Eötvös Loránd Tudományegyetem
Bölcsészettudományi Kar

Doktori Disszertáció

Recski Gábor

Számítógépes módszerek a szemantikában

Nyelvtudományi Doktori Iskola
Tolcsvai Nagy Gábor MHAS

Elméleti Nyelvészet Doktori Program
Bánréti Zoltán CSc.

A bizottság tagjai:
Kiefer Ferenc MHAS (elnök)
Rebrus Péter PhD. (titkár)
Vincze Veronika PhD.
Alberti Gábor DSc.
Komlósy András CSc.

Témavezető:
Kornai András DSc.

Budapest, 2016

ADATLAP
a doktori értekezés nyilvánosságra hozatalához

I. A doktori értekezés adatai

A szerző neve: Recski Gábor András..
MTMT-azonosító: 10034572...
A doktori értekezés címe és alcíme: Computational Methods in Semantics........................
DOI-azonosító: 10.15476/ELTE.2016.126...
A doktori iskola neve: Nyelvtudományi Doktori Iskola...
A doktori iskolán belüli doktori program neve: Elméleti Nyelvészet Doktori Program.……
A témavezető neve és tudományos fokozata: Kornai András, DSc. ....
A témavezető munkahelye: MTA Számítástechnikai és Automatizálási Kutatóintézet.........

II. Nyilatkozatok

1. A doktori értekezés szerzőjeként

a) hozzájárulok, hogy a doktori fokozat megszerzését követően a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az ELTE Digitális Intézményi Tudástárban. Felhatalmazom az ELTE BTK Doktori és Tudományszervezési Hivatal ügyintézőjét, Manhercz Mónikát, hogy az értekezést és a téziseket feltöltse az ELTE Digitális Intézményi Tudástárba, és ennek során kitöltse a feltöltéshez szükséges nyilatkozatokat.

b) kérem, hogy a mellékelt kérelemben részletezett szabadalmi, illetőleg oltalmi bejelentés közzétételéig a doktori értekezést ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;

c) kérem, hogy a nemzetbiztonsági okból minősített adatot tartalmazó doktori értekezést a minősítés (dátum)-ig tartó időtartama alatt ne bocsássák nyilvánosságra az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban;

d) kérem, hogy a mű kiadására vonatkozó mellékelt kiadó szerződésre tekintettel a doktori értekezést a könyv megjelenéséig ne bocsássák nyilvánosságra az Egyetemi Könyvtárban, és az ELTE Digitális Intézményi Tudástárban csak a könyv bibliográfiai adatait tegyék közzé. Ha a könyv a fokozatszerzést követőn egy évig nem jelenik meg, hozzájárulok, hogy a doktori értekezésem és a tézisek nyilvánosságra kerüljenek az Egyetemi Könyvtárban és az ELTE Digitális Intézményi Tudástárban.

2. A doktori értekezés szerzőjeként kijelentem, hogy

a) az ELTE Digitális Intézményi Tudástárba feltöltendő doktori értekezés és a tézisek saját eredeti, önálló szellemi munkám és legjobb tudomásom szerint nem sértem vele senki szerzői jogait;

b) a doktori értekezés és a tézisek nyomtatott változatai és az elektronikus adathordozón benyújtott tartalmak (szöveg és ábrák) mindenben megegyeznek.

3. A doktori értekezés szerzőjeként hozzájárulok a doktori értekezés és a tézisek szövegének Plágiumkereső adatbázisba helyezéséhez és plágiumellenőrző vizsgálatok lefuttatásához.

Kelt: Budapest, 2016. augusztus 29.

a doktori értekezés szerzőjének aláírása

Eötvös Loránd University
Faculty of Arts

PhD. Dissertation

Gábor Recski

Computational methods in semantics

Doctoral School of Linguistics
Gábor Tolcsvai Nagy MHAS

Theoretical Linguistics Doctoral Programme
Zoltán Bánréti CSc.

Members of the Committee:
Ferenc Kiefer MHAS (chair)
Péter Rebrus PhD.
Veronika Vincze PhD.
Gábor Alberti DSc.
András Komlósy CSc.

Supervisor:
András Kornai DSc.

Budapest, 2016

Acknowledgements

First and foremost I would like to thank my advisor András Kornai for his ongoing support of my studies and research over the past seven years, during which he supervised all work related to both my masters theses and virtually all projects and assignments related to my studies at ELTE and my work at the Institute for Computer Science (SZTAKI). Not only was András always available and willing to spend his time and energy on lengthy discussions of any topic, it was his efforts that secured a truly inspiring work environment for my studies and research, sometimes against all odds.

For acquainting me with natural language processing, sharing their tremendous experience, teaching me to code, and for being two of the smartest and coolest people I know, I'd like to thank former MOKK colleagues Péter Halácsy and Dániel Varga. I am forever grateful for their patience and helpfulness.

I am thankful to the professors at the Department of Theoretical Linguistics and I am particularly indebted to László Kálmán, who over the last 10 years has been my professor of logic, semantics, syntax, computer science, computational linguistics, and many other language-related topics. I thank him for the countless thought-provoking discussions during his Tuesday seminars, after classes, on nyelveszforum, and at semantics camp. For making electronic copies of the Hungarian dictionaries *A Magyar Nyelv Nagyszótára* and *Magyar Értelmező Kéziszótár* available to me I am grateful to Nóra Ittzés and Gábor Prószéky.

Finally I would like to say thanks to all my current and former colleagues at the Research Group for Mathematical Linguistics: Judit Ács, Gábor Borbély, Márton Makrai, Dávid Nemeskey, Katalin Pajkossy, and Attila Zséder. I feel truly honored to be working with a team of such amazing people.

Contents

1 Introduction 6

1.1 Theses . . . 7

1.2 Contributions . . . 7

1.3 Software . . . 7

2 Theories of word meaning 9

2.1 Katz and Fodor's semantics . . . 9

2.2 Graph-based models of semantics . . . 12

2.2.1 Quillian's Semantic Memory Model . . . 12

2.2.2 Schank's Conceptual Dependencies . . . 18

2.2.3 The KL-ONE family . . . 19

2.2.4 Sowa's Conceptual structures . . . 22

2.2.5 Abstract Meaning Representations . . . 23

2.2.6 WordNet . . . 25

2.3 Montague-style theories . . . 25

2.4 CVS representations . . . 27

2.4.1 Vectors as word representations . . . 27

2.4.2 Vectors beyond the word level . . . 28

3 The `4lang` system 30

3.1 The formalism . . . 30

3.1.1 Nodes . . . 31

3.1.2 The 0-edge . . . 31

3.1.3 1- and 2-edges . . . 31

3.2 Ambiguity and compositionality . . . 33

3.3 Reasoning . . . 36

3.4 Extra-linguistic knowledge . . . 38

3.5 Primitives of representation . . . 40

3.6 Theoretical significance 43

4 From text to concept graph 44

4.1 Dependency parsing 45

4.2 From dependencies to graphs 46

4.3 Utterances 46

4.4 Issues 49

4.4.1 Parsing errors 49

4.4.2 Relationships among clauses 50

4.5 Postprocessing dependencies 52

4.5.1 Coordination 52

4.5.2 Copulars and prepositions 52

4.6 Evaluation 53

4.6.1 Evaluation on raw text 54

4.6.2 Evaluation on gold dependencies 55

4.7 Hungarian 56

4.7.1 Dependencies 57

4.7.2 Morphology 58

4.7.3 Postprocessing 59

4.7.4 Evaluation and issues 61

5 Building definition graphs 67

5.1 Data sources 67

5.1.1 Longman Dictionary of Contemporary English 67

5.1.2 Collins Cobuild Dictionary 68

5.1.3 English Wiktionary 68

5.1.4 Dictionaries of Hungarian 69

5.2 Parsing definitions 69

5.2.1 Preprocessing 69

5.2.2 Constraining the parser 70

5.2.3 Building definition graphs 71

5.3 Expanding definition graphs 71

5.4 Issues and evaluation 72

5.4.1 Error analysis 72

5.4.2 Non-standard definitions 74

5.4.3 Word senses 75

5.4.4 Hungarian 76

5.4.5 Comparing dictionaries 77

6 Applications **82**

6.1 Semantic similarity of sentences 82

6.1.1 The STS task 83

6.1.2 Datasets 83

6.1.3 Architecture of the `MathLingBudapest` systems 84

6.1.4 `4lang`-based similarity 85

6.1.5 Submissions 87

6.1.6 Difficulties 88

6.2 Word similarity 89

6.2.1 Previous work 90

6.2.2 Setup 91

6.2.3 External models 91

6.2.4 `4lang`-based features 92

6.2.5 Results 94

6.3 Natural language understanding 96

7 System architecture **97**

7.1 Overview 97

7.2 Requirements 98

7.2.1 `pymachine` 98

7.2.2 `hunmorph` and `hundisambig` 98

7.2.3 Stanford Parser and CoreNLP 99

7.3 `dep_to_4lang` 99

7.4 `dict_to_4lang` 106

7.4.1 Parsing dictionaries 109

7.4.2 Preprocessing entries 124

7.4.3 Parsing definitions 126

7.4.4 Processing dependencies 136

7.5 The `Lexicon` class 142

7.6 The `Lemmatizer` class 149

7.7 The `pymachine` library 152

7.8 The `similarity` module 153

7.9 Configuration 166

8 Outlook 167
8.1 Outstanding issues . . . 167
8.1.1 True homonyms . . . 167
8.1.2 Alternate word forms, synonyms . . . 168
8.2 Sentence similarity and entailment . . . 168
8.3 Question Answering . . . 170
8.4 Parsing in `4lang` . . . 170
8.5 Likelihood of `4lang` representations . . . 171
8.5.1 A probabilistic approach . . . 172
8.5.2 An inference-based approach . . . 173
8.6 External sources . . . 173
8.6.1 World knowledge . . . 173
8.6.2 Constructions . . . 174

Appendices 177

A Configuration file of the `4lang` module 178

References 182

Chapter 1

Introduction

This thesis presents computational methods for creating semantic representations of natural language utterances and some early applications of such representations in various computational semantics tasks. All software presented in this thesis is free and open-source, distributed under an MIT license and downloadable from the URLs listed in Section 1.3. This introductory chapter presents the theses of the dissertation, acknowledges contributions from colleagues, and for each system presented, provides links to the software and references to key publications.

The remainder of the thesis is structured as follows: Chapter 2 gives a short review of existing theories of word meaning, with special focus on their applicability to natural language processing. Chapter 3 provides an overview of the `4lang` formalism for modeling meaning, but will not attempt a full discussion, since the `4lang` formalism is the product of joint work by half a dozen researchers (Kornai et al., 2015), rather than being a contribution of this thesis. Chapter 4 presents the `dep_to_4lang` pipeline, which creates `4lang`-style meaning representations from running text, Chapter 5 describes its application to monolingual dictionary definitions, `dict_to_4lang`, used to create large concept lexica automatically. Chapter 6 presents applications of the `text_to_4lang` module to various tasks in Computational Semantics, including a competitive system for measuring semantic textual similarity (STS) (Recski & Ács, 2015), and a hybrid ML-based system for measuring the similarity of English word pairs, which at the time of submission is the top-scoring algorithm on the popular SimLex benchmark dataset (Recski, Iklódi, et al., 2016). The chapter also briefly describes an experimental framework for natural language understanding (Nemeskey et al., 2013) based on `4lang` representations. Chapter 7 presents the architecture of the ca. 3000-line `4lang` codebase, serving both as an overview of how the main tools presented in the thesis are implemented and as comprehensive software documentation. Finally, Chapter 8 discusses our plans for future applications.

1.1 Theses

The main theses of the disseration are the following:

(T1) The `text_to_4lang` tool for building `4lang`-style semantic representations from English and Hungarian raw text

(T2) The `dict_to_4lang` tool for building `4lang` definition graphs from monolingual dictionaries of English and Hungarian

(T3) A competitive system for measuring the semantic similarity of English sentence pairs using definition graphs built by `dict_to_4lang`

(T4) The current state of the art algorithm for measuring the semantic similarity of English word pairs using features extracted from `4lang` graphs

1.2 Contributions

The `4lang` principles outlined in Chapter 3 are the result of collaboration with current and former members of the Research Group for Mathematical Linguistics at the Hungarian Academy of Sciences: Judit Ács, Gábor Borbély, András Kornai, Márton Makrai, Dávid Nemeskey, Katalin Pajkossy, and Attila Zséder. The systems presented in Chapters 4 and 5 constitute the author's work with only minor exceptions: the functions performing graph expansion (Section 5.3) are a result of joint work with Gábor Borbély, and a parser for the Collins Dictionary was contributed by Attila Bolevácz. The SemEval system presented in Section 6.1 were built in collaboration with Judit Ács, the more recent `wordsim` system presented in Section 6.2 is a result of joint work with Eszter Iklódi (Department of Automation and Applied Informatics, Budapest University of Technology and Economics), key ML components were contributed by Katalin Pajkossy. The experimental systems described in Section 6.3 were implemented together with Dávid Nemeskey and Attila Zséder.

1.3 Software

All software presented in the thesis is available for download under an MIT license, URLs are listed in Table 1.1. The `text_to_4lang` and `dict_to_4lang` tools (T1-T2) are parts of the `4lang` library, some dependencies are included in the package `pymachine`. The state of the `4lang` codebase at the time of submission of this thesis is preserved in the

System	Code	Main publication
4lang	github.com/kornai/4lang	(Recski, 2016)
pymachine	github.com/kornai/pymachine	
semeval	github.com/juditacs/semeval	(Recski & Ács, 2015)
4lang	github.com/recski/wordsim	(Recski, Iklódi, et al., 2016)

Table 1.1: Software libraries presented in this thesis

branch recski_thesis. The sentence similarity system (T3) is preserved in the semeval repository, the word similarity system (T4) is part of the wordsim package. All external dependencies of these systems are freely downloadable under various open-source licenses.

Chapter 2

Theories of word meaning

This chapter gives a survey of approaches to modeling the semantics of natural language, focusing on key ideas in representing word meaning. Our overview is neither complete, nor does it provide a full introduction to any theory in particular, it is merely an overview of major contributions to word meaning representation. We begin with a short overview of the historically central Katz and Fodor's *Structure of a Semantic Theory* (Section 2.1), followed by reviews of several graph-based models of word meaning in Section 2.2, among others the Semantic Memory Model of Quillian, the KL-ONE family of formalisms, or the more recent Abstract Meaning Representation framework. An overview of Montagovian approaches to word meaning is given in Section 2.3. Finally, in Section 2.4, we discuss continuous vector space semantics, the approach to representing word meaning that is currently most widely used in natural language processing.

2.1 Katz and Fodor's semantics

In their paper *The Structure of a Semantic Theory*, Katz and Fodor (1963) set a lower bound on what a theory of semantics must include. Their examples show three skills of a competent speaker to be independent of their knowledge of grammar: (i) handling ambiguity (the bill is large, but need not be paid), (ii) detecting anomaly (the paint is silent) and (iii) paraphrasing (What does the note say? Does it say X?).

In setting an upper bound on the domain of semantics, they disown the issue of disambiguating between various readings of the same sentence (in isolation) based on context, since that would require modeling all extralinguistic knowledge:

> "...if a theory of setting selection is to choose the correct reading for the sentence *Our store sells alligator shoes*, it must represent the fact that, to date, alligators

> do not wear shoes, although shoes for people are sometimes made from alligator skin". (Katz & Fodor, 1963, p.178)

Katz and Fodor conclude that the upper bound on a semantic theory should be that of semantic interpretation - a function that maps each sentence to a set of semantic representations, one corresponding to each possible reading of the sentence. They make clear that they impose this limit merely for practicality, because they "cannot in principle distinguish between the speaker's knowledge of his language and his knowledge of the world, because (...) part of the characterization of a LINGUISTIC ability is a representation of virtually all knowledge about the world that speakers share." (Katz & Fodor, 1963, p.179, emphasis in original) In Section 3.3 we shall also argue that any apparatus capable of representing the meaning of natural language utterances must be capable of representing all of (naive, non-technical) world knowledge.)

