPs_T],NP_PPs),
101          grab(c_entity,d_a_cnp_pp,[NP_H|NP_T],
                NP_C),
102          grab_rs(V_C,[word(P,prep,no)|PPs_T],
                PP_Rs).
103
104  grab(CN,vp,[word(_,v,yes(V_C))|NP_PPs],c(V_C,[r(
         pnt,NP_C)])):-
105          isa(V_C,CN),
106          grab(c_entity,d_a_cnp_pp,NP_PPs,NP_C).
107
108  grab(CN,d_a_cnp_pp,NP_PPs,c(NP_CN,Rs)):-
109          append([NP_H|NP_T],PPs,NP_PPs),
110          grab(CN,d_a_cnp,[NP_H|NP_T],c(NP_CN,
                NP_Rs)),
111          isa(NP_CN,CN),
112          grab_rs(NP_CN,PPs,PPs_Rs),
113          append(NP_Rs,PPs_Rs,Rs).
114
115  grab(CN,d_a_cnp,[NP_H|NP_T],C):-
116          (NP_H=word(_,d,_)
117          ->
118           grab(CN,a_cnp,NP_T,C)
119          ;
120           grab(CN,a_cnp,[NP_H|NP_T],C)).
```

```
121
122 grab(CN, a_cnp, [word(_, adj, yes(A_C))|NP_T], c(C, [r
        (chr, c(A_C, []))|Rs])):-
123         grab(CN, a_cnp, NP_T, c(C, Rs)).
124
125 grab(CN, a_cnp, [word(N, n, N_C_O)|NP_T], C):-
126         grab(CN, cnp, [word(N, n, N_C_O)|NP_T], C).
127
128 grab(CN, cnp, [word(_, n, yes(N_C))], c(N_C, [])):-
129         isa(N_C, CN).
130
131 grab(CN, cnp,
132       [word(_, n, yes(N1_C)), word(N2, n, yes(N2_C))|
            NP_T],
133       c(C, [r(wrt, c(N1_C, []))|Rs])):-
134         grab(CN, cnp, [word(N2, n, yes(N2_C))|NP_T],
              c(C, Rs)).
135
136 :-pred grab_rs(simpleConcept::in,
137                list(word)::in,
138                list(rolePair)::out) is nondet.
139
140 grab_rs(__, [], []).
141
142 grab_rs(CN, [word(P, prep, no), PPs_H|PPs_T], [R|Rs])
      :-
143         append([PP1_H|PP1_T], REST, [word(P, prep,
              no), PPs_H|PPs_T]),
144         grab_r(CN, [PP1_H|PP1_T], R),
145         grab_rs(CN, REST, Rs).
146
147 :-pred grab_r(simpleConcept::in,
148               list(word)::in,
149               rolePair::out) is nondet.
150
151 grab_r(CN,
```

```
152            [word(pstring(_,_,P),prep,no)|NP],
153            r(R,NP_C)):-
154             p2r(P,R),
155             validRole(CN,R,CN_2),
156             grab(CN_2,d_a_cnp_pp,NP,NP_C).
157
158 grab_r(
159            CN,
160            [word(pstring(_,_,"in"),prep,_),
161             word(pstring(_,_,"form"),n,_),
162             word(pstring(_,_,"of"),prep,_)
163            |NP],
164            r(chr,NP_C)):-
165             isa(CN,c_substance),
166             grab(c_continuant,d_a_cnp_pp,NP,NP_C).
167
168 grab_r(
169            CN,
170            [word(pstring(_,_,"in"),prep,_),
171             word(pstring(_,_,"the"),d,_),
172             word(pstring(_,_,"form"),n,_),
173             word(pstring(_,_,"of"),prep,_)
174            |NP],
175            r(chr,NP_C)):-
176             isa(CN,c_substance),
177             grab(c_continuant,d_a_cnp_pp,NP,NP_C).
178
179 %%Valid role definitions
180 :-pred validRole_d(simpleConcept::in,
181                    role::in,
182                    simpleConcept::out) is
                          semidet.
183
184 validRole_d(c_causing,pnt,c_process).
185 validRole_d(c_occurrent,loc,c_continuant).
186 validRole_d(c_biological_process,pnt,c_substance
```

```
    ).
187 validRole_d(c_occurrent,tmp,c_occurrent).
188
189 validRole_d(c_continuant,loc,c_continuant).
190 validRole_d(c_continuant,wrt,c_entity).
191
192 %% Inferred valid role
193 :-pred validRole(simpleConcept::in,
194                  role::in,
195                  simpleConcept::out) is nondet.
196
197 validRole(CN,R,CN_2):-
198         isa(CN,CN_Ancestor),
199         validRole_d(CN_Ancestor,R,CN_2).
```

# A.14 File createOntology.m

The createOntology Mercury module.

```
1  :-module createOntology.
2
3  :-interface.
4
5  :-import_module io.
6
7  :-pred main(io::di,io::uo) is det.
8
9  :-implementation.
10
11 :-import_module list.
12 :-import_module assoc_list.
13 :-import_module std_util.
14 :-import_module hash_table.
15 :-import_module exception.
```

```
16 :-import_module string.
17 :-import_module char.
18
19 :-pred read_isa_d(isaDefinition::out,io::di,io::
     uo) is det.
20
21 read_isa_d(IsaDefinition,!IO):-
22         io.see("ontology_definition.txt",
              InResult,!IO),
23         (if InResult=ok
24         then isa_d(IsaDefinition,!IO),
25          io.seen(!IO)
26         else
27         io.write_string("Can't open input.",!IO)
              ,
28          io.nl(!IO),
29          throw("ontology_definition.txt")).
30
31 :-pred write_ontology(isaDefinition::in,
32                       isaTable::hash_table_ui,
33                       io::di,
34                       io::uo) is det.
35
36 write_ontology(IsaDefinition,IT,!IO):-
37          io.tell("ontology.m",Result,!IO),
38          (if Result=ok
39          then
40          write_preamble(!IO),
41           write_simpleConcept(IsaDefinition,!IO)
              ,
42           write_string(":-implementation.",!IO),
                nl(!IO),
43           printIsaTable(IT,!IO),
44           io.told(!IO)
45          else
46          io.write_string("Can't write ontology.m
```

```
                 ." ,!IO),
47               io.nl(!IO),
48               throw(Result)).
49
50  :-pred write_ontology_graph_concept({string,list
       (string)}::in,
51                                              io::di,
52                                              io::uo) is
                                                   det.
53
54  write_ontology_graph_concept({C,Parents},!IO):-
55          io.write_list(Parents,
56                        "\n",
57                        write_ontology_graph_element
                             (C),
58                        !IO),
59          io.nl(!IO).
60
61  :-pred write_ontology_graph_element(string::in,
62                                              string::in,
63                                              io::di,
64                                              io::uo) is
                                                   det.
65
66  write_ontology_graph_element(C,Parent,!IO):-
67          write_string("\"",!IO),
68          write_string(C,!IO),
69          write_string("\" -> \"",!IO),
70          write_string(Parent,!IO),
71          write_string("\"",!IO).
72
73  :-pred write_ontology_graph(isaDefinition::in,
74                                     io::di,
75                                     io::uo) is det.
76
77  write_ontology_graph(IsaDefinition,!IO):-
```

```
78            io.tell("ontology.dot",Result,!IO),
79            (if Result=ok
80            then
81            write_string("digraph ontology {\n",!IO)
                  ,
82             write_string("graph [rankdir = \"RL
                 \"];\n\n",!IO),
83             write_list(IsaDefinition,
84                        "",
85                        write_ontology_graph_concept
                            ,
86                        !IO),
87             write_string("}\n",!IO),
88             io.told(!IO)
89            else
90            io.write_string("Can't write ontology.
                dot.",!IO),
91             io.nl(!IO),
92             throw(Result)).
93
94  main(!IO):-
95            read_isa_d(IsaDefinition,!IO),
96            createIsaTable(IsaDefinition,IT),
97            write_ontology(IsaDefinition,IT,!IO),
98            write_ontology_graph(IsaDefinition,!IO).
99
100 :-type isaTable == hash_table(string,list(string
      )).
101
102 :-pred isa(isaTable,string,string).
103 :-mode isa(hash_table_ui,in,in) is semidet.
104
105 :-type isaDefinition ==
106           list({string,list(string)}).
107
108 :-pred isa_d(isaDefinition::out,io::di,io::uo)
```

```
    is det.
109
110 isa_d(Result,!IO):-
111         io.read_line_as_string(Line_Option,!IO),
112         (if Line_Option=ok(Line)
113         then
114         Tokens=string.words(char.is_whitespace,
                Line),
115          (if Tokens=[H|T]
116          then
117          isa_d(Rest,!IO),
118           Result=[{H,T}|Rest]
119          else
120          isa_d(Result,!IO))
121         else
122         Result=[]
123         ).
124
125 :-func getParents(isaDefinition,string)
126         =list(string).
127
128 getParents([],_C) = [].
129
130 getParents([{C1,Parents}|IsaDefinition],C) =
131 (if C1=C then Parents else getParents(
    IsaDefinition,C)).
132
133 :-pred getConceptList(isaDefinition::in,
134                         list(string)::out) is det.
135
136 getConceptList([],[]).
137 getConceptList([{C,_Parents}|Rest],[C|Result]):-
138         getConceptList(Rest,Result).
139
140 :-pred write_simpleConcept(isaDefinition::in,
141                                io::di,
```

```
142                                 io::uo) is det.
143
144 write_simpleConcept(IsaDefinition,!IO):-
145          write_string(":-type simpleConcept --->"
              ,!IO),
146          nl(!IO),
147          write_string("\tc_",!IO),
148          getConceptList(IsaDefinition,Concepts),
149          write_list(Concepts,";\n\tc_",
             write_string,!IO),
150          write_string(".",!IO),
151          nl(!IO),
152          nl(!IO).
153
154 :-pred allSuperConcepts(isaDefinition::in,
155                         list(string)::in,
156                         list(string)::out) is
                               det.
157
158 allSuperConcepts(_IsaDefinition,[],[]).
159
160 allSuperConcepts(IsaDefinition,[C|Cs],[C|Result
     ]):-
161          Parents=getParents(IsaDefinition,C),
162          append(Cs,Parents,NewCs),
163          allSuperConcepts(IsaDefinition,
164                          sort_and_remove_dups(
                                 NewCs),
165                          Result).
166 :-pred printIsaTable(isaTable::hash_table_ui,io
     ::di,io::uo) is det.
167
168 printIsaTable(IsaTable,!IO):-
169          IsaTableList=to_assoc_list(IsaTable),
170          printIsaTableList(IsaTableList,!IO).
171
```

```
172 :-pred printIsaTableList(assoc_list(string,list(
        string))::in,io::di,io::uo) is det.
173
174 printIsaTableList([],!IO).
175
176 printIsaTableList([C_Ancestors|Rest],!IO):-
177         fst(C_Ancestors,C),
178         snd(C_Ancestors,Ancestors),
179         write_list(Ancestors," ",writeOneIsa(C)
                ,!IO),
180         nl(!IO),
181         printIsaTableList(Rest,!IO).
182
183 :-pred writeOneIsa(string::in,
184                    string::in,
185                    io::di,
186                    io::uo) is det.
187
188 writeOneIsa(C,Ancestor,!IO):-
189         write_string("isa(c_",!IO),
190         write_string(C,!IO),
191         write_string(",c_",!IO),
192         write_string(Ancestor,!IO),
193         write_string(").",!IO).
194
195 :-pred createIsaTable(isaDefinition::in,
196                       isaTable::hash_table_uo)
                             is det.
197
198 createIsaTable(IsaDefinition,Result):-
199         getConceptList(IsaDefinition,Cs),
200                                  % create an
                                       empty hash
                                       table
201         HashTable = new_default(
                generic_double_hash),
```

```
202             modifyIsaTable(IsaDefinition,HashTable,
                    Cs,Result).
203
204 :-pred modifyIsaTable(isaDefinition,
205                       isaTable,
206                       list(string),
207                       isaTable).
208 :-mode modifyIsaTable(in,
209                       hash_table_di,
210                       in,
211                       hash_table_uo) is det.
212
213 modifyIsaTable(_IsaDefinition,HashTable,[],
        HashTable).
214
215 modifyIsaTable(IsaDefinition,HashTable,[C|Cs],
        Result):-
216             modifyIsaTable(IsaDefinition,HashTable,
                    Cs,NewHashTable),
217             allSuperConcepts(IsaDefinition,[C],
                    CAncestors),
218             Result=det_insert(NewHashTable,
219                               C,
220                               sort_and_remove_dups(
                                      CAncestors)).
221
222
223                                     %:-pred isa_n(
                                            string,string)
                                            .
224                                     %:-mode isa_n(in
                                            ,in) is
                                            semidet.
225                                     %:-mode isa_n(in
                                            ,out) is
                                            nondet.
```

```
226
227                                                   %TODO:
                                                          Inefficient
                                                          implementation
                                                           of isa_n.
228                                                   %isa_n(X,X).
229                                                   %isa_n(X,Y):-
                                                          isa_d(X,Z),
                                                          isa_n(Z,Y).
230
231
232 isa(IT,X,Y):-
233         XAncestors = IT^elem(X),
234         member(Y,XAncestors).
235
236 :-pred write_preamble(io::di,io::uo) is det.
237
238 write_preamble(!IO):-
239         write_string(":-module␣ontology.",!IO),
              nl(!IO),
240         nl(!IO),
241         write_string(":-interface.",!IO),nl(!IO)
              ,
242         nl(!IO),
243         write_string(":-pred␣isa(simpleConcept,
              simpleConcept).",!IO),
244         nl(!IO),
245         write_string(":-mode␣isa(in,in)␣is␣
              semidet.",!IO),
246         nl(!IO),
247         write_string(":-mode␣isa(in,out)␣is␣
              nondet.",!IO),
248         nl(!IO),
249         nl(!IO).
```

## A.15 File html.m

The Mercury module.

```
1  :-module html.
2
3  :-interface.
4
5  :-import_module
6          string,
7          list,
8          io,
9          int,
10         wn,
11         obo.
12
13 %%Positioned string. Contains the begin index
       and end
14 %%index of a word, in STL convention.
15 :-type pstring --->
16         pstring(int,int,string).
17
18 :-pred read_sentences(
19         list(list(pstring))::out,
20         io::di,
21         io::uo) is det.
22
23 :-pred tag_and_write_sentences(
24          wn::in,
25          obo::in,
26          list(list(pstring))::in,
27          io::di,
28          io::uo) is det.
29
30 :-implementation.
31
```

```
32 :-import_module
33         char,
34         aux_io,
35         exception,
36         lexicon,
37         solutions.
38
39 read_sentences(Result,!IO):-
40         search_for_body(0,
41                         '_','_','_','_','_',
42                         Next_Position,
43                         !IO),
44         search_for_sentence(Next_Position,
45                             [[[]]],
46                             Inversed_Sentences,
47                             !IO),
48         map(make_sentence,
49             Inversed_Sentences,
50             Sentences),
51         list.filter(list.is_not_empty,
52                     Sentences,
53                     Result).
54
55 %%Searches for whole <body> tag.
56 :-pred search_for_body(int::in,
57                        char::in,
58                        char::in,
59                        char::in,
60                        char::in,
61                        char::in,
62                        int::out,
63                        io::di,io::uo) is det.
64
65 search_for_body(Position,
66                 A,B,C,D,E,
67                 Next_Position,
```

```
68                 !IO):-
69          (if unify([A,B,C,D,E],['<','b','o','d','
               y'])
70          then search_for_tag_end(Position,
71                                  Next_Position,
72                                  ['b','o','d','y'
                                       ],
73                                  _,
74                                  !IO)
75          else (io.read_char(F_Option,!IO),
76                search_for_body_option(Position,
77                                       A,B,C,D,E,
78                                       F_Option,
79                                       Next_Position
                                           ,
80                                       !IO))).
81
82 :-pred search_for_body_option(int::in,
83                               char::in,
84                               char::in,
85                               char::in,
86                               char::in,
87                               char::in,
88                               io.result(char)::
                                    in,
89                               int::out,
90                               io::di,io::uo) is
                                    det.
91
92 search_for_body_option(Position,
93                        _A,B,C,D,E,
94                        ok(F),
95                        Next_Position,
96                        !IO):-
97          char.to_lower(F,Lower_F),
98          search_for_body(int.plus(Position,1),
```

```
99                                 B,C,D,E,Lower_F,
100                                 Next_Position,
101                                 !IO).
102
103 search_for_body_option(_Position,
104                               _A,_B,_C,_D,_E,
105                               error(Error),
106                               _Next_Position,
107                               !IO):-
108         throw(Error).
109
110 search_for_body_option(Position,
111                               _A,_B,_C,_D,_E,
112                               eof,
113                               Position,
114                               !IO).
115
116 :-pred search_for_char(int::in,
117                               char::in,
118                               int::out,
119                               io::di,
120                               io::uo) is det.
121
122 search_for_char(Position,
123                    C,
124                    Next_Position,
125                    !IO):-
126         io.read_char(C_Option,!IO),
127         search_for_char_option(Position,
128                                    C,
129                                    C_Option,
130                                    Next_Position,
131                                    !IO).
132
133 :-pred search_for_char_option(int::in,
134                                      char::in,
```

```
135                                         io.result(char)::
                                               in,
136                                         int::out,
137                                         io::di,
138                                         io::uo) is det.
139
140   search_for_char_option(Position,
141                          C,
142                          ok(D),
143                          Next_Position,
144                          !IO):-
145           N=int.plus(Position,1),
146           (if unify(C,D)
147           then Next_Position=N
148           else if unify(D,'<')
149           then (search_for_tag_end(N,N1,[],_,!IO),
150                   search_for_char(N1,C,Next_Position
                        ,!IO))
151           else search_for_char(N,C,Next_Position,!
                 IO)
152           ).
153
154   search_for_char_option(_Position,
155                          _C,
156                          error(E),
157                          _Next_Position,
158                          !IO):-
159           throw(E).
160
161   search_for_char_option(Position,
162                          _C,
163                          eof,
164                          Position,
165                          !IO).
166
167   :-pred search_for_tag_end(int::in,
```

```
168                                 int::out,
169                                 list(char)::in, %
                                       Inside tag so far
170                                 list(char)::out, %All
                                       inside tag
171                                 io::di,
172                                 io::uo) is det.
173
174 search_for_tag_end(P,
175                       Next,
176                       Contents_So_Far,
177                       Contents,
178                       !IO):-
179         io.read_char(C_Option,!IO),
180         search_for_tag_end_option(P,
181                                        Next,
182                                        C_Option,
183                                        Contents_So_Far
                                           ,
184                                        Contents,
185                                        !IO).
186
187 :-pred search_for_tag_end_option(int::in,
188                                        int::out,
189                                        io.result(char)
                                           ::in,
190                                        list(char)::in,
191                                        list(char)::out
                                           ,
192                                        io::di,
193                                        io::uo) is det.
194
195 search_for_tag_end_option(P,
196                                 Next,
197                                 ok(C),
198                                 Contents_So_Far,
```

```
199                                     Contents,
200                                     !IO):-
201         N=int.plus(P,1),
202         (if unify(C,'>')
203         then (Next=N,
204                 Contents=Contents_So_Far)
205         else search_for_tag_end(N,
206                                         Next,
207                                         [C|
                                                Contents_So_Far
                                                ],
208                                         Contents,
209                                         !IO)).
210
211 search_for_tag_end_option(_P,
212                                     _Next,
213                                     error(E),
214                                     _Contents_So_Far,
215                                     _Contents,
216                                     !IO):-
217         throw(E).
218
219 search_for_tag_end_option(P,P,eof,Contents,
        Contents,!IO).
220
221 %%Positioned character - position and the
        character:
222 :-type pchar --->
223         pchar(int,char).
224
225 :-type inv_word
226         == list(pchar).
227
228 :-inst inv_word_inst
229         == ground.
230
```