In describing the components of a semantic theory, Katz and Fodor define the lexicon to contain separate entries for multiple *senses* of each word, and at the same time they state that the grammar and the lexicon together are still insufficient for a deterministic semantic interpretation, because of the multiple senses associated with most word forms. A *projection rule* that selects the appropriate sense of each word form in a sentence is postulated. This rule requires the senses of each word to be structured in the lexicon as exemplified in Figure 2.1. In Chapter 3 we shall describe the `4lang` representation of word meaning that is radically *monosemic*, i.e. makes as little use of *word senses* as possible and would map a word such as *bachelor* to a single representation that is compatible with all uses of the word.

Note that the representation of lexical items in Figure 2.1 also includes a theory of semantic primitives (*human, male, animal,* etc., Katz and Fodor refer to these as *semantic markers*), much in the spirit of Prague-style phonological theory (Trubetzkoy, 1958). A significant problem with this approach is that they have little to say about where the set of all semantic markers available might come from, i.e. what the primitives of their representation should be. All remaining lexical information about a word sense that is not contained in the semantic markers, i.e. the parts in square brackets in Figure 2.1 are called *distinguishers.* This distinction between the layers of markers and distinguishers is not unlike that between Aristotle's *genus* and *differentia* (Smith, 2015). Katz and Fodor also claim that distinguishers are out of reach for a theory of semantics:

> "The distinction between markers and distinguishers is meant to coincide with the distinction between that part of the meaning of a lexical item which is systematic for the language and that part which is not. In order to describe the

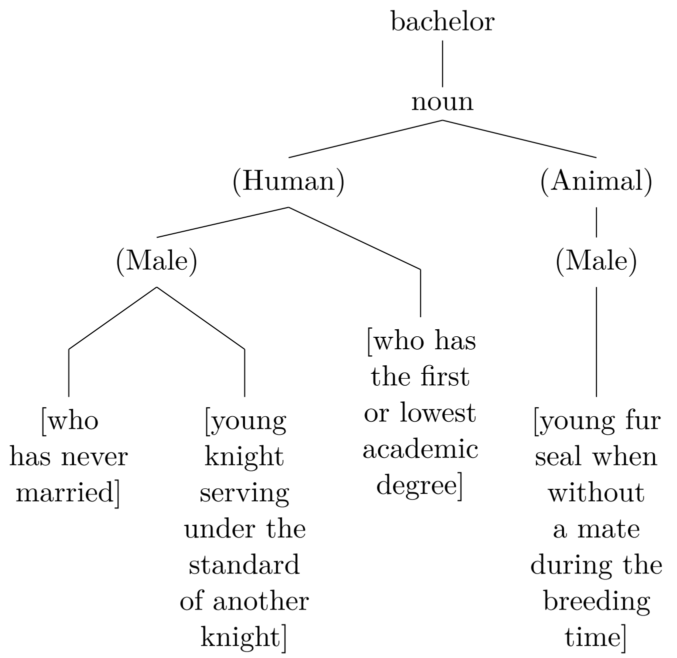

bachelor
noun
(Human)
(Animal)
(Male)
(Male)
[who has never married]
[young knight serving under the standard of another knight]
[who has the first or lowest academic degree]
[young fur seal when without a mate during the breeding time]

Figure 2.1: Decomposition of lexical items (Katz & Fodor, 1963, p.186)

> systematicity in the meaning of a lexical item, it is necessary to have theoretical constructs whose formal interrelations compactly represent this systematicity. The semantic markers are such constructs. The distinguishers, on the other hand, do not enter into theoretical relations within a semantic theory. The part of the meaning of a lexical item that a dictionary represents by a distinguisher is the part of which a semantic theory offers no general account." (Katz & Fodor, 1963, p.178)

What this last statement amounts to is that the (finite) set of semantic markers is a set of universal primitives that is sufficient for representing the language-independent component of word meaning. Then, if some non-English word is a hypernym of $\texttt{bachelor}_1$ - *man who has never married*, then its set of semantic markers must be a subset of the markers in the entry for $\texttt{bachelor}_1$. On the other hand, if we find a word in some language that is the hyponym of $\texttt{bachelor}_1$, e.g. a word `w` that means *a man who has never married and lives with his parents*, we must conclude that our original representation for $\texttt{bachelor}_1$ was inadequate, since the components of its meaning beyond *male* and *human*, whatever they may be, are shared with the entry `w` and should therefore be encoded by semantic markers, not distinguishers. Since the potential absence of such a word `w` from all human languages can only be accidental, we have to conclude that the distinction between meaning encoded by markers and by distinguishers is also arbitrary. Bolinger (1965, p.560) makes a similar argument, demonstrating that for virtually any component of any distinguisher in Figure 2.1 it is possible to construct an example that justifies ‘promoting‘ that particular component to marker status, and concluding that “it is possible to do away with the dualism by converting the distinguisher into a string of markers”. We shall return to his examples in Section 3.5 when we argue for a theory of meaning representation that encodes word meaning using language-independent primitives – and nothing else!

Finally, Katz and Fodor claim that word meaning representations may contain limitations on the semantic content of elements with which the given word can combine. In their example, an excerpt from *The shorter Oxford English dictionary*, the entry *honest* contains the definition ‘... of women: chaste, virtuous’; such requirements they would represent by adding constraints such as (Human) and (Female) on certain *paths* of the representation (paths in the sense of Figure 2.1). Section 3.2 will discuss how such constraints may be enforced by a `4lang`-based system that lacks a notion of *paths* or *senses*.

2.2 Graph-based models of semantics

This section reviews popular systems for representing meaning using graphs – networks of nodes and edges connecting them. We shall summarize the basic principles of Quillian’s 1960s *Memory Model* in Section 2.2.1, Schank’s Conceptual Dependencies in Section 2.2.2, the KL-ONE family of Knowledge Representation systems, widely used between the late 1970s and early 1990s, in Section 2.2.3, Sowa’s Conceptual Structures in Section 2.2.4, and finally in Section 2.2.5 the most recent formalism of *Abstract Meaning Representations* which has been gaining popularity in the past 4 years. All these systems share some common principles of representation with each other and with `4lang`, e.g. that each map lexical items to nodes in some graph and use directed edges to represent asymmetric relationships between them. Where they differ significantly is their elements of representation or their notions of a syntax-semantics interface.

2.2.1 Quillian’s Semantic Memory Model

Memory model Quillian’s theory of *word concepts* (1968) is of particular interest to us. Not only does he propose to represent word meaning by means of directed graphs of concepts (much like the `4lang` theory that serves as the basis of this thesis and will be introduced in Chapter 3), it also defines graph configurations that are in many ways similar to those in `4lang`. Quillian also suggests that definitions of concepts should be learned automatically, which is exactly what our module `dict_to_4lang` does (see Chapter 5).

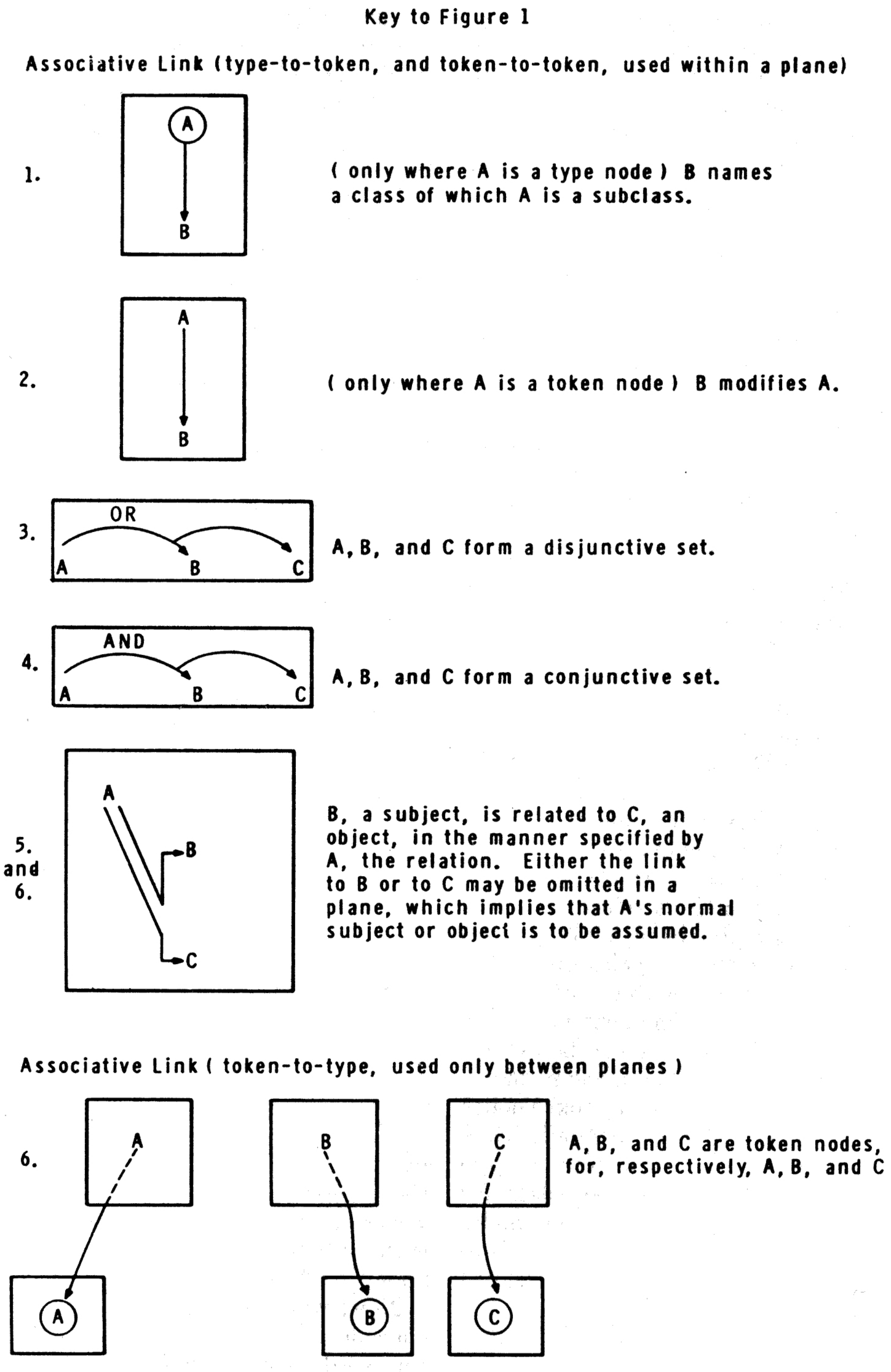
Quillian proposes to encode meaning as a graph of nodes representing concepts, and *associative links* between nodes, which may encode a variety of semantic relationships between these concepts. Figure 2.2 reproduces Quillian's original presentation of associative link types. Types 1 and 2, which stand for hypernymy and attribution respectively – encode relationships that `4lang` will treat as a single relation (along with predication, see Section 3.1). Also, his links of type 5 and 6 are not unlike the binary configuration in `4lang` graphs.

Quillian proposes two types of nodes: *type nodes* are unique for each concept and serve to define them as networks of other concepts. *Token nodes* occur multiple times for each concept when they themselves are used in definitions. In Section 6.3, when we review early attempts at inferencing on `4lang` representations, we shall see that this distinction is not unlike that of *static* and *active* nodes made by (Nemeskey et al., 2013). Quillian organizes nodes into *planes*, one for each type node and its definition graph, and emphasizes the need to perform an exhaustive search of an arbitrary number of such planes for a complete definition of any concept:

> *"a word's full concept is defined in the model memory to be all the nodes that can be reached by an exhaustive tracing process, originating at its initial, patriarchal type node, together with the total sum of relationships among these nodes specified by within-plane, token-to-token links* (...) This information will start off with the more "compelling" facts about machines, such as that they are usually man-made, involve moving parts, and so on, and will proceed "down" to less and less inclusive facts, such as that typewriters are machines, and then eventually will get to much more remote information about machines, such as the fact that a typewriter has a stop that prevents its carriage from flying off everytime it is returned." (Quillian, 1968, p.413, emphasis in original)

Quillian concludes that the bulk of information associated with a concept such as *machine* must be an unstructured list of all concepts that refer to types of machines and as such have edges directed towards tokens of *machine*. A distinction is made, then, between the *definition* of some concept, i.e. the tokens accessible (in the digraph sense) from its type node, and the network of all nodes connected to any token of the concept, all potentially carrying information about the concept – in Section 3.2 we shall argue that it is the latter that must be accessible to any language understanding mechanism.

Unlike Katz and Fodor, Quillian suggests not to represent the complex meaning of a word by means of a hierarchical structure of word senses. Instead he suggests that the unified network of all concepts that link to either the type node or to some token node



Key to Figure 1

Associative Link (type-to-token, and token-to-token, used within a plane)

1. (only where A is a type node) B names a class of which A is a subclass.

2. (only where A is a token node) B modifies A.

3. A, B, and C form a disjunctive set.

4. A, B, and C form a conjunctive set.

5. and 6. B, a subject, is related to C, an object, in the manner specified by A, the relation. Either the link to B or to C may be omitted in a plane, which implies that A's normal subject or object is to be assumed.

Associative Link (token-to-type, used only between planes)

6. A, B, and C are token nodes, for, respectively, A, B, and C

FIG. 1. Sample Planes from the Memory.

Figure 2.2: Associative links (Quillian, 1968, p.412)

of the concept being defined should by itself serve as a store of all knowledge associated with some word. He criticizes hierarchical structures of word senses commonly found in explanatory dictionaries by pointing out that "the common elements within and between various meanings of a word are many, and any outline designed to get some of these together under common headings must at the same time necessarily separate other common elements, equally valid from some other point of view" (Quillian, 1968, p.419). Nevertheless, the memory model still makes use of *word senses* and the proposed mechanism for building semantic representations from any given sentence still requires to select for each word exactly one of several encoded senses. In Section 3.2 we shall propose a *radically monosemic* approach to representing word meaning which abolishes the concept of multiple word senses (with the exception of true homonyms such as the *trunk* of a car and the *trunk* of an elephant).

Quillian also suggests that most concept definitions could be acquired algorithmically given a small set of predefined primitives and definitions written in natural language:

> "if one could manage to get a small set of basic word meanings adequately encoded and stored in computer memory, and a workable set of combination rules formalized as a computer program, he could then bootstrap his store of encoded word meanings by having the computer itself "understand" sentences that he had written to constitute the definitions of other single words" (Quillian, 1968, p.416)

It is precisely this bootstrapping process that the `dict_to_4lang` module of the `4lang` library, described in detail in Chapter 5, performs using definitions from explanatory dictionaries of English and Hungarian as well as a set of some 2,200 manually predefined concepts.

Language understanding The above model of semantic memory serves as the basis of a full-fledged language understanding system introduced in (Quillian, 1969). The process the *Teachable Language Comprehender* (TLC) applies to language understanding involves retrieving for each entity in the input text a list of concepts and entities in its memory that the text may be mentioning. For these newly created copies of concepts, the TLC also initializes pointers for each valency of the given concept: e.g. given a mention of *client*, defined as seen in Figure 2.3, pointers to employer and employee are created as such that should eventually be filled in the process of comprehending the full text. TLC then conducts for each pointer a search for compatible properties present in its current representation of the input, thus generating a list of candidates for the pointer. E.g. given the phrase *lawyer's client*, `lawyer` will eventually be found as compatible with the property `employer` of `client`, since both are linked to the property `professional`. This iterative search process also incorporates anaphora resolution: pointers may be filled with referents already present in the model of the current input. The next step involves trying to justify connections from syntax: TLC's memory also contains a set of *form tests*, each of which encode some particular configuration that is typical of a semantic relation (e.g. in this case "X's Y" or "Y of X") An example of a sample TLC session is reproduced from (Quillian, 1969) in Figure 2.4.

Note that Quillian's model is that of a *teachable* language comprehender; his account also involves feedback given by human supervisors of the process, teaching the system e.g. new form tests for each link of each concept as they occur. Such a system could be trained through human labor to make highly reliable judgments as to whether some entities in a text refer to a client and her employer. Human supervision would be necessary for practically all concepts with arguments. The framework we propose in this thesis is intended to be more robust by using more generic concept representations. The `4lang` representation of `client` may be as simple as `work` $\xleftarrow{1}$ `FOR` $\xrightarrow{2}$, but this is with the intention of leaving open as many interpretations as possible (see Section 3.2 for more discussion).

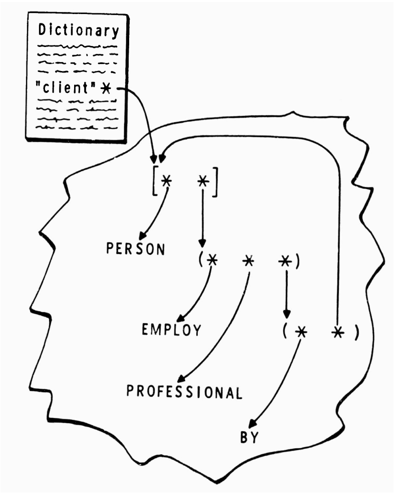
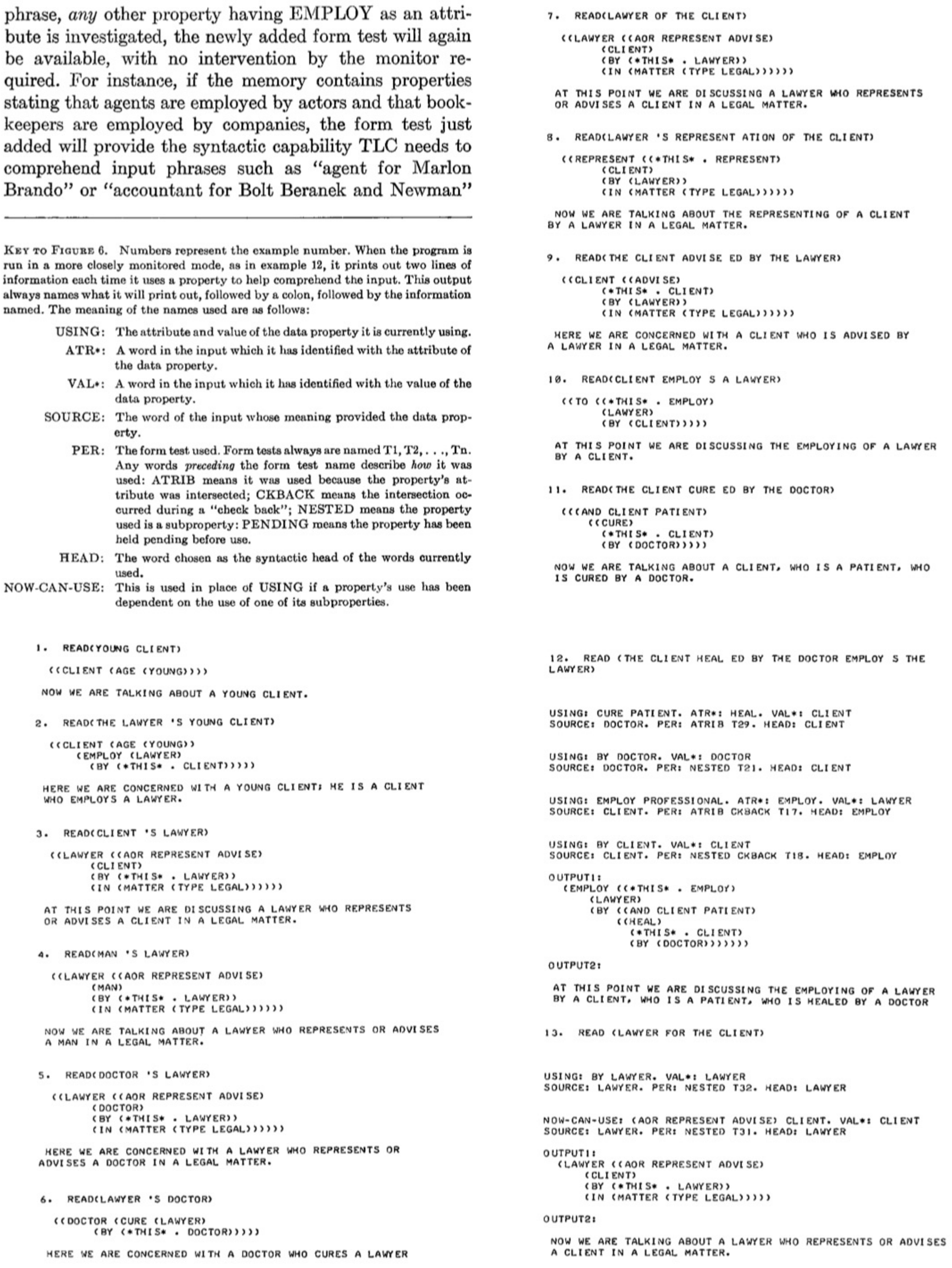


Figure 2.3: Quillian's definition of *client* (Quillian, 1969, p.462)

Quillian's theory of the semantic memory has had widespread effect on both the theory and application of (computational) semantics. (Collins & Loftus, 1975) proposed a method for natural language understanding using spreading activation over Quillian's semantic memory model. Anderson and Bower (1973) introduced the `HAM` question answering system based on a model of *associative memory* similar to Quillian's. Subsequent associative models include the spreading activation-based `ACT` system (Anderson, 1976) and the memory model `MEMOD` (Rumelhart et al., 1972). More recent models of (lexical) semantics still rely on an associative structure of the lexicon, a notable example being Abstract Meaning Representations, to be introduced in Section 2.2.5. Finally, the `4lang`



phrase, *any* other property having EMPLOY as an attribute is investigated, the newly added form test will again be available, with no intervention by the monitor required. For instance, if the memory contains properties stating that agents are employed by actors and that bookkeepers are employed by companies, the form test just added will provide the syntactic capability TLC needs to comprehend input phrases such as "agent for Marlon Brando" or "accountant for Bolt Beranek and Newman"

KEY TO FIGURE 6. Numbers represent the example number. When the program is run in a more closely monitored mode, as in example 12, it prints out two lines of information each time it uses a property to help comprehend the input. This output always names what it will print out, followed by a colon, followed by the information named. The meaning of the names used are as follows:

USING: The attribute and value of the data property it is currently using.

ATR*: A word in the input which it has identified with the attribute of the data property.

VAL*: A word in the input which it has identified with the value of the data property.

SOURCE: The word of the input whose meaning provided the data property.

PER: The form test used. Form tests always are named T1, T2, . . ., Tn. Any words *preceding* the form test name describe *how* it was used: ATRIB means it was used because the property's attribute was intersected; CKBACK means the intersection occurred during a "check back"; NESTED means the property used is a subproperty: PENDING means the property has been held pending before use.

HEAD: The word chosen as the syntactic head of the words currently used.

NOW-CAN-USE: This is used in place of USING if a property's use has been dependent on the use of one of its subproperties.

```
1.  READ(YOUNG CLIENT)

  ((CLIENT (AGE (YOUNG))))

 NOW WE ARE TALKING ABOUT A YOUNG CLIENT.

2.  READ(THE LAWYER 'S YOUNG CLIENT)

  ((CLIENT (AGE (YOUNG))
      (EMPLOY (LAWYER)
        (BY (*THIS* . CLIENT)))))

 HERE WE ARE CONCERNED WITH A YOUNG CLIENT; HE IS A CLIENT
 WHO EMPLOYS A LAWYER.

3.  READ(CLIENT 'S LAWYER)

  ((LAWYER ((AOR REPRESENT ADVISE)
         (CLIENT)
         (BY (*THIS* . LAWYER))
         (IN (MATTER (TYPE LEGAL))))))

 AT THIS POINT WE ARE DISCUSSING A LAWYER WHO REPRESENTS
 OR ADVISES A CLIENT IN A LEGAL MATTER.

4.  READ(MAN 'S LAWYER)

  ((LAWYER ((AOR REPRESENT ADVISE)
         (MAN)
         (BY (*THIS* . LAWYER))
         (IN (MATTER (TYPE LEGAL))))))

 NOW WE ARE TALKING ABOUT A LAWYER WHO REPRESENTS OR ADVISES
 A MAN IN A LEGAL MATTER.

5.  READ(DOCTOR 'S LAWYER)

  ((LAWYER ((AOR REPRESENT ADVISE)
         (DOCTOR)
         (BY (*THIS* . LAWYER))
         (IN (MATTER (TYPE LEGAL))))))

 HERE WE ARE CONCERNED WITH A LAWYER WHO REPRESENTS OR
ADVISES A DOCTOR IN A LEGAL MATTER.

6.  READ(LAWYER 'S DOCTOR)

  ((DOCTOR (CURE (LAWYER)
         (BY (*THIS* . DOCTOR)))))

 HERE WE ARE CONCERNED WITH A DOCTOR WHO CURES A LAWYER
```

```
7.  READ(LAWYER OF THE CLIENT)

  ((LAWYER ((AOR REPRESENT ADVISE)
         (CLIENT)
         (BY (*THIS* . LAWYER))
         (IN (MATTER (TYPE LEGAL))))))

 AT THIS POINT WE ARE DISCUSSING A LAWYER WHO REPRESENTS
 OR ADVISES A CLIENT IN A LEGAL MATTER.

8.  READ(LAWYER 'S REPRESENT ATION OF THE CLIENT)

  ((REPRESENT ((*THIS* . REPRESENT)
         (CLIENT)
         (BY (LAWYER))
         (IN (MATTER (TYPE LEGAL))))))

 NOW WE ARE TALKING ABOUT THE REPRESENTING OF A CLIENT
BY A LAWYER IN A LEGAL MATTER.

9.  READ(THE CLIENT ADVISE ED BY THE LAWYER)

  ((CLIENT ((ADVISE)
         (*THIS* . CLIENT)
         (BY (LAWYER))
         (IN (MATTER (TYPE LEGAL))))))

 HERE WE ARE CONCERNED WITH A CLIENT WHO IS ADVISED BY
A LAWYER IN A LEGAL MATTER.

1Ø.  READ(CLIENT EMPLOY S A LAWYER)

  ((TO ((*THIS* . EMPLOY)
         (LAWYER)
         (BY (CLIENT)))))

 AT THIS POINT WE ARE DISCUSSING THE EMPLOYING OF A LAWYER
 BY A CLIENT.

11.  READ(THE CLIENT CURE ED BY THE DOCTOR)

  (((AND CLIENT PATIENT)
      ((CURE)
         (*THIS* . CLIENT)
         (BY (DOCTOR)))))

 NOW WE ARE TALKING ABOUT A CLIENT, WHO IS A PATIENT, WHO
 IS CURED BY A DOCTOR.



12.  READ (THE CLIENT HEAL ED BY THE DOCTOR EMPLOY S THE
LAWYER)


USING: CURE PATIENT. ATR*: HEAL. VAL*: CLIENT
SOURCE: DOCTOR. PER: ATRIB T29. HEAD: CLIENT


USING: BY DOCTOR. VAL*: DOCTOR
SOURCE: DOCTOR. PER: NESTED T21. HEAD: CLIENT


USING: EMPLOY PROFESSIONAL. ATR*: EMPLOY. VAL*: LAWYER
SOURCE: CLIENT. PER: ATRIB CKBACK T17. HEAD: EMPLOY


USING: BY CLIENT. VAL*: CLIENT
SOURCE: CLIENT. PER: NESTED CKBACK T18. HEAD: EMPLOY

OUTPUT1:
   (EMPLOY ((*THIS* . EMPLOY)
       (LAWYER)
       (BY ((AND CLIENT PATIENT)
           ((HEAL)
             (*THIS* . CLIENT)
             (BY (DOCTOR)))))))

OUTPUT2:

 AT THIS POINT WE ARE DISCUSSING THE EMPLOYING OF A LAWYER
 BY A CLIENT, WHO IS A PATIENT, WHO IS HEALED BY A DOCTOR

13.  READ (LAWYER FOR THE CLIENT)


USING: BY LAWYER. VAL*: LAWYER
SOURCE: LAWYER. PER: NESTED T32. HEAD: LAWYER


NOW-CAN-USE: (AOR REPRESENT ADVISE) CLIENT. VAL*: CLIENT
SOURCE: LAWYER. PER: NESTED T31. HEAD: LAWYER

OUTPUT1:
  (LAWYER ((AOR REPRESENT ADVISE)
       (CLIENT)
       (BY (*THIS* . LAWYER))
       (IN (MATTER (TYPE LEGAL)))))

OUTPUT2:

 NOW WE ARE TALKING ABOUT A LAWYER WHO REPRESENTS OR ADVISES
 A CLIENT IN A LEGAL MATTER.
```

Figure 2.4: Sample session of the Teachable Language Comprehender (Quillian, 1969, p.470)

theory of semantic representation, the basis of all systems introduced in this thesis, also employs a network of associated concepts as its primary tool for representing linguistic semantics and for encoding world knowledge.

2.2.2 Schank's Conceptual Dependencies

Another formalism developed in the 1960s for representing meaning as networks of concepts is Schank's theory of Conceptual Dependencies (1969; 1972), henceforth CD. CD distinguishes between 6 concept categories, which indicate how dependencies between pairs of them should be interpreted. *Actors* and *objects* form the PP category, they may govern their attributes of type PA (e.g. book ← red) and they may be governed by *actions* (ACT) (e.g. steals ← book). Bidirectional dependencies hold between actors and actions as well as actors and attributes, these propositions are known as *conceptualizations*, e.g. man ⇌ steals. Conceptualizations can themselves take part in dependency relations, e.g. the sentence *John's love is good* will be represented by the network in Figure 2.5. CD representations also represent tense and mood of propositions by distinguishing between 8 types of the two-way dependency relation used in conceptualizations, e.g. a conditional statement will invoke the edge $\overset{C}{\rightleftharpoons}$.

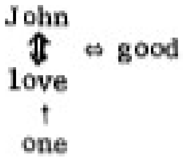


Figure 2.5: Conceptual dependency representation of *John's love is good* (Schank & Tesler, 1969, p.8)

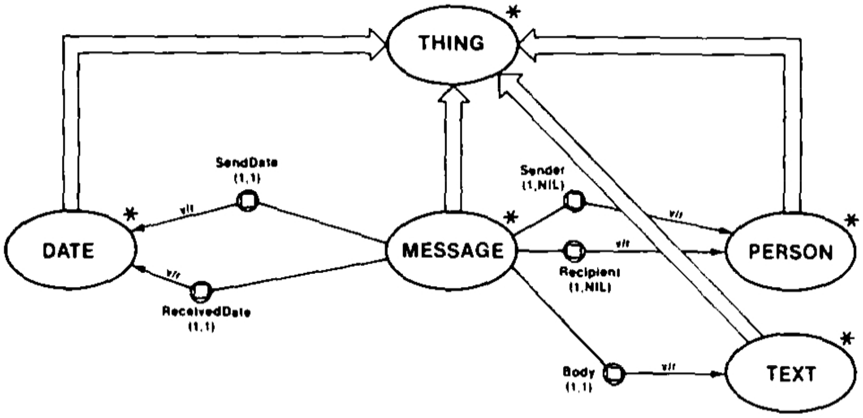
CD networks, like the 4lang graphs introduced in the next chapter, are language- and grammar-independent representations. The generation of CD representations from analyzed linguistic input is achieved via *realization* rules, e.g. one of English that maps the sequence ADJ + N to the CD template PA → PP. The system presented in Chapter 4 of this thesis will implement rules that are essentially similar, since they will generate 4lang subgraphs from dependency relations in the output of a syntactic parser. CD also makes use of simple *constructions*, explicitly mapping a phrase such as *a cup of water* to the representation cup ⇌ contains ← water.