```
231 :-type inv_sentence
232         == list(inv_word).
233
234 :-inst inv_sentence_inst
235         == bound([inv_word_inst|ground]).
236
237 :-type inv_sentences
238         == list(inv_sentence).
239
240 :-inst inv_sentences_inst
241         == bound([inv_sentence_inst|ground]).
242
243 %% The first argument is the list of sentences
       found
244 %% so far. Each sentence is a list of words.
       Each
245 %% word is a list of characters. All of them are
246 %% in reversed order.
247 %% It's an iterative predicate (tail-recursive).
248 :-pred search_for_sentence(int::in,
249                            inv_sentences::in(
                                  inv_sentences_inst)
                                  ,
250                            inv_sentences::out(
                                  inv_sentences_inst)
                                  ,
251                            io::di,
252                            io::uo) is det.
253
254 search_for_sentence(P,
255                     Sentences,
256                     Result,
257                     !IO):-
258         io.read_char(C_Option,!IO),
259         search_for_sentence_option(P,
260                                    Sentences,
```

```
261                                                        C_Option ,
262                                                        Result ,
263                                                        !IO) .
264
265 %%:-pred is_sentence_ender(char::in) is semidet.
266
267 %%is_sentence_ender('.').
268 %%is_sentence_ender('!').
269 %%is_sentence_ender('?').
270 %%is_sentence_ender(':').
271
272
273 :-pred make_sentence(list(list(pchar))::in,
274                      list(pstring)::out) is det.
275
276 make_sentence(Internal,S):-
277         list.reverse(Internal,Internal_Reversed)
            ,
278         list.filter(list.is_not_empty,
279                     Internal_Reversed,
280                     Filtered),
281         list.map(make_word,
282                  Filtered,
283                  S).
284
285 :-pred make_word(list(pchar)::in,
286                  pstring::out) is det.
287
288 make_word([],_):-
289         throw("Empty list given to make_word.").
290
291 make_word([pchar(P,A)|As],pstring(Last_P,P,
    Lower_String)):-
292         get_last_position([pchar(P,A)|As],Last_P
            ),
293         list.map(strip_pchar,
```

```
294                     [pchar(P,A)|As],
295                     Chars),
296          string.from_rev_char_list(Chars,String),
297          string.to_lower(String,Lower_String).
298
299 :-pred get_last_position(list(pchar)::in,
300                              int::out) is det.
301
302 get_last_position([],_):-
303          throw("Empty list given to
                get_last_position.").
304
305 get_last_position([pchar(P,_A)],P).
306
307 get_last_position([_X,Y|T],P):-
308          get_last_position([Y|T],P).
309
310 :-pred strip_pchar(pchar::in,char::out) is det.
311
312 strip_pchar(pchar(_P,A),A).
313
314
315 :-pred search_for_sentence_option(int::in,
316                                        inv_sentences
                                              ::in(
                                              inv_sentences_inst
                                              ),
317                                        io.result(char
                                              )::in,
318                                        inv_sentences
                                              ::out,
319                                        io::di,
320                                        io::uo) is det
                                              .
321
322 search_for_sentence_option(P,[[W|Ws]|Sentences],
```

```
323                                     ok(C),
324                                     Result,
325                                     !IO):-
326             Next=int.plus(P,1),
327             (if char.is_whitespace(C)
328             then search_for_sentence(Next,
329                                         [[[],W|Ws]|
                                                Sentences],
330                                         Result,
331                                         !IO)
332             else if char.is_alpha(C)
333             then search_for_sentence(Next,
334                                         [[[pchar(P,C)|W
                                                ]|Ws]|
                                                Sentences],
335                                         Result,
336                                         !IO)
337             else if unify(C, '<')
338             then (search_for_tag_end(Next,
339                                         New_Next,
340                                         [],
341                                         Tag_Contents,
342                                         !IO),
343                      (if breakable_tag(Tag_Contents)
344                      then
345                      search_for_sentence(New_Next,
346                                            [[[]],[W|Ws]|
                                                 Sentences],
347                                            Result,
348                                            !IO)
349                      else
350                      search_for_sentence(New_Next,
351                                            [[W|Ws]|
                                                 Sentences],
352                                            Result,
353                                            !IO)
```

```
354                 )
355                )
356          else if unify(C, '(')
357          then (search_for_char(Next, ')', New_Next
                ,!IO),
358                search_for_sentence(New_Next,
359                                    [[W|Ws]|
                                          Sentences],
360                                    Result,
361                                    !IO))
362          else (
363               %%End of sentence char.
364               search_for_sentence(Next,
365                                    [[[]],[W|Ws]|
                                          Sentences],
366                                    Result,
367                                    !IO)
368               )
369          ).
370
371 search_for_sentence_option(_P,_S,error(E),
        _Result,!IO):-
372          throw(E).
373
374 search_for_sentence_option(_P,
375                            Sentences,
376                            eof,
377                            Sentences,
378                            !IO).
379
380 tag_and_write_sentences(WN,
381                         OBO,
382                         Sentences,
383                         !IO):-
384          io.write_string("<html><body>\n",!IO),
385          write_list(Sentences,
```

```
386                             "<span␣style=\"background-
                                    color:␣green;␣width:␣1em
                                    ;\"> </span>\n",
387                             tag_and_write_sentence(WN,OBO
                                    ),
388                             !IO),
389         io.write_string("</body></html>\n",!IO).
390
391 :-pred tag_and_write_sentence(wn::in,
392                               obo::in,
393                               list(pstring)::in(
                                      non_empty_list),
394                               io::di,
395                               io::uo) is det.
396
397 tag_and_write_sentence(WN,
398                        OBO,
399                        [pstring(P0,P1,H)|T],
400                        !IO):-
401         string.capitalize_first(H,H1),
402         io.write_list([pstring(P0,P1,H1)|T],
403                       "␣",
404                       tag_and_write_word(WN,OBO)
                          ,
405                       !IO),
406         io.write_string(".",!IO).
407
408 :-pred tag_and_write_word(wn::in,
409                           obo::in,
410                           pstring::in,
411                           io::di,
412                           io::uo) is det.
413
414
415 tag_and_write_word(WN,
416                    OBO,
```

```
417                        pstring(P0,P1,W),
418                        !IO):-
419          solutions(pred(P_O_S::out) is nondet:-
420                    lexicon.getLexicalInfo(WN,
421                                                     pstring(
                                                            P0,P1,
                                                           W),
422                                                     word(_,
423                                                          P_O_S
                                                            ,
424                                                          _))
                                                            ,
425                   P_O_Ss),
426          (
427           P_O_Ss=[],
428           io.write_string("<span␣style=\"border:␣
                 medium␣red␣solid;\">",!IO),
429           (if obo.label(OBO,W,_)
430           then io.write_string("<b>",!IO)
431           else true),
432           io.write_string(W,!IO),
433           (if obo.label(OBO,W,_)
434           then io.write_string("</b>",!IO)
435           else true),
436           io.write_string("</span>",!IO)
437          ;
438           P_O_Ss=[_|_],
439           (if obo.label(OBO,W,_)
440           then io.write_string("<b>",!IO)
441           else true),
442           io.write_string(W,!IO),
443           (if obo.label(OBO,W,_)
444           then io.write_string("</b>",!IO)
445           else true),
446           io.write_string("[",!IO),
447           io.write_list(P_O_Ss,
```

```
448                             ",",
449                             io.write,
450                             !IO),
451           io.write_string("]",!IO)
452          ).
453
454 :- pred breakable_tag(list(char)::in) is semidet.
455
456 breakable_tag(Cs):-
457          string.from_rev_char_list(Cs,Contents),
458          string.to_lower(Contents,Contents_Lower)
             ,
459          list.member(Prefix,
460                        [
461                         "h1","/h1",
462                         "h2","/h2",
463                         "h3","/h3",
464                         "h4","/h4",
465                         "h5","/h5",
466                         "h6","/h6",
467                         "hr","/hr",
468                         "blockquote","/blockquote",
469                         "button","/button",
470                         "caption","/caption",
471                         "center","/center",
472                         "code","/code",
473                         "col","/col",
474                         "colgroup","/colgroup",
475                         "dd","/dd",
476                         "del","/del",
477                         "dir","/dir",
478                         "dfn","/dfn",
479                         "dl","/dl",
480                         "dt","/dt",
481                         "form","/form",
482                         "frame","/frame",
```

```
483                             "frameset","/frameset",
484                             "iframe","/iframe",
485                             "input","/input",
486                             "legend","/legend",
487                             "li","/li",
488                             "link","/link",
489                             "menu","/menu",
490                             "meta","/meta",
491                             "noframes","/noframes",
492                             "noscript","/noscript",
493                             "ol","/ol",
494                             "optgroup","/optgroup",
495                             "option","/option",
496                             "p","/p",
497                             "pre","/pre",
498                             "samp","/samp",
499                             "select","/select",
500                             "table","/table",
501                             "tbody","/tbody",
502                             "td","/td",
503                             "textarea","/textarea",
504                             "tfoot","/tfoot",
505                             "th","/th",
506                             "thead","/thead",
507                             "title","/title",
508                             "tr","/tr",
509                             "ul","/ul",
510                             "xmp","/xmp"
511                             ]),
512            string.prefix(Prefix,Contents_Lower).
```

## A.16 File parse.m

A deprecated parse Mercury module. It is not used in the final implementation of the system, though it might be of some interest.