Concepts denoting actions in CD are defined using a set of 10-12 primitives such as PTRANS: *The transfer of location of an object* or MBUILD: *The construction of a thought or of new information by an agent.* Each of these primitives enforces restrictions on concepts governed by an action - in this respect CD is similar to the KL-ONE representation summarized in Section 2.2.3, which uses *Roles* and *RoleSets* to place restrictions on relations of a concept. For example, the slots associated with any PTRANS action are ACTOR, OBJECT, FROM, and TO.

2.2.3 The KL-ONE family

The KL-ONE system (Brachman & Schmolze, 1985) and its successors (Moser, 1983; Brachman et al., 1983) are systems for Knowledge Representation (KR) rather than models of linguistic semantics. They are of great historical significance in the field of Artificial Intelligence and their formalisms are in many ways similar to both `4lang` and the other graph-based models mentioned in this section.

Representation Like many other approaches, KL-ONE adopts the tradition of representing information as a network of nodes and links between them. Nodes in KL-ONE networks represent *Concepts*, which are defined by three components: a list of *superconcepts*, whose properties they inherit, a list of *Roles*, describing the relationships between the concept and other concepts, and *structural descriptions*, which describe the relationship between Roles. *RoleSets* specify attributes that hold for all fillers occupying some Role, e.g. that in case of the concept `message`, the `sender` must be a `person`; such conditions are known as *Value Restrictions*. Structural Descriptions (SDs) of KL-ONE concepts serve to characterize the relationship between Roles of a Concept, e.g. that an `important message` is such that the sender is the supervisor of the recipient. A sample KL-ONE concept is depicted in Figure 2.6, along with its equivalent in JARGON, an English-like, human-readable specification language for KL-ONE.



"A MESSAGE is, among other things, a THING with at least one Sender, all of which are PERSONs, at lease one Recipient, all of which are PERSONs, a Body, which is a TEXT, a SendDate, which is a DATE, and a ReceivedDate, which is a DATE."

Figure 2.6: A primitive concept in KL-ONE and its specification in JARGON (Brachman & Schmolze, 1985, p.183)

KL-ONE explicitly forbids any violations of Value Restrictions, a clear symptom that it is a formalism for the representation of (formalized) knowledge rather than a tool for modeling language meaning directly. To account for exceptions, it is the inheritance of properties between concepts that may be defined in a way that allows for potential violations; e.g. `elephants` are defined as `four-legged-mammals`, “unless you have information to the contrary” (Brachman & Schmolze, 1985, p.190). The relationship between a concept and its super-concepts is known in KL-ONE as *subsumption*. RoleSets may enter in to a similar relationship called *restriction*, which results in the RoleSet of some concept inheriting the properties of a RoleSet of some super-concept – similar to how classes inherit functions from their superclasses in programming languages.

Semantic parsing The outline of a system mapping natural language input to KL-ONE representation is also presented in (Brachman & Schmolze, 1985). We briefly review its capabilities, since the main contribution of our thesis is also a system for mapping raw text to its meaning representation. Similar to the `text_to_4lang` system, which we describe in Chapter 4, the natural language understanding system described by Brachman and Schmolze relies on a syntactic parser (Bobrow, 1979a), the output of which is then used to build semantic representations. For the latter step, the PSI-KLONE tool is used (Bobrow, 1979b), the output of which can then serve as the input to a component responsible for handling pragmatics, bookkeeping of knowledge acquired in various contexts, etc.

The main idea behind the PSI-KLONE system is that the syntactic representation serving as its input is already encoded in a KL-ONE network, with Concepts such as `NP`, RoleSets such as `PP-modifier`, etc. The system processes a sentence by fragments received from the parser, providing feedback to it if the semantic interpretation fails and the parsing hypothesis cannot be maintained. The interpretation process itself relies on maps from words to lemmas and from lemmas to Concepts, e.g. *teaches* is mapped to the `TEACH-VERB` concept via *teach*, *professor* is mapped to `TEACHER-NOUN`, etc. Concepts retrieved this way are combined with the *syntaxonomy*, the KL-ONE network describing the relationships between syntactic units, e.g. that `VERB` is a sub-concept of `CLAUSE` which is a sub-concept of `PHRASE`. An example representation is shown in Figure 2.7.

Another account of PSI-KLONE (Sondheimer et al., 1984) sheds light on the next steps of semantic interpretation. *Frames* are KL-ONE concepts that describe a ‘semantically distinguishable type of phrase’; e.g. the frame associated with the sending of messages is represented by the `SEND-CLAUSE` concept, whose Roles encode the selection restrictions that apply to such an event and map syntactic functions to semantic relations. For example, a `SEND-CLAUSE` must contain a `TRANSMISSION-VERB` and `MESSAGE-NOUN`, among others, and semantic restrictions on each are imposed in the form of Value Restrictions. The process of mapping a syntactic parse to a KL-ONE network is therefore directly re-

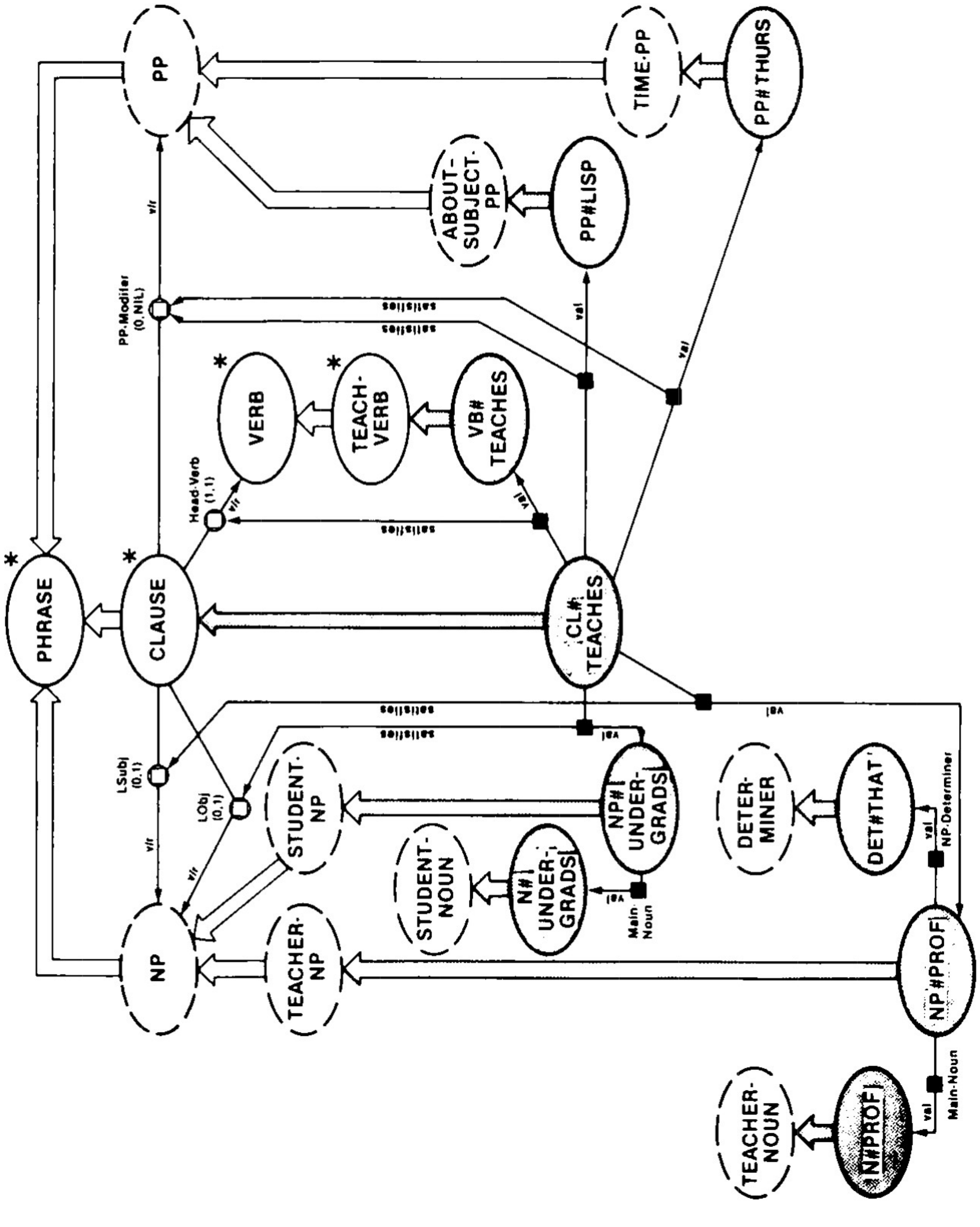


Figure 2.7: KL-ONE representation of *That professor teaches undergraduates about Lisp on Thursday* produced by PSI-KLONE (Brachman & Schmolze, 1985, p.214)

sponsible for producing semantically felicitous representations, unlike the text_to_4lang pipeline described in this thesis, which will produce 4lang graphs describing any states-of-affairs based on its input. Slots of KL-ONE frames are tied to concepts via rules of the form Paraphrase-as X. The frame depicted in Figure 2.8 provides two example rules, stating that the indirect and direct object of a SEND-CLAUSE are to be paraphrased as ADDRESSEE and MESSAGE, respectively. Semantic generalizations over groups of frames can be captured via common super-concepts, known as *abstract case frames*, e.g. all Concepts describing completion of an activity, such as *come*, *reach*, *finish* or *arrive*, can be grouped under an abstract frame from which they inherit the potential to accept time-modifiers. Further descendants of the KL-ONE family include KRYPTON (Brachman et al., 1983), KL-TWO (Vilain, 1985), MANTRA (Bittencourt, 1988), and CLASSIC (Borgida et al., 1989).

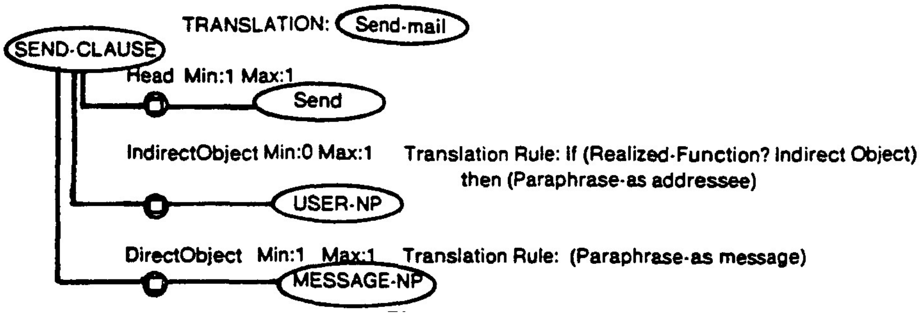

TRANSLATION:
Send-mail
SEND-CLAUSE
Head Min:1 Max:1
Send
IndirectObject Min:0 Max:1
Translation Rule: If (Realized-Function? Indirect Object) then (Paraphrase-as addressee)
USER-NP
DirectObject Min:1 Max:1
Translation Rule: (Paraphrase-as message)
MESSAGE-NP

Figure 2.8: Example of a KL-ONE *frame* (Sondheimer et al., 1984, p.104)

2.2.4 Sowa's Conceptual structures

Conceptual Structures (Sowa, 1984, 1992) have gained popularity in the beginning of the 1990s. Relying on a multitude of well-established tools of both AI and linguistics such as λ-calculus, unification, thematic roles or dependency graphs, Conceptual Structures (CS) intend to serve as both a model of linguistic semantics and a form of universal knowledge representation. CS representations consist of *concepts* and *conceptual relations*. The former may themselves contain arbitrary CS representations and are then referred to as *contexts*. An example CS representation is shown in Figure 2.9.

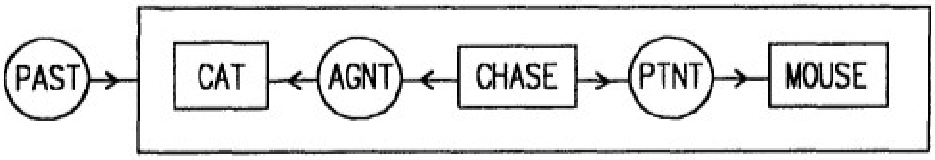


Figure 2.9: CS graph for *A cat chased a mouse* (Sowa, 1992, p.80)

Part of the CS apparatus is the operator ϕ, which maps CS representations to formulas of first-order logic. The structure in Figure 2.9 is mapped to the formula

$$\text{past}((\exists x)(\exists y)(\exists z)(\text{cat}(x) \wedge \text{chase}(y) \wedge \text{mouse}(z) \wedge \text{agnt}(y,x) \wedge \text{ptnt}(y,z)))$$

Standard attribute-value representations (AVMs) used by many KR systems can be straightforwardly mapped to CS representations by mapping attribute values to concepts and attribute types to concept relations that hold between the given concept and the concept corresponding to the entity described by the AVM. Such mappings allow for the automatic creation of CS-style models of various knowledge bases. (Sowa, 1992) describes the transformation of entries created as part of the Cyc project (Lenat & Guha, 1990), plans for building 4lang representations in a similar manner are put forward in Section 8.6 of this thesis. Tools for performing inference over CS graphs have been introduced in e.g. (Fargues et al., 1986) and (Garner & Tsui, 1988).

2.2.5 Abstract Meaning Representations

Abstract Meaning Representation, or AMR (Banarescu et al., 2013), is a more recent formalism for representing the meaning of linguistic structures as directed graphs. The last few years have seen a rise in AMR-related work, including a corpus of AMR-annotated text (Banarescu et al., 2013), several approaches to generating AMRs from running text (Vanderwende et al., 2015; Peng et al., 2015; Pust et al., 2015), and various applications to computational semantics (Pan et al., 2015; Liu et al., 2015).

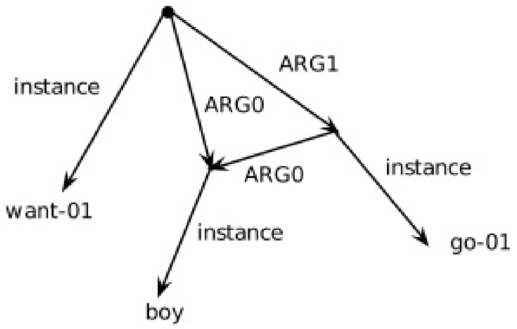
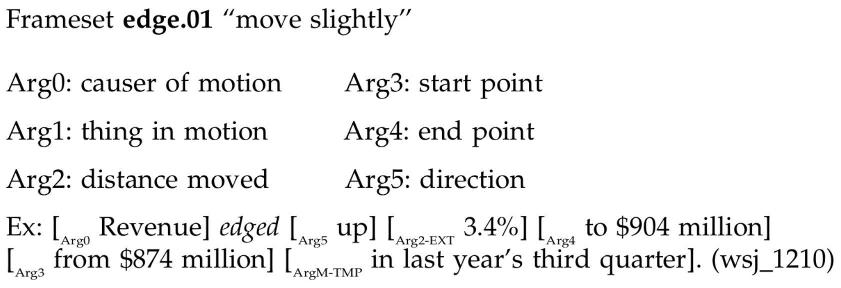


Figure 2.10: AMR representation of *The boy wants to go* (Banarescu et al., 2013, p.179)

Nodes of AMR graphs represent concepts of two basic types: they are either English words, or *framesets* from PropBank (Palmer et al., 2005), used to abstract away from English syntax. PropBank framesets are essentially English verbs (or verb-particle constructions) with a list of possible arguments along with their semantic roles; an example frameset can be seen in Figure 2.11. Unlike the 4lang representation used in this thesis,



Frameset **edge.01** “move slightly”

Arg0: causer of motion Arg3: start point

Arg1: thing in motion Arg4: end point

Arg2: distance moved Arg5: direction

Ex: [$_{\text{Arg0}}$ Revenue] *edged* [$_{\text{Arg5}}$ up] [$_{\text{Arg2-EXT}}$ 3.4%] [$_{\text{Arg4}}$ to $904 million] [$_{\text{Arg3}}$ from $874 million] [$_{\text{ArgM-TMP}}$ in last year’s third quarter]. (wsj_1210)

Figure 2.11: A PropBank frameset (Palmer et al., 2005, p.76)

AMR also makes a distinction similar to Quillian’s type and token nodes by separating nodes that represent some entity, event, property, etc. from nodes that are arguments of some frameset, linking the latter with an *instance* relation to the former. The AMR representation of the sentence *The boy wants to go* would hence be that in Figure 2.10 as opposed to the `4lang` representation in Figure 2.12. AMRs also handles a wide range of phenomena that `4lang` currently doesn’t: the formalism provides relations to encode negation, modals, copulars, and questions. It also includes special relations to encode named entities – in the broader sense, i.e. including not only proper names but also e.g. dates, quantities, etc. The formalism accommodates a wide range of phenomena typical of English, AMR creators admit that “AMR is heavily biased towards English. It is not an Interlingua.” (Banarescu et al., 2013, p.179).

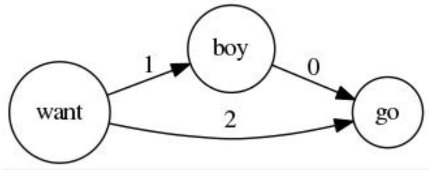


Figure 2.12: `4lang` representation of *The boy wants to go*

AMRs have rapidly gained popularity over the last 3-4 years. Recent parser systems for mapping text to AMR representations include a system based on Hyperedge Replacement Grammars (Peng et al., 2015), a discriminative graph-based parser (Flanigan et al., 2014), a CCG parser (Artzi et al., 2015), a Machine Translation system (Pust et al., 2015), and also a tool which uses dependency parsing as an intermediate step for generating AMRs (Chen, 2015), similar to the method used by the `text_to_4lang` module for generating `4lang` representations from raw text (see Chapter 4). At the 2016 `SemEval` competition, Task 8 (Meaning Representation Parsing) required the 11 participating teams to train systems on AMR-annotated English text and then generate AMR representations for previously unseen English text (May, 2016). All top-scoring systems were derivatives of the CAMR system of (Wang et al., 2015), who process raw text with a standard dependency parser and implement a transition-based parser for transforming dependency trees into AMR graphs.

2.2.6 WordNet

Although not a formalism for semantics in general, we finally mention the WordNet ontology, since it remains one of the most widely used sources of lexical semantic information in natural language processing. WordNet (Miller, 1995) is a database mapping word forms to word senses (or *synsets*) and encoding lexical relations between them such as synonymy, antonymy, hyponymy/hypernymy, etc. WordNet is available for 20+ languages, among which the largest is the English Wordnet, consisting of over 150,000 word forms and nearly 120,000 synsets. In Chapter 6 of this thesis we shall describe multiple systems that use WordNet as one of their resources for extracting lexical relations between words.

2.3 Montague-style theories

A considerable amount of the literature on the semantics of natural language has in the past few decades focused on *Montagovian* representations of meaning (Montague, 1970a, 1970b, 1973; Kamp, 1981; Groenendijk & Stokhof, 1991). The shared agenda of these approaches is to provide a mapping from linguistic structures to logical formulae; the bulk of actual work is concerned with handling particular portions of syntax. Nearly all such accounts take Montague's original treatment of word meaning for a given. It has been shown that at least 84 percent of the information content of an average utterance is encoded by word meaning (Kornai, 2012), yet most proposed interpretations of sentences such as *Every man loves a woman such that she loves him* rarely have anything to say about the concepts `man, woman`, or `love`. There are some generic principles of how word meaning should be represented in logical formulae: nouns like *man* are typically thought of as functions that decide for all objects of the world whether they are men or not, verbs like *love* are thought of as describing an event in a way that for any event in the world one can decide whether an act of loving has taken place. Such principles have little practical value, however, when linking particular utterances to states-of-affairs. To our knowledge, no lexicon with a substantive list of meaning postulates has ever been built. In Chapter 5 we shall construct `4lang`-style meaning representations for all headwords of monolingual dictionaries of English.

If common nouns like *giraffe* and adjectives like *blue* are both seen as selecting a subset of all objects in the world, then an NP such as *blue giraffe* might map to the intersection of these subsets. The same mechanism fails for *enormous fleas*: the representation of *enormous* must be updated to accommodate the fact that you cannot tell if some size is enormous unless you know whose size it is (e.g. half an inch is enormous for a flea but tiny for a giraffe). Clearly there does not exist a function that selects a universal set of enormous fleas – what constitutes large may depend e.g. on the speaker's previous experience. Yet if we are to account for the fact that people can use this phrase successfully in conversations, we must map *enormous* to some function that might take as its parameter not only an entry encoding shared beliefs of speakers about defining properties of fleas, but also some information regarding their beliefs of the size of fleas. It is tempting to handle such a phenomenon by simply defining the interfaces with *extra-linguistic* knowledge, after which the meaning of *small blue giraffe* can be a formula with parameters for speakers' knowledge of what size range counts as small for a giraffe, what shades of color counts as blue, perhaps even what set of characteristics would make something/somebody a giraffe. Travis (1997) describes this approach in *A Companion to the Philosophy of Language*:

> What some words say, or contribute to what is said in using them, varies across speakings of them. Where this is so, the meaning of the words does two things. First, it determines on just what facts about a speaking the semantic contribution of the words so spoken depends. Second, it determines just how their semantics on a speaking depends on these facts. Specifically, it determines a specifiable function from values of those factors to the semantics the words would have, if spoken where those values obtain. (Travis, 1997, p.92)

Proponents of Montagovian theories of semantics may claim that the subject of their study (*meaning* in a narrow sense) is the component of the effect an utterance has on the information state of speakers that is unchanged across "speakings". Nevertheless, such a representation of e.g. *small blue giraffe* must contain information about the meaning of each of the individual concepts `small, blue,` and `giraffe`. It is one thing to disown the issue of inter-speaker variation on which colors are blue, what sizes of giraffes are small, etc., but surely what makes the phrase more informative than e.g. *small blue animal* is that the variation among all giraffes is considerably smaller than the variation among all animals. That MG accounts of semantics do not decompose the meaning of content words is problematic because we have seen that to construct the meaning of even the simplest kinds of phrases, one needs to account for how their meanings interact. Any mechanism with a chance to interpret *small giraffe* or *young giraffe* will have to make reference to a particular set of components of the meaning of `giraffe`, otherwise we cannot make predictions about the size or age of the giraffe. The necessity of decomposing word meaning has already been argued for by (Katz & Fodor, 1963), but the actual use of meaning postulates in MG remains restricted to the resolution of technical problems caused by handling intensionality; for a survey, see (Zimmermann, 1999). In Chapter 3 we shall present a theory of meaning representation that encodes word meaning as a network of concepts, making them accessible to mechanisms responsible for constructing the meaning of larger structures.

2.4 CVS representations

The most widely used models of word meaning today are continuous vector spaces (CVS). State-of-the-art systems in most standard NLP tasks rely on *word embeddings*, mappings from words of a language to real-valued vectors, trained on datasets containing 10^6-10^{10} words. In this section we review key aspects of CVS semantics, which set the current standard for representing word meaning (cf. Section 2.4.1). Remarkably, they do so using elements of representations that – unlike `4lang` representations – do not lend themselves to compositionality in any obvious way (cf. Section 2.4.2).

2.4.1 Vectors as word representations

Methods used to obtain mappings from words to vectors are based on the *distributional hypothesis* (Harris, 1954), which states that words are similar if they appear in similar contexts. When training word embeddings on large bodies of unannotated text, the most commonly used algorithms (Mikolov, Chen, et al., 2013; Pennington et al., 2014) will take into account all contexts the word has occurred in (typically some fixed-size sequence of surrounding words) and attempt to find vectors for each word that minimizes the difference between the predicted and observed probability of the word appearing in those contexts. Embeddings trained this way can be evaluated by using them as the initial layers of neural network models trained for a variety of NLP tasks such as named entity recognition, chunking, POS-tagging, etc. (Collobert & Weston, 2008; Turian et al., 2010). Word vectors are also often measured by their direct applicability to particular tasks such as answering word analogy questions (Mikolov, Yih, & Zweig, 2013) or finding missing words in text (*cloze test*) (Zweig et al., 2012). Analogical questions such as “man is to woman as king is to X” can be answered successfully by taking the vectors associated with each word ($\vec{m}$, $\vec{w}$, $\vec{k}$ for man, woman, and king, respectively) and finding the word whose vector has the greatest cosine similarity to $\vec{k} + \vec{w} - \vec{m}$. The fact that this strategy is relatively successful indicates that the relational hypothesis holds to some extent:

word representations trained based on distribution are at least implicitly related to word meaning, making them candidates for use in computational semantics systems. Indeed, word embeddings have been used successfully in state of the art systems for e.g. Semantic Role Labeling (Foland Jr & Martin, 2015), Knowledge Base Construction (Nickel et al., 2015), and Semantic Textual Similarity (Han et al., 2015). Vector representations are also practical for establishing a connection between linguistic and non-linguistic data, a striking indication is the work presented in (Karpathy et al., 2014), mapping text fragments to pictures for information retrieval (image search).

2.4.2 Vectors beyond the word level

In this section we mention only a few examples that are relevant to our thesis. For a generic overview of compositionality in CVS semantics, the reader is referred to Section 2 of (Grefenstette & Sadrzadeh, 2015). An example of training vectors that represent linguistic units larger than a single word is the Compositional Vector Grammar (CVG) parser introduced in (Socher, Bauer, et al., 2013), which outperforms by a significant margin the state of the art in syntactic parsing by combining the standard PCFG approach with recursive neural networks (RNNs) trained on each layer of a parse tree, assigning vectors not only to words but all nonterminals of the grammar. The `text_to_4lang` system introduced in Chapter 4 relies on CVGs for syntactic parsing, therefore we now provide a very brief overview of them as presented in (Socher, Bauer, et al., 2013).

PCFG parsers such as that implemented by the Stanford Parser will return for some input sentence a ranked list of candidate parses. If a grammar is able to generate the correct parse tree for nearly all sentences, i.e. the correct parse can be expected to be among the candidates returned for some sentence, then increasing parsing accuracy amounts to improving the component responsible for ranking candidates based on their likelihood. CVGs combine the power of PCFGs and RNNs by devising a method to rerank parse trees in the output of a standard PCFG parser using neural networks trained on a treebank. The core idea is that in calculating the score of a given syntactic derivation (parse tree) for a sentence, the likelihood of each derivation step should be assigned based on not only the observed frequency of the given structure, but rather its likeliness to cover the particular sequence of words, and that this calculation should factor in word forms via a distributional model, approximating the properties of rare or unseen words using more frequent ones that appear in similar contexts. *Syntactically untied* networks (SU-RNNs) learn separate parameters for each rewrite rule. The parameters for a rule of the form $A \to BC$ are encoded by the *syntactic triplet* $((A, a), (B, b), (C, c))$, where b and c are vectors of R^n assigned to the non-terminals B and C, respectively, and A is computed as

$f(W^{(B,C)}([b, c]))$, where $[b, c]$ is a vector in R^{2n} obtained by concatenating b and c, and $W^{B,C}$ is a matrix in $R^{n \times 2n}$ which is learnt during the training process. f is the element-wise nonlinearity function tanh. The process is summarized in Figure 2.13. This process allows the parser to rank competing parse trees based on a likelihood that is sensitive to the distribution of individual words as observed in data that is orders of magnitude larger than those available for training the PCFG parser.

Compositionality of word vectors has also been explored in the context of Sentiment Analysis (Socher, Perelygin, et al., 2013; Zhu et al., 2015) and Semantic Textual Similarity (Sultan et al., 2015). The latter work assigns vectors to sentences by calculating the componentwise average of all word vectors. Socher, Perelygin, et al. (2013) use *Recursive Neural Tensor Networks* (RNTNs) to obtain vectors for each node in the parse tree of a sentence.

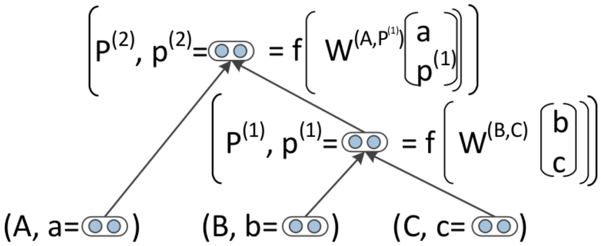

(P(2), p(2)= = f(W(A,P(1)) [a p(1)]))
(P(1), p(1)= = f(W(B,C) [b c]))
(A, a=)
(B, b=)
(C, c=)

Figure 2.13: Example of a syntactically untied RNN (Socher, Bauer, et al., 2013, p.459)

Chapter 3

The 4lang system

This chapter describes the 4lang system for representing meaning using directed graphs of concepts. Since the underlying theory is not the main contribution of this thesis, but rather the work of half a dozen researchers over the course of 6 years, we shall not attempt a full presentation of the 4lang principles. Instead we shall introduce the formalism in Section 3.1, then continue to discuss some specific aspects relevant to this thesis. 4lang's approach to multiple word senses is summarized in Section 3.2, Section 3.3 is concerned with reasoning based on 4lang graphs. Treatment of extra-linguistic knowledge is discussed in Section 3.4. Finally, Section 3.5 considers the primitives of the 4lang representation and contrasts them with some earlier approaches mentioned in Chapter 2.

For a complete presentation of the theory of lexical semantics underlying 4lang the reader is referred to (Kornai, 2010) and (Kornai, 2012). (Kornai et al., 2015) compares 4lang to contemporary theories of word meaning. 4lang is also the name of a manually built dictionary[1] mapping 2,200 English words to concept graphs (as well as their translations in Hungarian, Polish, and Latin, hence its name). The dictionary is described in (Kornai & Makrai, 2013). For work on extending 4lang to include the top 40 languages (by Wikipedia size), see (Ács et al., 2013).

3.1 The formalism

4lang represents the meaning of words, phrases and utterances as directed graphs whose nodes correspond to language-independent concepts and whose edges may have one of three labels, based on which they'll be referred to as 0-edges, 1-edges, and 2-edges. (The 4lang theory represents concepts as Eilenberg-machines (Eilenberg, 1974) with three *partitions*, each of which may contain zero or more pointers to other machines and therefore also

[1] https://github.com/kornai/4lang/blob/master/4lang

represent a directed graph with three types of edges. The additional capabilities offered by Eilenberg-machines have not so far been applied by the author, some of them have not even been implemented yet, therefore it makes more sense to consider the representations under discussion as plain directed graphs.) First we shall discuss the nature of `4lang` *concepts* - represented by the nodes of the graph, then we'll introduce the types of relationships encoded by each of the three edge types.

3.1.1 Nodes

Nodes of `4lang` graphs correspond to *concepts*. `4lang` concepts are not words, nor do they have any grammatical attributes such as part-of-speech (category), number, tense, mood, voice, etc. For example, `4lang` representations make no difference between the meaning of *freeze* (N), *freeze* (V), *freezing*, or *frozen*. Therefore, the mapping between words of some language and the language-independent set of `4lang` concepts is a many-to-one relation. In particular, many concepts will be defined by a single link to another concept that is its hypernym or synonym, e.g. $\texttt{above} \xrightarrow{0} \texttt{up}$ or $\texttt{grasp} \xrightarrow{0} \texttt{catch}$. Encyclopedic information is omitted, e.g. `Canada`, `Denmark`, and `Egypt` are all defined as `country` (their definitions also containing a pointer to an external resource, typically to Wikipedia). In general, definitions are limited to what can be considered the shared knowledge of competent speakers – e.g. the definition of `water` contains the information that it is a colorless, tasteless, odorless liquid, but not that it is made up of hydrogen and oxygen. We shall now go through the types of links used in `4lang` graphs.