```
1  % :-pred parseNP(list(word)::in, element::out) is
       nondet.
2
3  % parseNP([noun(N)], np([w(N)])).
4
5  % parseNP([determiner(_)|NP],NPP):-
6  %         parseNP(NP,NPP).
7
8  % parseNP([noun(N)|NP], np([w(N),NPP])):-
9  %         parseNP(NP,NPP).
10
11 % parseNP(NP, np(RES)):-
12 %       append([NP1H|NP1T],[PPH|PPT],NP),
13 %       parseNP([NP1H|NP1T], np(NP1)),
14 %       parsePP([PPH|PPT],PP),
15 %       append(NP1,[PP],RES).
16
17 % :-pred parseS(list(word)::in, element::out) is
      nondet.
18
19 % parseS(S, s([NP,VP])):-
20 %         append([NPH|NPT],[VPH|VPT],S),
21 %         parseNP([NPH|NPT],NP),
22 %         parseVP([VPH|VPT],VP).
23
24 % :-pred parseVP(list(word)::in, element::out) is
       nondet.
25
26 % parseVP([v(V)|PPS], vp([w(V)|PPPS])):-
27 %         parsePPS(PPS,PPPS).
28
```

```
29 % parseVP([v(V)|REST],vp([w(V),PNP|PPPs])):-
30 %         append(PPs,[NPH|NPT],REST),
31 %         parsePPS(PPs,PPPs),
32 %         parseNP([NPH|NPT],PNP).
33
34 % :-pred parsePPS(list(word)::in, list(element)::
      out) is nondet.
35
36 % parsePPS([],[]).
37 % parsePPS(PPS,[PP|PPPST]):-
38 %         append([PP1H|PP1T],REST,PPS),
39 %         parsePP([PP1H|PP1T],PP),
40 %         parsePPS(REST,PPPST).
41
42 % :-pred parsePP(list(word)::in, element::out) is
      nondet.
43
44 % parsePP([preposition(P)|NP],pp([w(P),PNP])):-
45 %         parseNP(NP,PNP).
46
47 :-pred sent(list(string)::out) is det.
48
49 sent(["john",
50       "swims",
51       "in",
52       "the",
53       "pool",
54       "in",
55       "winter"]).
56
57 sent(["insulin",
58       "forces",
59       "storage",
60       "of",
61       "glucose",
62       "in",
```

```
63          "liver",
64          "cells"]).
```

## A.17 File compile.sh

A simple script for compiling the Earley parser.

```
1  #!/bin/bash
2
3  mosmlc -c -structure Earley.sig Earley.sml
```

## A.18 File types.sml

An example illustrating how a simple generative ontology could be integrated into the SML typesystem.

```
1  datatype Vitamin = VitaminA|VitaminB|VitaminC
2
3  datatype STUFF = STUFF_V of Vitamin
4
5  datatype PHYSOBJ = PHYSOBJ
6
7  datatype CONCR = CONCR_S of STUFF | CONCR_P of
       PHYSOBJ
8
9  datatype Lack = Lack
10
11 datatype SYMPTOM = Symptom
12
13 datatype OCCUR
14   = OCCUR_S of STATE
```

```
15     | OCCUR_WRT of OCCUR * STUFF
16
17  and STATE
18     = STATE_L of Lack
19     | STATE_S of SYMPTOM
20     | STATE_CBY of STATE * OCCUR
21
22  datatype UNIV = UNIV_C of CONCR | UNIV_O of
       OCCUR
23
24
25  val a:STATE
26     = STATE_CBY(STATE_S(Symptom),
27                 OCCUR_WRT(OCCUR_S(STATE_L(Lack)),
28                           STUFF_V(VitaminB)))
```

## A.19 File analyze.sml

A very simple ontological analyzer implemented as an experiment in SML.

```
1   datatype element
2     = W of string list
3     | S of element list
4     | VP of element list
5     | NP of element list
6     | PP of element list
7
8   datatype role
9     = TMP (* temporal aspects (generic role) *)
10    | LOC (* location, position *)
11    | PRP (* purpose, function *)
12    | WRT (* with respect to *)
```

```
13     | CHR (* characteristic (property ascription)
        *)
14     | CUM (* cum (i.e., with accompanying) *)
15     | BMO (* by means of, instrument, via *)
16     | CBY (* caused by *)
17     | CAU (* causes *)
18     | CMP (* comprising, has part *)
19     | POF (* part of *)
20     | AGT (* agent of act or process *)
21     | PNT (* patient of act or process *)
22     | SRC (* source of act or process *)
23     | RST (* result of act or process *)
24     | DST (* destination of moving process *)
25
26 datatype concept
27    = continuant of string list
28    | occurrent of string list
29    | entity of string list
30    | cc of concept * ((role * concept) list)
31
32 exception tce of string
33
34 fun ntc ["man"] = continuant ["man"]
35    | ntc ["fish"] = continuant ["fish"]
36    | ntc ["trip"] = occurrent ["trip"]
37    | ntc ["winter"] = occurrent ["winter"]
38    | ntc ["lake"] = continuant ["lake"]
39    | ntc ["John"] = continuant ["John"]
40    | ntc ["pool"] = continuant ["pool"]
41
42 fun prune words =
43        let
44            val useless = ["a","the",
45                           "my","mine",
46                           "your","yours",
47                           "his","her","its",
```

```
48                              "our","ours",
49                              "their","theirs"]
50          fun isUseless word =
51              let val lword = String.map Char.
                  toLower word
52              in
53                  List.exists (fn a => a=lword)
                      useless
54              end
55      in
56          List.filter (fn word => not (isUseless
              word)) words
57      end
58
59  fun nominalize "catches" = "catching"
60    | nominalize "catched" = "catching"
61    | nominalize "swims" = "swimming"
62    | nominalize "swimmed" = "swimming"
63
64  val example1
65    = S [NP [W ["A","man"]],
66         VP [W ["catches"],
67             NP [W ["a","fish"]],
68             PP [W ["on"],
69                 NP [W ["his","trip"]]],
70             PP [W ["to"],
71                 NP [W ["the","lake"]]]
72             ]
73         ]
74
75  val john1
76    = S [NP [W ["John"]],
77         VP [W ["swims"],
78             PP [W ["in"],
79                 NP [W ["the","pool"]]],
80             PP [W ["in"],
```

```
81                         NP [W ["winter"]]]
82                    ]
83             ]
84
85  val john2
86     = S [NP [W ["John"]],
87             VP [W ["swims"],
88                    PP [W ["in"],
89                         NP [NP [W ["the","pool"]],
90                                PP [W ["in"],
91                                       NP [W ["winter"]]
92                                       ]
93                                ]
94                         ]
95                    ]
96             ]
97
98  fun pm "on" = [TMP,LOC]
99    | pm "to" = [DST]
100   | pm "in" = [TMP,LOC]
101
102 (* lcp - list cross product *)
103 fun lcp f l1 l2 =
104       let
105            fun combine e1 l2 = map (fn e2 => f (e1,
                  e2)) l2
106       in
107            foldl (fn (e1,r1) => r1 @ (combine e1 l2
                  )) [] l1
108       end
109
110 fun nd_path [] = []
111   | nd_path [l] = map (fn a => [a]) l
112   | nd_path (l::ls) =
113       let
114            val paths = nd_path ls
```

```
115        in
116            lcp (fn (e,p) => e::p) l paths
117        end
118
119  exception add_AGT_match
120
121  fun add_AGT (agt,cc(c,rs)) = cc(c,(AGT,agt)::rs)
122    | add_AGT _ = raise add_AGT_match
123
124  exception tc_match
125  exception r_match
126
127  (* tc : element -> concept list *)
128  fun tc (S [np as NP _,vp as VP _])
129    = let
130          val npcs = tc np
131          val vpcs = tc vp
132        in
133          lcp add_AGT npcs vpcs
134        end
135
136    | tc (NP [W w]) = [ntc (prune w)]
137
138    | tc (NP((np as NP _)::pps)) =
139        let
140            val npcs = tc np
141            val pppaths = nd_path (map r pps)
142        in
143            lcp (fn (e1,e2) => cc(e1,e2)) npcs
               pppaths
144        end
145
146    | tc (VP((W [v])::(np as NP _)::pps)) =
147        let
148            val vc = occurrent [nominalize v]
149            val npcs = tc np
```

```
150             val pppaths = nd_path (map r pps)
151         in
152             lcp (fn (npc,pppath) => cc(vc,(PNT,npc)
                  ::pppath))
153               npcs pppaths
154         end
155
156     | tc (VP((W [v])::pps)) =
157         let
158             val vc = occurrent [nominalize v]
159             val pppaths = nd_path (map r pps)
160         in
161             map (fn pppath => cc(vc,pppath)) pppaths
162         end
163     | tc _ = raise tc_match
164
165 and r (PP [W [p],np as NP _]) =
166         let
167             val meanings = pm p
168             val concepts = tc np
169         in
170             lcp (fn (e1,e2) => (e1,e2)) meanings
                  concepts
171         end
172     | r _ = raise r_match
173
174
175 val c1 = tc example1
176 val cjohn1 = tc john1
177 val cjohn2 = tc john2
```

# A.20 File `Earley.sig`

The signature of the SML Earley parser implemented.

```
1  signature Earley =
2  sig
3
4      (* Types of terminals and nonterminals*)
5
6      type t = char
7      type n = string
8
9      (* Terminals and nonterminals are symbols *)
10
11     type symbol
12
13     val t : t -> symbol
14     val n : n -> symbol
15
16     (*  [recognize rules  root sequence]  will
          return  true if
17     sequence  belongs  to  language  described
          by  rules  and
18     root. rules is  the list of grammar rules
          and root is the
19     starting symbol.*)
20
21     val recognize : (n*symbol list) list -> n ->
            t list -> bool
22
23 end
```

## A.21 File Earley.sml

The implementation of the SML Earley parser implemented.

```
1  structure Earley :> Earley =
2  struct
3
4  (* User will use this simple way of rules
       specification *)
5
6  type t = char
7  type n = string
8
9  (* And below are internal types *)
10
11 (* We need to extend set of terminal symbols
       with terminator *)
12 datatype terminal
13   = is of t (* any input symbol *)
14   | terminator (* special terminator symbol *)
15
16 datatype symbol
17   = nter of n (* nonterminal, name of the
         syntactic class *)
18   | ter of terminal (* terminal symbol *)
19
20
21 fun t t = ter (is t)
22 fun n n = nter n
23
24 datatype lhsSymbol
25   = lhs of n (* nonterminal, name of the
         syntactic class *)
26   | phi (* special symbol used in starting set
         *)
27
```

```
28 type rules
29   = (n, symbol list list) Polyhash.hash_table
30
31 exception notFound
32
33 datatype state = state of
34            lhsSymbol (* lhs of the rule*)
35            * (symbol list) (* already parsed
                 symbols *)
36            * (symbol list) (* not yet parsed
                 symbols *)
37            * terminal (* one look-ahead symbol *)
38            * int (* where parsing started *)
39
40 fun makeStartState n
41   = state (phi,
42             [],
43             [nter n],
44             terminator,
45             0)
46
47 (* type input = terminal Array.array *)
48
49 fun input ts
50   = Array.fromList ((List.map is ts)@[terminator
         ])
51
52 (* We represent set as map from set elements to
      unit *)
53 type stateSet = (state,unit) Polyhash.hash_table
54
55 fun createEmptyStateSet () =
56     let
57         val emptyStateSet : stateSet
58           = Polyhash.mkPolyTable (10,notFound)
59     in
```

```
60            emptyStateSet
61        end
62
63  exception internalError
64
65  fun createStateSetArray size startState =
66      let
67          val a = Array.array (size,
              createEmptyStateSet ())
68      in
69          if size < 1
70          then raise internalError
71          else (Array.update a 0 startState ; a)
72      end
73
74  fun predictor (nonterminal,lookAhead,rules,
      started) =
75      let
76          fun aux (nonterminal,[],lookAhead,
              started)
77            = []
78            | aux (nonterminal,rhs::rhss,lookAhead
                ,started)
79              = state (lhs nonterminal,[],rhs,
                  lookAhead,started)
80                :: aux (nonterminal,rhss,lookAhead
                    ,started)
81          val alternatives = Polyhash.find rules
              nonterminal
82      in
83          aux (nonterminal,alternatives,lookAhead,
              started)
84      end
85
86  (* isSubstring (l,a,p) checks if list l is a
      substring of
```

```
87  array a starting at position p of the array.*)
88
89  (* fun isSubstring ([],_,_) = true
90     | isSubstring (terminator::rest,input,position
        )
91       = position >= Array.length input
92     | isSubstring ((l terminal)::rest,input,
        position)
93       = Array.sub (input,position) = terminal
94         andalso isSubstring (rest,input,position
            +1)*)
95
96  fun completer ([],_) = []
97     | completer ((state (lhs1,
98                          beforeDot,
99                          (nter nonterminal)::
                              afterDot,
100                         lookAhead,
101                         started))::states,
102                lhs2)
103      = if nonterminal = lhs2
104        then
105            (state (lhs1,
106                    beforeDot@[nter nonterminal],
107                    afterDot,
108                    lookAhead,
109                    started)):: (completer (states
                        ,lhs2))
110        else
111            completer (states,lhs2)
112    | completer ((state (_,
113                         _,
114                         (ter terminal)::_,
115                         _,
116                         _))::states,
117                lhs2)
```

```
118        = completer (states ,lhs2)
119      | completer ((state (_,
120                           _,
121                           [] ,
122                           _,
123                           _)) :: states ,
124                  lhs2)
125        = completer (states ,lhs2)
126
127   (*  expand returns pair  of lists ,  where first
          list contains
128   states for this position , second for the next
        position . *)
129
130   exception whatToDo (* Get rid of this TODO*)
131
132   fun expand (position ,
133               state (_,
134                      _,
135                      (nter nonterminal)::(ter
                           terminal)::_,
136                      _,
137                      _),
138               _,
139               rules ,
140               input)
141     = (predictor(nonterminal ,terminal ,rules ,
           position) ,[])
142
143     | expand (position ,
144               state (_,
145                      _,
146                      (nter nonterminal1)::(nter
                           nonterminal2)::_,
147                      _,
148                      _),
```

```
149                _,
150                rules,
151                input)
152        = raise whatToDo
153
154    | expand (position,
155                state (_,
156                       _,
157                       [nter nonterminal],
158                       lookAhead,
159                       _),
160                _,
161                rules,
162                input)
163        = (predictor(nonterminal,lookAhead,rules,
            position),[])
164
165    | expand (position,
166                state (left,
167                       beforeDot,
168                       (ter terminal)::afterDot,
169                       lookAhead,
170                       started),
171                _,
172                _,
173                input)
174        (* scanner *)
175        = if Array.sub (input,position) = terminal
176          then
177              ([],[state(left,
178                         beforeDot@[ter terminal],
179                         afterDot,
180                         lookAhead,
181                         started)])
182          else
183              ([],[])
```

```
184
185    | expand (position,
186              state (phi,
187                     beforeDot,
188                     [],
189                     lookAhead,
190                     started),
191              stateSetArray,
192              rules,
193              input)
194      = raise whatToDo
195
196    | expand (position,
197              state (lhs left,
198                     beforeDot,
199                     [],
200                     lookAhead,
201                     started),
202              stateSetArray,
203              rules,
204              input)
205      (* completer *)
206      = if Array.sub (input,position) = lookAhead
207        then
208            (completer (List.map (fn (a,b) => a) (
                 Polyhash.listItems (Array.sub (
                 stateSetArray,started))),
209                         left),
210              [])
211        else
212            ([],[])
213
214 fun rules listRules =
215     let
216         val rulesMap : rules
217           = Polyhash.mkPolyTable((length
```

```
                 listRules) div 2,
218                                     notFound)
219         fun aux ((left,right)::listRules,
                rulesMap)
220           = (case Polyhash.peek rulesMap left of
221                  SOME rights => Polyhash.insert
                         rulesMap (left,right::rights)
222                | NONE => Polyhash.insert
                       rulesMap (left,[right])
223                ;
224                aux (listRules,rulesMap))
225           | aux ([],rulesMap) = rulesMap
226     in
227         aux (listRules,rulesMap)
228     end
229
230 fun for (from,to,rulesMap,stateSetArray)
231   = if to <= from
232
233 fun recognize rulesList root ts =
234     let
235         val rulesMap
236           = rules rulesList insert
237         val stateSetAray
238           = createStateSetArray (length ts + 2)
               (makeStartState root)
239         val input
240           = input ts
241         val numberOfStates
242           = length ts + 2
243     in
244         for (0,numberOfStates,rulesMap,
                stateSetArray)
245     end
246
247
```

```
248 (***************************)
249 (**   TESTING              **)
250 (***************************)
251
252 (* User probably wants to specify rules in this
       way: *)
253
254 val ae
255   = [("E", [n "T"]),
256      ("E", [n "E",t #"+",n "T"]),
257      ("T", [n "P"]),
258      ("T", [n "T",t #"*",n "P"]),
259      ("P", [t #"a"])
260      ]
261
262 val res
263   = recognize ae "E" (explode "a+a*a")
264
265 end
```

## A.22 File Insulin.txt

The text is an excerpt from the Wikipedia article on insulin. The content is protected by the GNU Free Documentation License.

```
1 The actions of insulin on the global human
     metabolism level include:
2
3     * Control of cellular intake of certain
          substances, most prominently
4         glucose </wiki/Glucose> in muscle and
             adipose tissue (about ? of
5         body cells).
```

```
6      * Increase of DNA replication </wiki/
         DNA_replication> and protein
7        synthesis </wiki/Protein_synthesis> via
           control of amino acid uptake.
8      * Modification of the activity of numerous
         enzymes </wiki/Enzymes>
9        (allosteric effect </wiki/Allostery>).
10
11 The actions of insulin on cells include:
12
13     * Increased glycogen </wiki/Glycogen>
         synthesis - insulin forces
14       storage of glucose in liver (and muscle)
           cells in the form of
15       glycogen; lowered levels of insulin cause
           liver cells to convert
16       glycogen to glucose and excrete it into
           the blood. This is the
17       clinical action of insulin which is
           directly useful in reducing
18       high blood glucose levels as in diabetes.
19     * Increased fatty acid </wiki/Fatty_acid>
         synthesis - insulin forces
20       fat cells to take in blood lipids which
           are converted to
21       triglycerides </wiki/Triglycerides>; lack
           of insulin causes the
22       reverse.
23     * Increased esterification of fatty acids -
         forces adipose tissue
24       </wiki/Adipose_tissue> to make fats (ie,
           triglycerides) from fatty
25       acid esters; lack of insulin causes the
           reverse.
26     * Decreased proteinolysis </wiki/
         Proteinolysis> - forces reduction
```

```
27       of protein degradation; lack of insulin
            increases protein degradation.
28     * Decreased lipolysis </wiki/Lipolysis> -
         forces reduction in
29       conversion of fat cell lipid stores into
            blood fatty acids; lack
30       of insulin causes the reverse.
31     * Decreased gluconeogenesis </wiki/
         Gluconeogenesis> - decreases
32       production of glucose from various
            substrates in liver; lack of
33       insulin causes glucose production from
            assorted substrates in the
34       liver and elsewhere.
35     * Increased amino acid uptake - forces cells
          to absorb circulating
36       amino acids; lack of insulin inhibits
            absorption.
37     * Increased potassium uptake - forces cells
         to absorb serum
38       potassium; lack of insulin inhibits
            absorption.
39     * Arterial muscle tone - forces arterial
         wall muscle to relax,
40       increasing blood flow, especially in micro
             arteries; lack of
41       insulin reduces flow by allowing these
            muscles to contract.
```

## A.23 File compile.sh

A simple BASH script for compiling the Link Grammar parsing program.

```
1 #!/bin/bash
2 gcc -ggdb -I/usr/include -o parse parse.c -llink
    -grammar
```

## A.24 File parse.c

The file is an experiment, which purpose is to make use of the Link Grammar API and libraries.