3.1.2 The 0-edge

The most common relation between concepts in `4lang` graphs is the 0-edge, which represents attribution ($\texttt{dog} \xrightarrow{0} \texttt{friendly}$); the `IS_A` relation (hypernymy) ($\texttt{dog} \xrightarrow{0} \texttt{animal}$); and unary predication ($\texttt{dog} \xrightarrow{0} \texttt{bark}$). Since concepts do not have grammatical categories, this uniform treatment means that the same graph can be used to encode the meaning of phrases like *water freezes* and *frozen water*, both of which would be represented as $\texttt{water} \xrightarrow{0} \texttt{freeze}$.

3.1.3 1- and 2-edges

Edge types 1 and 2 connect binary predicates to their arguments, e.g. $\texttt{cat} \xleftarrow{1} \texttt{catch} \xrightarrow{2} \texttt{mouse}$). The formalism used in the `4lang` dictionary explicitly marks binary (transitive) elements – by using UPPERCASE printnames. The pipeline that we'll introduce in Chapter 4 will

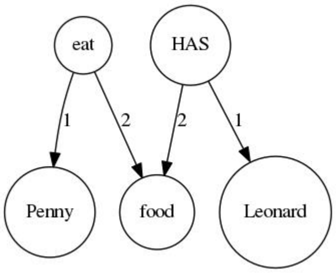


Figure 3.1: 4lang graph with two types of binaries.

HAS	shirt $\xleftarrow{1}$ HAS $\xrightarrow{2}$ collar
IN	letter $\xleftarrow{1}$ IN $\xrightarrow{2}$ envelope
AT	bridge $\xleftarrow{1}$ AT $\xrightarrow{2}$ river
CAUSE	humor $\xleftarrow{1}$ CAUSE $\xrightarrow{2}$ laugh
INSTRUMENT	sew $\xleftarrow{1}$ INSTRUMENT $\xrightarrow{2}$ needle
PART_OF	leaf $\xleftarrow{1}$ PART_OF $\xrightarrow{2}$ plant
ON	smile $\xleftarrow{1}$ ON $\xrightarrow{2}$ face
ER	slow $\xleftarrow{1}$ ER $\xrightarrow{2}$ speed
FOLLOW	Friday $\xleftarrow{1}$ FOLLOW $\xrightarrow{2}$ Thursday
MAKE	bee $\xleftarrow{1}$ MAKE $\xrightarrow{2}$ honey

Table 3.1: Most common binaries in the 4lang dictionary

not make use of this distinction, any concept can have outgoing 1- and 2-edges. Binaries marked with uppercase are nevertheless clearly set apart from other concepts by the fact that they are *necessarily* binary, i.e. they must always have exactly two outgoing edges. We retain the uppercase marking for those binary elements that do not correspond to any word in a given phrase or sentence, e.g. the meaning of the sentence *Penny ate Leonard's food* will be represented by the graph in Figure 3.1[2]. The top ten most common binaries used in 4lang are listed in Table 3.1 and examples are shown for each.

Given two concepts c_1 and c_2 such that c_2 is a predicate that holds for c_1, 4lang will allow for one of two possible connections between them: $c_1 \xrightarrow{0} c_2$ if c_2 is a one-place predicate and $c_2 \xrightarrow{1} c_1$ if c_2 is a two-place predicate. It would be counter-intuitive and unpractical to treat these configurations as mutually exclusive in the 4lang-based

[2] Evidence for different patterns of linking predicates and their arguments could be obtained from ergative languages (Dixon, 1994), these shall not be discussed here.

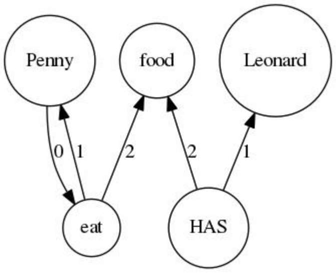


Figure 3.2: Revised `4lang` graph with two types of binaries for the sentence *Penny ate Leonard's food*

systems presented in this thesis. Two-place predicates often appear with a single argument (e.g. *John is eating*), and representing such a statement as `John` $\xrightarrow{0}$ `eat` while the sentence *John is eating a muffin* warrants `John` $\xleftarrow{1}$ `eat` $\xrightarrow{2}$ `muffin` would mean that we consider the relationship between `John` and `eat` dependent on whether we have established the object of his eating. Therefore we choose to adopt a modified version of the `4lang` representation where the 0-connection holds between a subject and predicate regardless of whether the predicate has another argument. The example graph in Figure 3.1 can then be revised to obtain that in Figure 3.2[3].

The meaning of each `4lang` concept is represented as a `4lang` graph over other concepts – a typical definition in the `4lang` dictionary can be seen in Figure 3.3; this graph captures the facts that birds are vertebrates, that they lay eggs, and that they have feathers and wings. The generic applicability of the `4lang` relations introduced in Section 3.1 have the consequence that to create, understand, and manipulate `4lang` representations one need not make the traditional distinction between entities, properties, and events. The relationships `dog` $\xrightarrow{0}$ `bark` and `dog` $\xrightarrow{0}$ `inferior` (Kornai, in preparation) can be treated in a uniform fashion, when making inferences based on the definitions of each concept, e.g. that `dog` $\xleftarrow{1}$ `MAKE` $\xrightarrow{2}$ `sound` or that calling another person a *dog* is insulting.

3.2 Ambiguity and compositionality

`4lang` does not allow for multiple senses when representing word meaning, all occurrences of the same word form – with the exception of true homonyms like *trunk* 'the very long

[3] Since the `text_to_4lang` pipeline presented in Chapter 4 assigns `4lang` graphs to raw text based on the output of dependency parsers that treat uniformly the relationship between a subject and verb irrespective of whether the verb is transitive or not, the `4lang` graphs we build will include a 1-edge between all verbs and their subjects. We do not consider this a shortcoming: for the purposes of semantic analysis we do not see the practicality of a distinction between transitive and intransitive verbs – we only recognize the difference between the likelihood (based on data) of some verb taking a certain number of arguments.

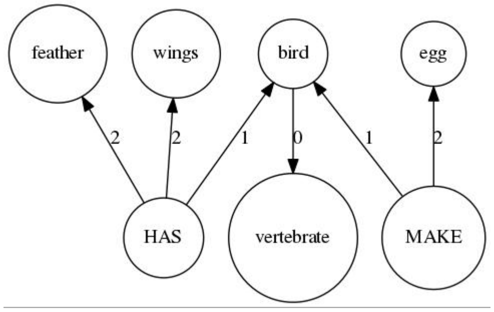


Figure 3.3: 4lang definition of `bird`.

nose of an elephant' and *trunk* 'the part at the back of a car where you can put bags, tools etc'[4] – must be mapped to the same concept, whose definition in turn must be generic enough to allow for all possible uses of the word. As Jakobson famously noted, such a monosemic approach might define the word *bachelor* as 'unfulfilled in typical male role' (Fillmore, 1977). Such definitions place a great burden on the process responsible for combining the meaning of words to create representations of phrases and utterances (see Chapter 4), but it has the potential to model the flexibility and creativity of language use:

> "we note here a significant advantage of the monosemic approach, namely that it makes interesting predictions about novel usage, while the predictions of the polysemic approach border on the trivial. To stay with the example, it is possible to envision novel usage of *bachelor* to denote a contestant in a game who wins by default (because no opponent could be found in the same weight class or the opponent was a no-show). The polysemic theory would predict that not just seals but maybe also penguins without a mate may be termed bachelor – true but not very revealing."(Kornai, 2010, p.182)

One typical consequence of this approach is that `4lang` definitions will not distinguish between `bachelor` and some concept `w` that means 'unfulfilled male' – both could be defined in `4lang` as e.g. `male, LACK`. This is not a shortcoming of the representation, rather it is in accordance with the principles underlying it; the concepts `unfulfilled` and `male` cannot be combined (e.g. to create a representation describing an *unfulfilled male*) without making reference to some nodes of the graph representing the meaning of `male`; if something is a 'typical male role', this should be indicated in the definition graph of `male` (if only by inbound pointers), and without any such information, *unfulfilled male* cannot be interpreted at all.

[4] All example definitions, unless otherwise indicated, are taken from the Longman Dictionary of Contemporary English (Bullon, 2003)

This does not mean that `male` cannot be defined without listing all stereotypes associated with the concept. If the piece of information that ‘being with a mate at breeding time’ is a typical male role – which is necessary to account for the interpretation of *bachelor* as ‘young fur seal when without a mate at breeding time’ – is to be accessed by some inference mechanism, then it must be present in the form of some subgraph containing the nodes `seal`, `mate`, `male`, and possibly others. Then, a `4lang`-based natural language understanding system that is presented with the word *bachelor* in the context of mating seals for the first time may explore the neighborhood of these nodes until it finds this piece of information as the only one that ‘makes sense’ of this novel use of *bachelor*. Note that this is a model of novel language use in general. Humans produce and understand without much difficulty novel phrases that most theories would label ‘semantically anomalous’. In particular, all language use that is commonly labeled *metaphoric* involves accessing a lexical element for the purpose of activating some of its meaning components, while ignoring others completely. It is this use of language that `4lang` wishes to model, as it is most typical of everyday communication (Richards, 1937; Wilks, 1978; Hobbs, 1990). When we present the `dict_to_4lang` system for building `4lang` definitions from dictionary entries, we shall discuss the possibility of gathering information from multiple definitions of a single headword (see Section 5.4.3).

Another `4lang` principle that facilitates metaphoric interpretation is that any link in a `4lang` definition can be overridden. In fact, the only type of negation used in `4lang` definitions, `LACK`, carries the potential to override elements that might otherwise be activated when definitions are expanded: e.g. the definition of `penguin`, which undoubtedly contains $\xrightarrow{0}$ `bird`, may also contain $\xleftarrow{1}$ `LACK` $\xrightarrow{2}$ `fly` to block inference based on `bird` $\xrightarrow{0}$ `fly`. That any element can freely be overridden ensures that novel language use does not necessarily cause contradiction: “[T]o handle ‘the ship plowed through the sea’, one lifts the restriction on ‘plow’ that the medium be earth and keeps the property that the motion is in a substantially straight line through some medium” (Hobbs, 1990, p.55). Since a `4lang` definition of `plow` must contain some version of $\xrightarrow{2}$ `earth`, there must be a mechanism allowing to override it and not make inferences such as `sea` $\xrightarrow{0}$ `earth`[5].

[5] Note that such an inference must access some form of world knowledge in addition to the definition of each concept: the definition of `ship` will contain $\xleftarrow{1}$ `ON` $\xrightarrow{2}$ `water` (or similar), but to infer that this makes it incompatible with the `earth` in the definition of `plow` one must also be aware that water and earth cancel each other out in the context of where a vehicle runs

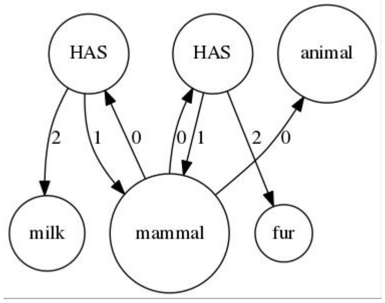


Figure 3.4: 4lang definition of mammal.

3.3 Reasoning

The 4lang principles summarized so far place a considerable burden on the inferencing mechanism. Given the possibility of defining all concepts using only a small set of primitives, and a formalism that strictly limits the variety of connections between concepts, we claim to have laid the groundwork for a semantic engine with the chance of understanding creative language use. Generic reasoning has not yet been implemented in 4lang, we only present early attempts in Section 5.3 and some specific applications in Chapter 6. Here we shall simply outline what we believe could be the main mechanisms of such a system.

The simplest kind of lexical inference in 4lang graphs is performed by following paths of 0-edges from some concept to determine the relationships in which it takes part. The concept mammal is defined in 4lang as an animal that has fur and milk (see Figure 3.4), from which one can conclude that the relations $\xleftarrow{1}$ HAS $\xrightarrow{2}$ milk and $\xleftarrow{1}$ HAS $\xrightarrow{2}$ fur also hold for all concepts whose definition includes $\xrightarrow{0}$ mammal (we shall assume that this simple inference can be made when we construct 4lang definitions from dictionary definitions in Chapter 5). Similar inferences can be made after *expanding* definitions, i.e. replacing concept nodes with their definition graphs (see Section 5.3 for details). If the definition of giraffe contains $\xrightarrow{0}$ mammal, to which we add edges $\xleftarrow{1}$ HAS $\xrightarrow{2}$ fur and $\xleftarrow{1}$ HAS $\xrightarrow{2}$ milk, this expanded graph will allow us to infer the relations giraffe $\xleftarrow{1}$ HAS $\xrightarrow{2}$ fur and giraffe $\xleftarrow{1}$ HAS $\xrightarrow{2}$ milk. As mentioned in the previous section, this process requires that relations present explicitly in a definition override those obtained by inference: penguins are birds and yet they cannot fly, humans are mammals without fur, etc.

A more complicated procedure is necessary to detect connections between nodes of an expanded definition and nodes connected to the original concept. Recall Quillian's example in Section 2.2.1: given the phrase *lawyer's client* his iterative search process will eventually find lawyer to be compatible with the employer property of client, since both are professionals. A similar process can be implemented for 4lang graphs; consider the definition graphs for lawyer and client in Figures 3.5 and 3.6, built automatically from

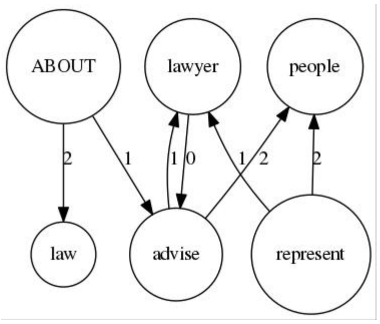


Figure 3.5: Definition graph for `lawyer`

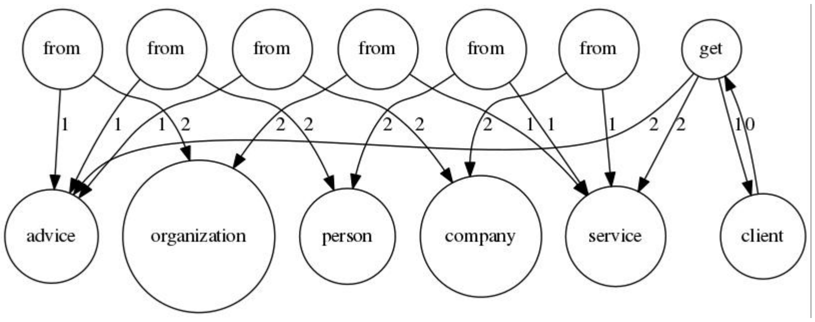


Figure 3.6: Definition graph for `client`

definitions in the Longman dictionary, as described in Chapter 5, then pruned manually. (These graphs, being the output of the `dict_to_4lang` system and not manual annotation, have numerous issues: the word *people* in the Longman dictionary definition of `lawyer` was not mapped to `person`, nor have the words *advice* and *advise* been mapped to the same concept. After correcting these errors manually, nodes with identical names in the graph for *lawyer's client* (Figure 3.7) can form the starting point of the inference process.) Let us now go over the various steps of inference necessary to reduce this graph to the most informative representation of *lawyer's client*. Note that we do not wish to impose any logical order on these steps; they should rather be the 'winners' of a process that considers many transformations in parallel and ends up keeping only some of them. A simple example of such a system will be described in Section 6.3.