```
1  #include <link-grammar/link-includes.h>
2
3  int main(const int argc,
4           char** argv) {
5
6      Dictionary    dict;
7      Parse_Options opts;
8      Sentence      sent;
9      Linkage       linkage;
10     char *        diagram;
11     int           i, num_linkages;
12
13     CNode* p_cnode;
14     char* tree;
15
16     char * input_string
17       = "Grammar is useless because there is
            nothing to say -- Gertrude Stein.";
18
19     opts  = parse_options_create();
20     dict
21       = dictionary_create("/usr/share/link-
            grammar/en/4.0.dict",
22                           "/usr/share/link-
```

```
                                    grammar/en/4.0.
                                    knowledge",
23                                0,
24                                "/usr/share/link-
                                    grammar/en/4.0.affix
                                    ");
25
26      sent = sentence_create(input_string, dict);
27      num_linkages = sentence_parse(sent, opts);
28
29      if (num_linkages > 0) {
30        linkage = linkage_create(0, sent, opts);
31
32        //p_cnode=linkage_constituent_tree(linkage
            );
33
34        tree=linkage_print_constituent_tree(
            linkage,1);
35        //printf("%s\n",tree);
36        if(tree)
37          string_delete(tree);
38        //linkage_free_constituent_tree(p_cnode);
39
40
41        diagram = linkage_print_diagram(linkage);
42        printf("%s\n",diagram);
43        string_delete(diagram);
44
45        linkage_delete(linkage);
46      }
47
48      sentence_delete(sent);
49      dictionary_delete(dict);
50      parse_options_delete(opts);
51      return 0;
52  }
```

## A.25 File `insulin_OBO_concepts.txt`

The list of all phrases from the Wikipedia insulin paper that correspond to OBO concepts.

```
1  "a␣protein", "a␣steroid", "absent",
2  "absorption", "acetylcholine", "acid",
3  "acids", "actin", "actin␣cytoskeleton",
4  "activation", "active", "acute",
5  "addition", "adenylate", "adenylate
6  cyclase", "adequate", "adipose",
7  "adipose␣tissue", "adrenergic␣agonists",
8  "adult", "age", "aka", "al", "alberta",
9  "aldosterone", "alive", "all", "alpha",
10 "amino", "amino␣acid", "amino␣acid
11 residues", "amino␣acid␣sequence", "amino
12 acid␣uptake", "amino␣acids", "amount",
13 "an", "anatomy", "and", "androgen",
14 "animal", "anovulation", "anp",
15 "arterial␣wall", "as", "at", "atp",
16 "atrium", "attached␣to", "autoimmune",
17 "autoimmunity", "autonomic", "autonomic
18 nervous␣system", "average", "b",
19 "backbone", "bacteria", "basal", "base",
20 "bdnf", "be", "beta", "beta␣cell",
21 "binding", "biology", "blindness",
22 "blood", "blood␣glucose", "blood
23 pressure", "blood␣sugar", "blood
24 volume", "blue", "body", "body␣water",
25 "body␣weight", "bovine", "brain", "brain
26 damage", "breathing", "bulk", "by", "c",
27 "c␣", "c␣peptide", "calcitonin",
28 "calcitriol", "calcium", "calf", "can",
29 "cannula", "carbohydrate", "carbohydrate
30 metabolism", "carbohydrates", "carbon",
31 "carboxypeptidase", "catecholamines",
```

```
32 "catheter", "cck", "cell", "cell
33 membrane", "cell transplant", "central",
34 "central nervous system", "certain",
35 "chaos", "chemistry", "child",
36 "cholecystokinin", "cholesterol",
37 "chronic", "circulation",
38 "circumference", "clear", "cleavage",
39 "clinical treatment", "coffee",
40 "combination", "common", "community",
41 "complex", "composition",
42 "concentration", "conformation",
43 "confusion", "consciousness",
44 "conserved", "content", "continual",
45 "continuous", "control", "conversion",
46 "cortex", "cortisol", "cow", "crh",
47 "crystallography", "cyan", "cyclase",
48 "cytoskeleton", "d", "da", "damage",
49 "data", "death", "decreased",
50 "degenerate", "degradation", "delay",
51 "delivery", "development", "device",
52 "dhea", "diabetes", "diabetes mellitus",
53 "diabetic coma", "diabetic nephropathy",
54 "diabetic neuropathy", "diabetic
55 retinopathy", "diacylglycerol", "did",
56 "digestion", "direction", "disease",
57 "distributed", "disulfide", "dizziness",
58 "dna", "dna replication", "dog", "dogs",
59 "doi", "domain", "dopamine", "down",
60 "drinking", "drug", "drugs", "duct",
61 "ducts", "duration", "e", "e ",
62 "eating", "efficient", "effort",
63 "electricity", "elevated", "end",
64 "endocrine", "endocrine glands",
65 "endocrinology", "endogenous",
66 "endoplasmic reticulum", "energy",
67 "engineered", "enhancer",
```

```
68 "enteroglucagon", "epinephrine", "epo",
69 "equipment", "er", "essential
70 hypertension", "esterification",
71 "estradiol", "estrogen", "et",
72 "example", "exercise", "exocrine gland",
73 "exoprotease", "external",
74 "extracellular", "extreme", "family",
75 "fat", "fat cell", "fatty acid", "fatty
76 acid synthesis", "fatty acids",
77 "figure", "fish", "fisher", "fit",
78 "flies", "flow", "food", "food intake",
79 "form", "fructosamine", "function", "g",
80 "gastrin", "gastrointestinal",
81 "gastrointestinal tract", "gene",
82 "genetic", "genetic engineering",
83 "genetic susceptibility", "gh", "ghrh",
84 "gip", "gland", "glands", "glucagon",
85 "gluconeogenesis", "glucose", "glucose
86 dependent insulinotropic peptide",
87 "glucose transporter", "glut", "glut2",
88 "glycerol", "glycogen", "glycogen
89 synthesis", "glycolysis", "glycosuria",
90 "glycosylated hemoglobin", "gnrh",
91 "goes", "greater than", "green",
92 "growth", "growth factor", "growth
93 hormone", "had", "half life", "hand",
94 "has", "hcg", "he", "heart", "heart
95 attack", "hemoglobin", "high", "high
96 blood glucose levels", "him",
97 "hirsutism", "his", "histidine",
98 "homeostasis", "hormone", "hormones",
99 "horse", "host", "host cell", "human",
100 "humulin", "hydrogen", "hyperglycemia",
101 "hypertension", "hypoglycemia",
102 "hypoglycemic coma", "i", "idea", "igf",
103 "igf 1", "igf1", "ii", "image",
```

```
104 "immune", "immune system", "impaired",
105 "impaired glucose tolerance", "in",
106 "increased", "infection", "inhibin",
107 "inhibited", "injection", "inositol",
108 "ins", "inserted into", "insufficient",
109 "insulin", "insulin dependent", "insulin
110 like growth factor", "insulin receptor",
111 "insulin resistance", "insulin
112 sensitivity", "insulin signaling
113 pathway", "insulinoma", "interaction",
114 "intermediate", "internal",
115 "interrupted", "intestinal absorption",
116 "intestinal mucosa", "introduced into",
117 "inulin", "investigation", "ions",
118 "ip3", "is", "is a", "islet cell",
119 "islets of langerhans", "j", "jet
120 injection", "key", "kidney", "kidney
121 failure", "l", "large", "lead",
122 "learning", "left", "length", "leptin",
123 "less than", "lh", "ligature", "lipid",
124 "lipids", "lipolysis", "lipotropin",
125 "liver", "liver cell", "long", "low",
126 "low blood glucose levels", "lower",
127 "m", "magenta", "male", "mammals",
128 "man", "mass", "mature", "maturity",
129 "measure", "medication", "medicine",
130 "medulla", "melatonin", "membrane",
131 "membrane of endoplasmic reticulum",
132 "metabolic syndrome", "metabolism",
133 "method", "milk", "mode", "molecular
134 weight", "molecule", "molecules",
135 "motif", "motilin", "movement", "msh",
136 "mucosa", "muscle", "muscles", "n", "n
137 terminal", "narrow", "needle",
138 "nematode", "nerve", "nervous system",
139 "neuropathy", "new", "ngf", "nitrogen",
```

```
140 "no", "none", "norepinephrine",
141 "normal", "not", "novolin", "nph
142 insulin", "nt", "nt␣3", "number",
143 "nutrients", "o", "obesity", "old",
144 "open", "operation", "or", "oral",
145 "order", "organ", "oscar", "other",
146 "outside", "ovary", "ox", "oxidation",
147 "oxygen", "oxytocin", "pa", "page",
148 "pancreas", "pancreatic␣duct",
149 "pancreatic␣extract", "part␣of",
150 "patent", "pathway", "pcos", "peptide",
151 "peptide␣hormones", "ph", "pharmacy",
152 "phosphatidyl", "phosphatidyl␣inositol",
153 "phospholipase", "phospholipase␣c",
154 "phospholipid", "physical",
155 "physiology", "pig", "pituitary",
156 "plasma", "plasma␣membrane", "point",
157 "polycystic␣ovary", "polycystic␣ovary
158 syndrome", "polypeptide", "polypeptide
159 hormone", "polypeptides", "pomc",
160 "porous", "portal␣vein",
161 "posttranslational␣modification",
162 "potassium", "potassium␣uptake",
163 "potato", "prediabetes", "present",
164 "pressure", "primary␣structure", "pro",
165 "process", "professor", "progesterone",
166 "proinsulin", "prolactin",
167 "proliferative␣diabetic␣retinopathy",
168 "prominent", "prostaglandin",
169 "protamine", "protein", "protein
170 coding", "protein␣degradation", "protein
171 synthesis", "proteins", "psychiatrist",
172 "pulmonary", "purified", "r",
173 "radioimmunoassay", "range", "rate",
174 "ray", "re", "reaction", "read",
175 "recent", "receptor", "recombinant␣dna",
```

```
176 "red", "reduced", "reduction", "refseq",
177 "regular", "regular␣insulin",
178 "regulation", "relaxin", "renin",
179 "replication", "report", "reproductive",
180 "research", "respiratory", "response
181 to", "reticulum", "right", "risk",
182 "role", "s", "same", "scale", "science",
183 "second", "secretin", "secretion", "see
184 also", "self", "sensitivity",
185 "sequence", "sequence␣of", "serum",
186 "severe", "short", "signal", "signal
187 transduction", "signaling", "signaling
188 pathway", "signalling", "similar␣to",
189 "simple", "single", "sir", "site",
190 "skeletal␣muscle", "skin", "slight",
191 "small", "so", "solution",
192 "somatostatin", "sp", "space",
193 "spatial", "specialist", "species",
194 "speech", "step", "steroid", "steroids",
195 "stimulation", "storage", "strength",
196 "stress␣hormone", "structure", "sugar",
197 "superior", "supply", "survival",
198 "symmetry", "sympathetic", "sympathetic
199 nervous␣system", "syndrome",
200 "synthesis", "synthetic", "t", "tax",
201 "tertiary␣structure", "testing",
202 "testosterone", "this", "thyroid",
203 "thyroid␣hormone", "time", "tissue",
204 "to", "tone", "tract", "transduction",
205 "translocation", "transplant",
206 "transplantation", "transport",
207 "transporter", "treatment", "trh",
208 "triphosphate", "tsh", "type␣2", "type
209 i", "type␣i␣diabetes", "type␣ii", "u",
210 "uk", "unconsciousness", "uniprot",
211 "unit", "university", "unknown", "up",
```

```
212 "urine", "v", "vagus", "vagus nerve",
213 "value", "vasopressin", "vein",
214 "vertebrates", "vip", "voltage",
215 "volume", "w", "waist", "walker",
216 "water", "weight", "weight gain",
217 "weight loss", "white", "whole", "work",
218 "x", "x ray", "x ray crystallography",
219 "x ray diffraction", "yellow", "young",
220 "zinc", "zinc binding", "zinc ions"
```

## A.26 File `FindConcepts.java`

The following file has been used in the very early stage of the project for studying the OBO coverage of biomedical texts. In the recent version this functionality is implemented in Mercury `html` module.

```
1  import java.io.*;
2
3
4  class FindConcepts {
5
6      public static void main(String args[]) throws
           Exception {
7
8          BufferedReader r = new BufferedReader(
               new FileReader("insulin.raw"));
9
10         final String insulin = r.readLine();
11
12         r = new BufferedReader(new FileReader("
               conceptNames.txt"));
13
14         String line;
15         while((line=r.readLine())!=null) {
```

```
16                  if(line.length()==0)
17                      continue;
18                  if(insulin.contains(line))
19                      System.out.println(line);
20              }
21          }
22  }
```

## A.27 File createLists.sh

A BASH script for preliminary processing of WordNet lexical database.

```
1  #!/bin/bash
2
3  cat index.noun|grep '^[^ ][^ ]'|cut -d' ' -f1|
       grep '^[0-9a-zA-Z]*$' >nouns.txt
4  cat index.verb|grep '^[^ ][^ ]'|cut -d' ' -f1|
       grep '^[0-9a-zA-Z]*$' >verbs.txt
5  cat index.adj|grep '^[^ ][^ ]'|cut -d' ' -f1|
       grep '^[0-9a-zA-Z]*$' >adjectives.txt
6  cat index.adv|grep '^[^ ][^ ]'|cut -d' ' -f1|
       grep '^[0-9a-zA-Z]*$' >adverbs.txt
```

## A.28 File hand-grabbed_insulin_concepts.txt

This file has been created by hand in order to illustrate the intended working of the system for some exemplary sentences. The later design of the system was guided by those examples, in order to reply with as close results as possible.

```
1  ———
```

```
2
3 The actions of insulin on the global human
    metabolism level
4 include:
5
6 action[wrt:insulin ,pnt:metabolism[wrt:human,atr:
    global]]
7
8 ———
9
10 Control of cellular intake of certain
    substances, most
11 prominently glucose in muscle and adipose tissue
    .
12
13 control[wrt:intake[atr:cellular ,wrt:substance]]
14
15 glucose[loc:muscle]
16
17 tissue[atr:adipose]
18
19 ———
20
21 Increase of DNA replication and protein
    synthesis via
22 control of amino acid uptake.
23
24 increase[wrt:replication[wrt:DNA] ,wrt:synthesis[
    wrt:protein]]
25
26 control[pnt:uptake[wrt:acid[atr:amino]]]
27
28 ———
29
30 Modification of the activity of numerous enzymes
    .
```

```
31
32 modification[pnt:activity[wrt:enzyme]]
33
34 ———
35
36 The actions of insulin on cells include:
37
38 action[agt:insulin,pnt:cell]
39
40 ———
41
42 Increased glycogen synthesis –
43
44 synthesis[wrt:glycogen,atr:increased]
45
46 ———
47
48 insulin forces storage of glucose in liver cells
       in the form
49 of glycogen;
50
51 forcing[agt:insulin,
52         pnt:storage[wrt:glucose,
53                     loc:cell[wrt:liver],
54                     atr:glycogen]]
55
56 ———
57
58 lowered levels of insulin cause liver cells
       to convert
59 glycogen to glucose and excrete it into the
     blood.
60
61 causing[agt:level[wrt:insulin,atr:lowered],
62         pnt:cell[wrt:liver],
63         rst:conversion[src:glycogen,rst:glucose
```

```
                ],
64          rst:excretion[dst:blood]]
65
66  ———
67
68  This is the clinical action of insulin which
        is directly
69  useful in reducing high blood glucose levels as
        in diabetes.
70
71  action[chr:clinical,agt:insulin]
72
73  reduction[pnt:level[wrt:glucose[wrt:blood],chr:
        high]]
74
75  ———
76
77  Increased fatty acid synthesis -
78
79  synthesis[wrt:acid[chr:fatty],chr:increased]
80
81  increase[pnt:synthesis[wrt:acid[chr:fatty]]
82
83  ———
84
85  insulin forces fat cells to take in blood
        lipids which are
86  converted to triglycerides;
87
88  forcing[agt:insulin,
89          pnt:intake[pnt:lipid[wrt:blood],agt:cell
                [wrt:fat]]]
90
91  conversion[rst:triglycerides]
92
93  ———
```

```
94
95  lack of insulin causes the reverse.
96
97  causing[agt:lack[wrt:insulin],pnt:reverse]
98
99  ———
100
101 Increased esterification of fatty acids -
102
103 esterification[chr:increased,pnt:acid[chr:fatty
      ]]
104
105 ———
106
107 forces adipose tissue to make fats from fatty
      acid esters;
108
109 forcing[pnt:tissue[wrt:adipose],
110          rst:making[rst:fats,src:ester[wrt:acid[
               chr:fatty]]]]
111
112 ———
113
114 lack of insulin causes the reverse.
115
116 causing[agt:lack[wrt:insulin],pnt:reverse]
117
118 ———
119
120 Decreased proteinolysis -
121
122 proteinolysis[chr:decreased]
123
124 ———
125
126 forces reduction of protein degradation;
```

```
127
128 forcing[rst:reduction[wrt:degradation[wrt:
        protein]]]
129
130 ———
131
132 lack of insulin increases protein degradation
133
134 increase[agt:lack[wrt:insulin],pnt:degradation[
        wrt:protein]]
135
136 ———
137
138 Decreased lipolysis –
139
140 lipolysis[chr:decreased]
141
142 ———
143
144 forces reduction in conversion of fat cell lipid
        stores into
145 blood fatty acids;
146
147 forcing[rst:reduction[wrt:conversion[src:store,
148                                       rst:acid]]]
```

## A.29 File `getConceptNames.sh`

The following file has been used in the very early stage of the project for studying the OBO coverage of biomedical texts. In the recent version this functionality is implemented in Mercury `html` module.

```
1 #!/bin/bash
2
```

```
3
4 grep -F 'metadata_db:entity_label(' $* |sed "s
    /^.*[,][␣]*[']//"|sed "s/[']]).*//"|tr '[:
    upper:]' '[:lower:]'
5
6 grep -F 'metadata_db:entity_synonym(' $* |sed "s
    /^.*[,][␣]*[']//"|sed "s/[']]).*//"|tr '[:
    upper:]' '[:lower:]'
```

## A.30 File increase.owl

An exemplary OWL concept definition for the action of increasing.