We should be able to realize that the `person` who is `advice`d (and is `represent`ed by) the `lawyer` can be the same as the `client` who `get`s `advice` from the lawyer. To this end we must be able to make the inference that $\texttt{X} \overset{1}{\leftarrow} \texttt{get} \overset{2}{\rightarrow} \texttt{advice}$ and $\texttt{advice} \overset{2}{\rightarrow} \texttt{X}$ are synonymous. We believe a `4lang`-based system should be able to make such an inference in at least one of two independent ways. First, we expect our inference mechanism to compute, based on the definitions of `get` and `advice`, that $\texttt{X} \overset{1}{\leftarrow} \texttt{get} \overset{2}{\rightarrow} \texttt{advice}$ entails $\texttt{advice} \overset{2}{\rightarrow} \texttt{X}$ (and vice versa). Secondly, we'd like to be able to accommodate *construc-*

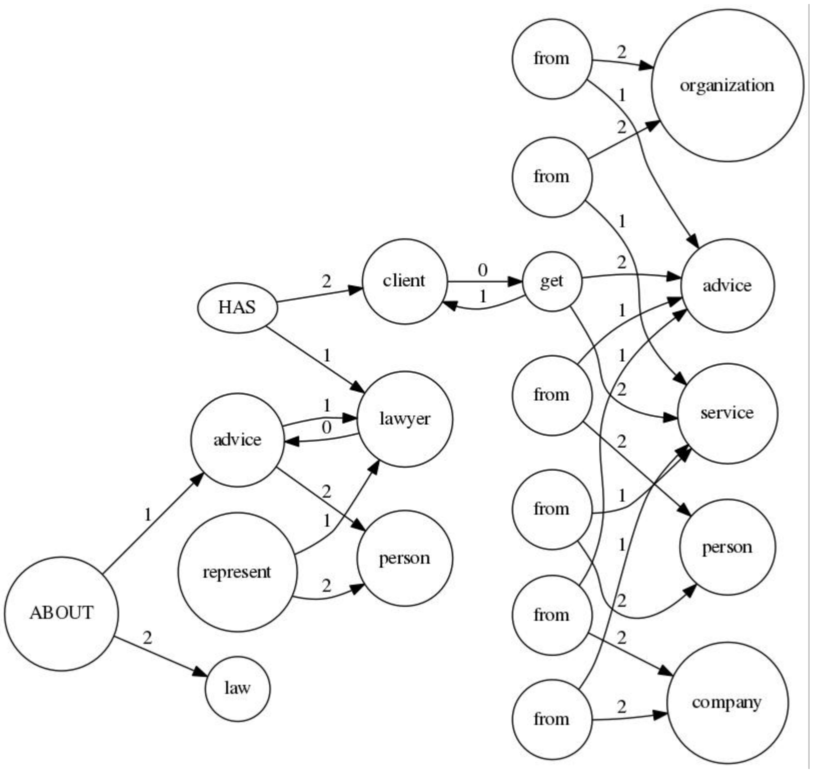


Figure 3.7: Corrected graph for *lawyer's client*

tions in the `4lang` system (see also Section 8.4) that may explicitly pair the above two configurations for some concepts but not for others (e.g. $\texttt{X} \xleftarrow{1} \texttt{get} \xrightarrow{2} \texttt{drink}$ should not trigger $\texttt{drink} \xrightarrow{2} \texttt{X}$).

We should also consider unifying the `person` node in $\texttt{person} \xleftarrow{1} \texttt{from} \xrightarrow{2} \texttt{advice}$ with `lawyer` in $\texttt{advice} \xrightarrow{1} \texttt{lawyer}$, which would once again require either some construction that states that when someone *advises*, then the *advice* is *from* her, or a generic rule that can guess the same connection. Given these inferences, the two `advice` can also be merged as likely referring to the same action, resulting in the final graph in Figure 3.8. The nodes `organization`, `company`, and `service` have been omitted from the figure to improve readability.

3.4 Extra-linguistic knowledge

Chapter 3 of (Kornai, in preparation) argues that knowledge representation for the purposes of natural language understanding requires a distinction between analytic and synthetic knowledge, and that the `4lang` theory is adequate to represent all analytic knowledge. When we discuss inference in terms of `4lang` representations, we only make reference

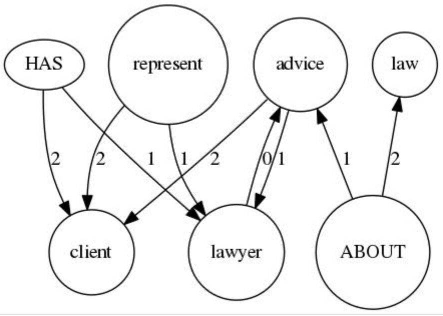


Figure 3.8: Inferred graph for *lawyer's client*

to knowledge that is clearly within the boundaries of the naive theories described by Kornai. We emphasize that we do not even need to establish any particular piece of knowledge as essential to our inferencing capabilities, just as in mathematics, where we do not need to establish the truth of the axioms. Returning to one of the simplest examples above, where `bird` $\xrightarrow{0}$ `fly` is overridden to accommodate both `penguin` $\xleftarrow{1}$ `LACK` $\xrightarrow{2}$ `fly` and `penguin` $\xrightarrow{0}$ `bird`, we need not decide whether the particular piece of information that penguins cannot fly is part of the meaning of `penguin`. Clearly it is possible for one to learn of the existence of penguins and that they are a type of bird without realizing that they cannot fly, and this person could easily make the incorrect *inference* that they can. Some components of word meaning, on the other hand, appear to be essential to the understanding of a particular concept, e.g. if a learner of English believes that *nephew* refers to the child of one's sibling, male or female (perhaps because in her native language a single word stands for both nephews and nieces, and because she has heard no contradicting examples), we say that she does not know the meaning of the word; `nephew` $\xrightarrow{0}$ `male` is internal to the concept `nephew` in a way that `penguin` $\xleftarrow{1}$ `LACK` $\xrightarrow{2}$ `fly` is not to `penguin`. This distinction is commonly made in semantics under the heading analytic vs. synthetic knowledge, but imperfections in acquiring analytic knowledge are common and a normal part of the language acquisition process. Carrying a conversation successfully only requires that the participants' representations of word meaning does not contradict each other in a way *relevant to the conversation at hand*[6]. Static lexical resources such as LDOCE or the `4lang` concept dictionary must make decisions about which pieces of information to include, and may do so based on some notion of how 'technical' or 'commonplace' they are. A person's ignorance of the fact that somebody's nephew is necessarily male is probably itself the result of one or several conversations about nephews that somehow remained

[6] This is also reflected in The Urban Dictionary's definition of *semantics: The study of discussing the meaning/interpretation of words or groups of words within a certain context; usually in order to win some form of argument* (http://www.urbandictionary.com)

consistent despite his incomplete knowledge about how the word is typically used.

3.5 Primitives of representation

In the following two chapters this thesis will present methods for 1) building 4lang representations from raw text and 2) building 4lang definition graphs for virtually all words based on monolingual dictionaries. Given these two applications, any text can be mapped to 4lang graphs and nodes of any graph can be expanded to include their 4lang definitions. Performing this expansion iteratively, all representations can be traced back to a small set of concepts. In case the Longman Dictionary is used to build definition graphs, the concepts listed in the 4lang dictionary will suffice to cover all of them, since it contains all words of the Longman Defining Vocabulary (LDV), the set of all words used in definitions of the Longman Dictionary (Boguraev & Briscoe, 1989). The set of concepts necessary to define all others can be further reduced: we show in (Kornai et al., 2015) that as few as 129 4lang concepts are enough to define all others in the 4lang dictionary, and thus, via monolingual dictionaries, practically all words in the English language.

In response to Katz and Fodor's markers and distinguishers (see Section 2.1), Bolinger (1965) argues that any component of word meaning that Katz and Fodor may consider to belong to the domain of distinguishers, and as such out of grasp for a semantic theory, can be further decomposed into markers. He demonstrates his point by providing example uses of the word *bachelor* that allow a competent speaker to disambiguate between the senses listed by Katz and Fodor, but only based on properties of senses that are below the last marker in K&F's decomposition (cf. Figure 2.1). Since each of these examples is a self-contained argument for the existence of some semantic category, we shall use some of them to demonstrate 4lang's ability to decompose meaning. In Figure 3.9 we present Bolinger's first five examples along with his original explanation of how each necessitates the introduction of some semantic marker.

Our account of these examples will be incomplete given the current limitations of our implemented systems, e.g. its current lack of treatment for modality, negation, and temporal relations. These already concern the first example: what is implemented of 4lang so far does not have a sophisticated system for representing temporal relations. The concepts after and before are used in 4lang definitions to encode event structure, e.g. the definition of discover contains know $\xrightarrow{0}$ after and effort $\xrightarrow{0}$ before. Whether the inference indicated by Bolinger can be made depends on how the definition of marry (Figure 3.10) is negated – given proper treatment, a man who *has never married* will be established as one for whom (before $\xleftarrow{0}$) marriage $\xleftarrow{2}$ IN $\xrightarrow{0}$ NOT holds, and become

1. *He became a bachelor.* This rules out the 'man who has never married' – it is impossible to become one who has never done something. We can extract the -ever part of never from the distinguisher and set up a marker (Nonbecoming).
2. *The seven-year-old bachelor sat on the rock.* The definition 'male who has never married' was deficient. It should have been something like 'adult male who has never married,' and from that expanded distinguisher we now extract the marker (Adult).
3. *Lancelot was the unhappiest of all the bachelors after his wife died.* This seems to justify raising (Unmarried) to marker status and wipes out the distinguisher on one of the branches: *bachelor*-noun-(Human)–(Male)-(Adult)-(Non-becoming)-(Unmarried).
4. *That peasant is a happy bachelor.* Being a peasant is not compatible with being a knight. There must be a marker of status lying around somewhere. A knight has to be of gentle birth. Let us extract (Noble) from the distinguisher (leaving the degree of nobility for the moment undisturbed as still part of the knight's distinguisher).
5. *George is one bachelor who is his own boss.* This eliminates the knight, and turns 'serving under' into another status marker that might be called (Dependent).

Figure 3.9: Examples and arguments for new markers (Bolinger, 1965, p.558-560)

should entail that for some predicate $\xrightarrow{0}$ `before` is false, rendering it incompatible with the *unmarried man* interpretation of `bachelor`.

Example 2 requires us to derive the incompatibility of `adult` with `7-year-old`. Since the definitions of *adult* in both Longman and `en.wiktionary` contain the term *fully grown*, this inference requires us to make reference to knowledge about the average age at which humans stop growing. The third example can be handled in `4lang` similarly to the first: the *unmarried (adult) male* reading of `bachelor` must entail that at no time in the past could $\xleftarrow{1}$ `IN` $\xrightarrow{2}$ `marriage` have been true. Example 4 requires a contradiction to be detected between `knight` and `peasant` – this can be straightforward given the right definition, but given our method of building definitions from dictionary definitions, we cannot expect our definition graphs to be as comprehensive as to include `noble` and `noble` $\xrightarrow{0}$ `LACK` in the respective graphs for `knight` and `peasant`. Instead we should be able to infer these relations from the definitions we do encounter: the Longman definition of *knight*: *'a man with a high rank in the past who was trained to fight while riding a horse'* should result in the subgraph `knight` $\xleftarrow{1}$ `HAS` $\xrightarrow{2}$ `rank` $\xrightarrow{0}$ `high`[7], the definition of `peasant`: *a poor farmer*

[7] Incidentally, to construct this graph we would also need to overcome a parsing error: the Stanford

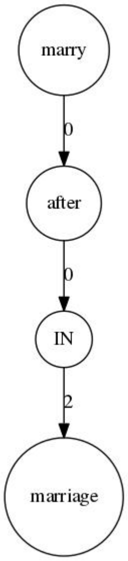


Figure 3.10: `4lang` definition of `marry`.

who owns or rents a small amount of land, either in past times or in poor countries will yield `peasant` $\xrightarrow{0}$ `poor`. These relations are not strictly incompatible, the original example also depends upon the assumption that being a peasant entails being of low rank – we have much better chances given a definition that makes this assumption itself, such as the one in the English Wiktionary: *A member of the lowly social class which toils on the land (...)*. In the latter case, all that remains is making the connection between `rank` and `class` ($\xrightarrow{0}$ `social`), but the former should also allow us, given a probabilistic system, to establish that a peasant is not likely to be a knight.

Finally, in Example 5, it is the incompatibility of 'being one's own boss' and the 'serving under' component of the *young knight serving under the standard of another knight* that must be established. The Longman definition of `boss`: *the person who employs you or who is in charge of you at work* will allow us to map *George is his own boss* to `George` $\xleftarrow[2]{1}$ `employ`, contradicting `George` $\xrightarrow{0}$ `serve` $\xleftarrow{1}$ `under` $\xrightarrow{2}$ `X` if the identity of `X` and `George` cannot be established, in this case explicitly excluded by the phrase *another knight*. We refrain from discussing the remaining 10 examples in (Bolinger, 1965). Details of the processes presented here are yet to be worked out, but we have shown that each inference is possible given our current set of semantic primitives.

Parser analyses this noun phrase as describing a man whose rank was trained and the rank is in the past. Parser errors such as this one will be discussed in Sections 4.4.1 and 8.4

3.6 Theoretical significance

This chapter provided a brief summary of the main principles behind the `4lang` system for representing the meaning of linguistic structures. Before we proceed to present a set of tools for building and manipulating `4lang` representations, as well as their applications to some tasks in computational semantics, let us point out some of the most important characteristics of `4lang` representations that make it our formalism of choice in the remainder of this thesis.

No categories `4lang` does not differentiate between concepts denoting actions, entities, attributes, etc., there are no categories of concepts equivalent to part-of-speech categories of words. This ensures, among other things, that words with a shared root are typically mapped to the same concept, and that ultimately utterances with the same information content can be mapped to inferentially identical `4lang` representations.

No polysemy `4lang` will only accommodate multiple senses of a word as a last resort. Distant but related uses of the same word must be interpreted via the same generic concept. This virtually eliminates the difficulty of word sense disambiguation.

Requires powerful inference The above principles require a mechanism for deriving all uses of a word from minimalistic definitions. Such a mechanism may stand a real chance at handling creative language use typical of everyday human communication (and responsible for polysemy in the first place).

No failure of interpretation No combinations of concepts and connections between them are forbidden by the formalism itself. Inference may judge certain states-of-affairs impossible, but the formalism will not fail the interpretation process.

Chapter 4

From text to concept graph

In this chapter we present our work on combining word representations like those described in Chapter 3 to create graphs that encode the meaning of phrases. We relegate the task of syntactic parsing to the state of the art Stanford Parser (DeMarneffe et al., 2006; Socher, Bauer, et al., 2013). The pipeline presented in this chapter processes sets of dependency triplets emitted by the Stanford Parser to create `4lang`-style graphs of concepts (our future plans to implement syntactic parsing in `4lang` are outlined in Section 8.4). This chapter is structured as follows: dependency parsing is briefly introduced in Section 4.1, the central `dep_to_4lang` module which maps dependencies to `4lang` graphs is presented in Sections 4.2 and 4.3. Major issues are discussed in Section 4.4, some solutions are presented in Section 4.5, manual evaluation of the `text_to_4lang` system is provided in Section 4.6. Finally, Section 4.7 presents the adaptation of `text_to_4lang` to Hungarian. The module presented in this chapter is accessible via the `text_to_4lang`[1] module of the `4lang` repository. Besides the ability to map chunks of running text to semantic representations, `text_to_4lang` will see another application that is crucial to the system described in this thesis: we process definitions of monolingual dictionaries to acquire word representations for lexical items that are not covered by `4lang`. The resulting module `dict_to_4lang` will be presented in Chapter 5. The modules `dep_to_4lang` and `dict_to_4lang` are also presented in (Recski, 2016), the adaptation to Hungarian is published in (Recski, Borbély, & Bolevácz, 2016).