```
1  <owl:Class rdf:about="&span;Increase">
2    <rdfs:subClassOf rdf:resource="&span;Process"/
       >
3
4    <owl:disjointWith rdf:resource="&span;Decrease
       "/>
5
6    <rdfs:label rdf:datatype="&xsd;string">
       increase</rdfs:label>
7
8    <rdfs:comment rdf:datatype="&xsd;string">
9     Definition: A processual entity that is
       a maximally
10    connected spatio-temporal whole and has
       bona fide
11    beginnings and endings corresponding
       to real
12    discontinuities. The ending is somehow "
       larger" than the
13    beginning.
14   </rdfs:comment>
```

```
15
16   <rdfs:comment rdf:datatype="&xsd;string">
17    Examples: Increased glycogen  synthesis,
        Increase  of DNA
18    replication.
19   </rdfs:comment>
20
21 </owl:Class>
22
23 <owl:ObjectProperty rdf:ID="hasParticipant">
24   <rdfs:label rdf:datatype="&xsd;string">
        has_participant</rdfs:label>
25
26   <rdfs:comment rdf:datatype="&xsd;string">
        HasParticipant is a primitive instance-level
         relation between a process, a continuant,
        and a time at which the continuant
        participates in some way in the process. The
         relation obtains, for example, when this
        particular process of oxygen exchange across
         this particular alveolar membrane
        hasParticipant this particular sample of
        hemoglobin at this particular time.</
        rdfs:comment>
27
28   <rdfs:comment rdf:datatype="&xsd;string">P
        hasParticipant C if and only if: given any
        process p that instantiates P there is some
        continuant c, and some time t, such that: c
        instantiates C at t and c participates in p
        at t</rdfs:comment>
29 </owl:ObjectProperty>
```

Appendix B

# Introduction to lattices

## B.1 Posets

We call a binary relation $R$ a poset (partially ordered set) iff:

$$poset(R) \leftrightarrow reflexive(R) \wedge antisymmetric(R) \wedge transitive(R)$$

As described in [16], from mathematical point of view a poset is a pair $(P, R)$ where $P$ is a set and $R$ is a relation (set of pairs: $R \subseteq P \times P$) such that R is a partial order (it is reflexive, antisymmetric and transitive).

We will however use the notion of a poset and partial order simultaneaously for a relation $\rho$, since we can always obtain such a mathematical view of a poset $(A_\rho, \rho)$ using definitions 3.3 and 3.2.

Let us consider the following example defininig the relation $part_of$:

$$r(cell, part_of, liver)$$
$$r(cell, part_of, heart)$$
$$r(liver, part_of, body)$$
$$r(heart, part_of, body)$$

Using definition 3.3 we can obtain a mathematical view of the set $A_{part_of}$ on which the relation is defined:

$$A_{part_of} = \{cell, liver, heart, body\}$$

Similarly, we can calculate the mathematical view of the relation $part_of$ as a set of pairs using definition 3.2:

$$part_of = \{ \quad (cell, liver), (cell, heart), \\ (liver, body), (heart, body) \quad \}$$

Hence, we can refer to relation $part_of$ as both a partial order and a poset.

It is clear that (verification is left as an exercise):

$$poset(part_of)$$

## B.2 Top and bottom

We can define the top element $T$ (denoted $\top$) for relation $R$ as:

$$r(T, topOf, R) \leftrightarrow \forall X, Y (r(X, R, Y) \rightarrow r(X, R, T) \wedge r(Y, R, T))$$

Similarly, the bottom element $B$ (denoted $\bot$) can be defined as:

$$r(B, bottomOf, R) \leftrightarrow \forall X, Y(r(X, R, Y) \rightarrow r(B, R, X) \wedge r(B, R, Y))$$

As an example for relation *part_of* defined in section B.1, we have:

$$r(body, topOf, part_of) \wedge r(cell, bottomOf, part_of)$$

There are relations $R$ without $\top$ and $\bot$ elements:

$$\exists R(\neg \exists T(r(T, topOf, R)))$$
$$\exists R(\neg \exists B(r(B, bottomOf, R)))$$

## B.3 Upper and lower bounds

We say that elements $X$ and $Y$ in relation $R$ have an upper bound $U$, a least upper bound (supremum) $S$, lower bound $L$, and a greatest lower bound (infimum) $I$ when:

$$\begin{aligned} upperBound(X, Y, R, U) &\leftrightarrow r(X, R, U) \wedge r(Y, R, U) \\ supremum(X, Y, R, S) &\leftrightarrow \forall U(upperBound(X, Y, R, U) \rightarrow r(S, R, U)) \\ lowerBound(X, Y, R, L) &\leftrightarrow r(L, R, X) \wedge r(L, R, Y) \\ infimum(X, Y, R, I) &\leftrightarrow \forall L(lowerBound(X, Y, R, L) \rightarrow r(L, R, I)) \end{aligned}$$

## B.4 Lattice as poset

We say that poset $R$ is a lattice iff:

$$\begin{aligned} lattice(R) \quad \leftrightarrow \quad & \forall X, Y (r(X, R, Y) \rightarrow \exists S (supremum(X, Y, R, S)) \\ & \wedge \exists I (infimum(X, Y, R, I))) \end{aligned}$$

We define *boundedLattice* as:

$$boundedLattice(R) \leftrightarrow lattice(R) \wedge \exists T (r(T, topOf, R)) \wedge \exists B (r(B, bottomOf, R))$$

## B.5 Hasse diagrams

The graphical representation of a poset known as Hasse diagram can be drawn in the following manner (from [16]): if $x \leq y$ then we place $x$ below $y$ and draw a line from $x$ to $y$, except that lines following from reflexivity and transitivity are omitted.

## B.6 *isa* as partial order

Of particular interest is the *isa* relation, which forms a poset when coupled with arbitrary set of classes. We can hence easily represent the taxonomy of our domain using the Hasse diagram.

We could insist that the *isa* relation is a lattice in order to enjoy nice mathematical properties for our classification.

## B.7 Lattices as algebras

We shall call a pair of operations $J, M$ a lattice, when:

$$\begin{aligned} lattice(J, M) \quad \leftrightarrow \quad & indempotent(J) \wedge indempotent(M) \wedge \\ & commutative(J) \wedge commutative(M) \wedge \\ & associative(J) \wedge associative(M) \wedge \\ & absorptive(J, M) \wedge absorptive(M, J) \end{aligned}$$

As defined in [16], from mathematical point of view, lattice is a tripple $(L, J, M)$, where $L$ is a nonempty set, $J$ and $M$ are binary operations defined on $L$, having special properties. In our FOL notation, however, the set $L$ is implicitly defined and can be obtained using definition 3.5.

We usually refer to the two operations as join (denoted $\vee$) and meet (denoted $\wedge$). Symbols $\vee$ and $\wedge$ are reserved for disjunction and conjunction in FOL, hence we shall avoid them in our FOL formulae.

## B.8 Dual nature of lattices

The two definitions below allow us to transform a lattice defined as partial order (section B.4) into lattice defined as an algebra (section B.7) and vice versa. The proof can be found in [16].

$$\begin{aligned} lattice(R) \quad \rightarrow \quad & \forall(supremum(X, Y, R, S) \rightarrow o(X, join, Y, S)) \\ & \wedge\forall(infimum(X, Y, R, I) \rightarrow o(X, meet, Y, I)) \\ & \wedge lattice(join, meet) \\ lattice(J, M) \quad \rightarrow \quad & \forall(o(X, J, Y, X) \rightarrow r(X, \leq, Y)) \wedge lattice(\leq) \end{aligned}$$

## B.9 Atoms

$$
\begin{aligned}
atom(A, R) \quad \leftrightarrow \quad & lattice(R) \wedge \\
& \forall (r(X, R, A) \rightarrow equal(X, A) \vee (equal(X, B) \wedge r(B, bottomOf, R)))
\end{aligned}
$$

Thanks to the algebraic nature of lattices (section B.7), we can use meet and join algebraic operations to specify our domain's classification as a lattice.

# Bibliography

[1] Troels Andreasen and Jørgen Fischer Nilsson. Grammatical specification of domain ontologies. *Data Knowl. Eng.*, 48(2):221–230, 2004.

[2] F. Baader and W. Nutt. Basic description logics, 2003.

[3] Collin F. Baker, Charles J. Fillmore, and John B. Lowe. The berkeley framenet project. In *Proceedings of the 17th international conference on Computational linguistics*, pages 86–90, Morristown, NJ, USA, 1998. Association for Computational Linguistics.

[4] Collin F. Baker and Hiroaki Sato. The framenet data and software. In *ACL '03: Proceedings of the 41st Annual Meeting on Association for Computational Linguistics*, pages 161–164, Morristown, NJ, USA, 2003. Association for Computational Linguistics.

[5] M. Ben-Ari. *Mathematical logic for computer science.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[6] Michael Böttner. Peirce grammar. *Grammars*, 4(1):1–19, 2001.

[7] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30(1–7):107–117, 1998.

[8] Chris Brink, Katarina Britz, and Renate A. Schmidt. *Peirce Algebras*, pages 339–358. 1994.

[9] Aldo Gangemi, Nicola Guarino, Claudio Masolo, and Alessandro Oltramari. Sweetening wordnet with dolce. *AI Mag.*, 24(3):13–24, September 2003.

[10] Peter Gerstl and Simone Pribbenow. Midwinters, end games, and body parts: a classification of part-whole relations. *Int. J. Hum.-Comput. Stud.*, 43(5-6):865–889, 1995.

[11] Asuncion Gomez-Perez, Oscar Corcho, and Mariano Fernandez-Lopez. *Ontological Engineering : with examples from the areas of Knowledge Management, e-Commerce and the Semantic Web. First Edition (Advanced Information and Knowledge Processing)*. Springer, July 2004.

[12] P. A. Jensen and J. F. Nilsson. Ontology-based semantics for prepositions. In *Syntax and Semantics of Prepositions*, Text, Speech and Language Technology, Vol. 29. Springer, 2006.

[13] Dave MacQueen. The standard ml of new jersey website. *http://www.smlnj.org/index.html.*

[14] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, USA, 1990.

[15] J. F. Nilsson. Ontological constitutions for classes and properties. In P. Øhrstrøm H. Schaerfe, P. Hitzler, editor, *14th Int. Conf. on Conceptual Structures, ICCS 2006*, Lecture Notes in Artificial Intelligence LNAI 4068. Springer, 2006.

[16] Jørgen Fischer Nilsson and Nikolaj Oldager. Introduction to orders and lattices. 2002.

[17] Sag, T. Wasow, and E. Bender. *Syntactic Theory: a formal introduction, Second Edition.* 2003.

[18] Barry Smith. Beyond concepts: Ontology as reality representation.

[19] Holger Stenzhorn. Basic formal ontology website. *http://www.ifomis.uni-saarland.de/bfo/.*

[20] Author unknown. The mercury project introduction. *http://www.cs.mu.oz.au/research/mercury/.*

[21] Author unknown. Obo foundry website. *http://obofoundry.org/.*

[22] Author unknown. Wordnet$^{tm}$ file formats. *WordNet 3.0 distribution documentation.*

[23] Christopher A. Welty and David A. Ferrucci. What's in an instance?

[24] Mauhab Zareh. Framenet frequently asked questions. *http://framenet.icsi.berkeley.edu/.*