[1] https://github.com/kornai/4lang/blob/master/src/text_to_4lang.py

4.1 Dependency parsing

We use a robust, state of the art tool, the Stanford Parser[2] to obtain dependency relations that hold between pairs of words in an English sentence. Unlike dependency parsers that have been trained on manually annotated dependency treebanks, the Stanford Parser discovers relations by matching templates against its parse of a sentence's constituent structure (DeMarneffe et al., 2006). This approach is more robust, since phrase structure parsers, and in particular the PCFG parser in the Stanford toolkit (Klein & Manning, 2003), are trained on much larger datasets than what is available to standard dependency parsers.

The Stanford Dependency Parser is also capable of returning *collapsed* dependencies, which explicitly encode relations between two words that are encoded in the sentence by a function word such as a preposition or conjunction. E.g. in case of the sentence *I saw the man who loves you*, standard dependency parses would contain the relation `nsubj(loves, who)` but not `nsubj(loves, man)`, even though *man* is clearly the subject of *loves*. Collapsed dependency parses contain these implicitly present dependencies and are therefore more useful for extracting the semantic relationships between words in the sentence. Furthermore, the Stanford Parser can postprocess *conjunct dependencies*: in the sentence *Bills on ports and immigration were submitted by Senator Brownback, Republican of Kansas*, the NP *Bills on ports and immigration* will at first be parsed into the relations `prep_on(Bills, ports)` and `cc_and(ports, immigration)`, then matched against a rule that adds the relation `prep_on(Bills, immigration)`. For our purposes we enable both types of postprocessing and use the resulting set of relations (or *triplets*) as input to the `dep_to_4lang` module, which uses them to build `4lang` graphs and will be introduced in Section 4.2.

The list of dependency relations extracted from a sentence (for a detailed description of each dependency relation see (De Marneffe & Manning, 2008a)) is clearly not intended as a representation of meaning; it will nevertheless suffice for constructing good quality semantic representations because of the nature of `4lang` relations: for sentences and phrases such as *Mary loves John* or *queen of France*, `4lang` representations are as simple as $\texttt{Mary} \xleftarrow{1} \texttt{love} \xrightarrow{2} \texttt{John}$ and $\texttt{France} \xleftarrow{1} \texttt{HAS} \xrightarrow{2} \texttt{queen}$ which can be straightforwardly constructed from the dependency relations `nsubj(love, Mary)`, `dobj(love, John)`, and `prep_of(queen, France)`. Any further details that one may demand of a semantic representation, e.g. that John is an experiencer or that France does not physically possess the queen, will be inferred from the `4lang` definitions of the concepts `love` and `queen`, in

[2] http://nlp.stanford.edu/software/lex-parser.shtml

the latter case also accessing the definitions of `rule` or `country`.

4.2 From dependencies to graphs

To construct `4lang` graphs using dependency relations in the parser's output, we created manually a mapping from relations to `4lang` subgraphs, assigning to each dependency one of nine possible configurations (see Table 4.1). Additionally, all remaining relations of the form `prep_*` and `prepc_*` are mapped to binary subgraphs containing a node corresponding to the given preposition. To map words to `4lang` concepts, we first lemmatize them using the `hunmorph` morphological analyzer and the `morphdb.en` database (Tron et al., 2005). Graph edges for each dependency are added between the nodes corresponding to the lemmas returned by `hunmorph`. The full mapping from dependencies to `4lang`-subgraphs is presented in Table 4.1. Figure 4.1 provides an example of how `4lang` subgraphs correspond to dependency triplets.

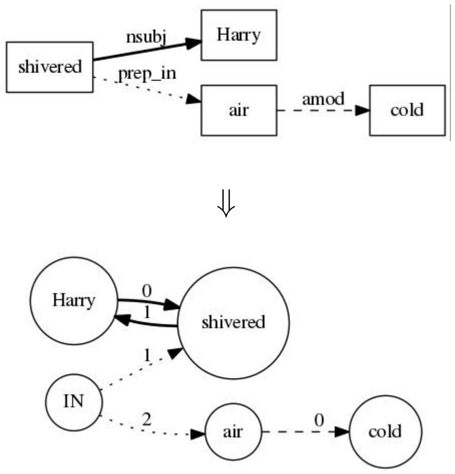


Figure 4.1: Constructing the graph for *Harry shivered in the cold air*

4.3 Utterances

Dependency relations obtained from multiple sentences can be used to update graphs over a single set of nodes, therefore the `text_to_4lang` pipeline presented in this chapter can be applied to documents of arbitrary size. Some of our preliminary experiments showed coreference resolution to be a significant challenge posed by processing several sentences into a single concept graph; we have therefore extended the `text_to_4lang` module to

Dependency	Edge
amod	$w_1 \xrightarrow{0} w_2$
advmod	
npadvmod	
acomp	
dep	
num	
prt	
nsubj	$w_1 \underset{0}{\overset{1}{\rightleftarrows}} w_2$
csubj	
xsubj	
agent	
dobj	$w_1 \xrightarrow{2} w_2$
pobj	
nsubjpass	
csubjpass	
pcomp	
xcomp	
poss	$w_2 \xleftarrow{1} \texttt{HAS} \xrightarrow{2} w_1$
prep_of	
tmod	$w_1 \xleftarrow{1} \texttt{AT} \xrightarrow{2} w_2$
prep_with	$w_1 \xleftarrow{1} \texttt{INSTRUMENT} \xrightarrow{2} w_2$
prep_without	$w_1 \xleftarrow{1} \texttt{LACK} \xrightarrow{2} w_2$
prep_P	$w_1 \xleftarrow{1} \texttt{P} \xrightarrow{2} w_2$

Table 4.1: Mapping from dependency relations to 4lang subgraphs

run the Stanford Coreference Resolution system (Lee et al., 2011) and use its output to unify nodes in the concept graphs. An example is shown in Figure 4.2.

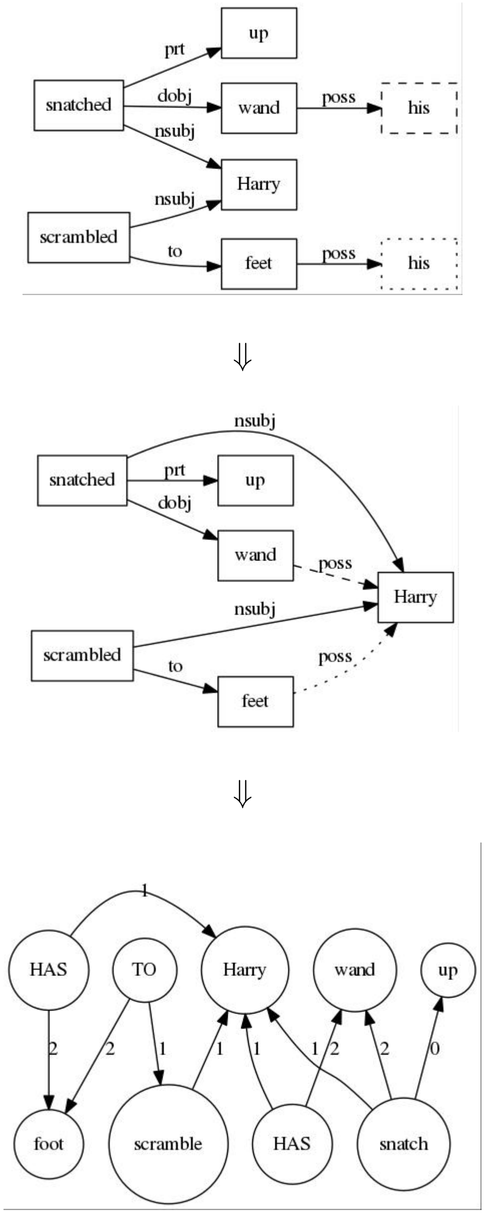


Figure 4.2: `text_to_4lang` processing of *Harry snatched up his wand and scrambled to his feet* with coreference resolution

4.4 Issues

4.4.1 Parsing errors

Using the Stanford Parser for dependency parsing yields high-quality output, it is however limited by the quality of the phrase structure grammar parser. Parsing errors constitute a major source of errors in our pipeline, occasionally resulting in dubious semantic representations that could be discarded by a system that integrates semantic analysis into the parsing process. Our long-term plans include implementing such a process within the 4lang framework using *constructions* (see Section 8.4), currently we rely on independent efforts to improve the accuracy of phrase structure grammar parsers using semantic information.

Results of a pioneering effort in this direction are already included in the latest versions of the Stanford Parser (including the one used in the 4lang system) and was introduced in Section 2.4.2: (Socher, Bauer, et al., 2013) improves the accuracy of the Stanford Parser by using *Compositional Vector Grammars*, RNN-based models that learn for each terminal rule $R^n \to R^{2n}$ linear transformations that can be applied to pairs of word vectors of length n to obtain an $n \times n$ matrix representing the nonterminal that is the result of applying the given rule. The purpose of this model is to account for the semantic relationships between words in the text that is to be parsed and words that have occurred in the training data. E.g. the sentence *He ate spaghetti with a spoon* can be structurally distinguished from *He ate spaghetti with meatballs* even if in the training phase the model has only had access to *[eat [spaghetti] [with a fork]]*, by grasping the similarity between the words *spoon* and *fork*.

This phenomenon of incorrect PP-attachment is the single most frequent source of anomalities in our output. For example, syntactic ambiguity in the Longman definition of basement: *a room or area in a building that is under the level of the ground*, which has the constituent structure in Figure 4.3 is incorrectly assigned the structure in Figure 4.4, resulting in the incorrect semantic representation in Figure 4.5 (instead of the expected graph in Figure 4.6). Most such ambiguities are easily resolved by humans based on lexical facts (in this case e.g. that buildings with some underground rooms are more common than buildings that are entirely under the ground, if the latter can be called buildings at all) but it seems that such inferencing is beyond the capabilities even for parsers using word embeddings. As already discussed in Section 3.3, such deductions can be made based on 4lang representations.

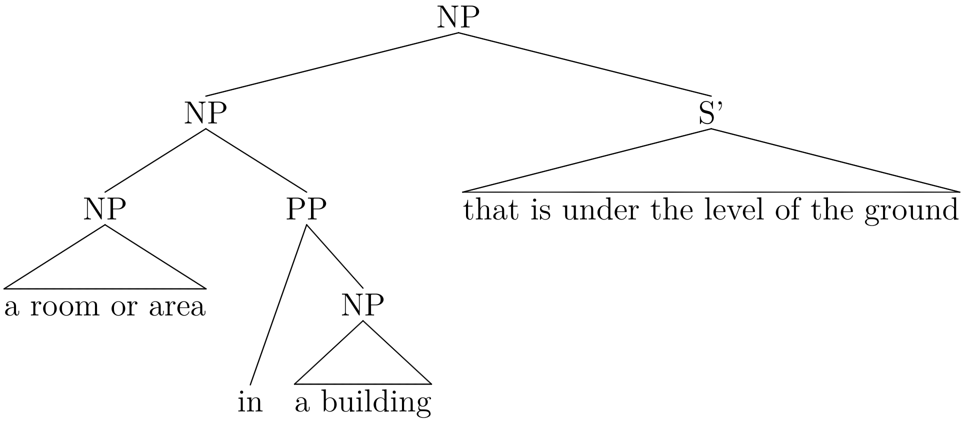


Figure 4.3: Constituent structure of *a room or area in a building that is under the level of the ground*

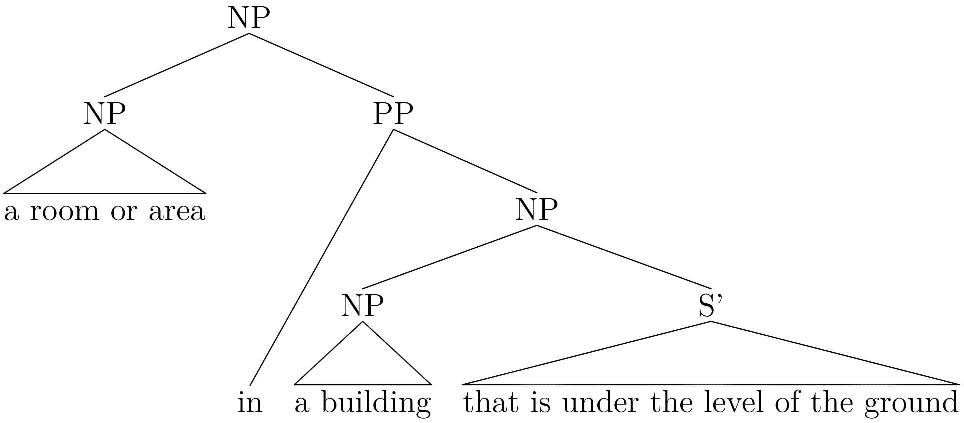


Figure 4.4: Incorrect parse tree for *a room or area in a building that is under the level of the ground*

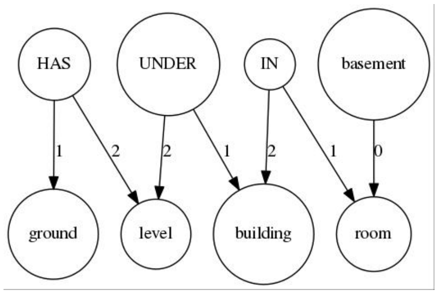


Figure 4.5: Incorrect definition graph for `basement`.

4.4.2 Relationships among clauses

The `text_to_4lang` system does not currently detect relationships between multiple clauses of a sentence expressed by conjunctions such as *because*, *unless*, *although*, etc., since they do not appear as syntactic dependency relations in the output of dependency

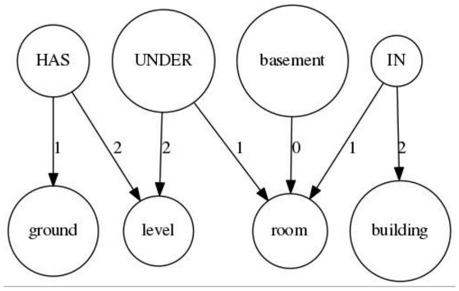


Figure 4.6: Expected definition graph for `basement`.

parsers (unlike e.g. clausal modifiers of noun phrases, which are processed by the Stanford Parser to obtain e.g. `nsubj(appears, liquid)` from the definition of `perspiration`: *liquid that appears on your skin because you are hot or nervous*). Such conjunctions should be treated on a case-by-case basis by constructions enforcing simple rules. Such a construction might state that for some sentence *X, because Y*, the `4lang` graphs corresponding to X and Y should be joined by $\xleftarrow{1}$ `CAUSE` $\xrightarrow{2}$. The definition of *perspiration* could then map to the graph in Figure 4.7.

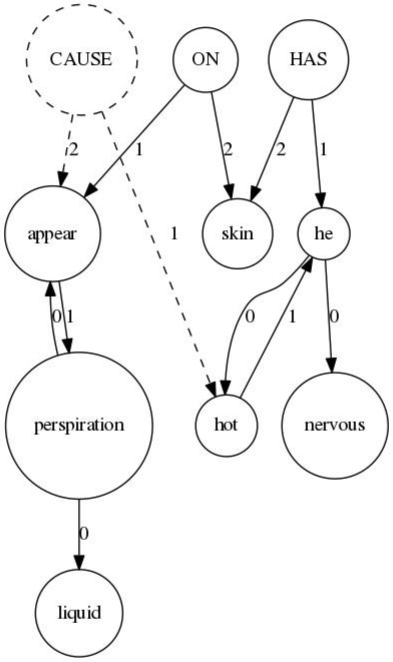


Figure 4.7: Definition graph built from **perspiration**: *liquid that appears on your skin because you are hot or nervous*

4.5 Postprocessing dependencies

Some of the typical issues of the graphs constructed by the process described in Section 4.2 can be resolved by postprocessing the dependency triplets in the parser's output before passing them to `dep_to_4lang`. Currently the `dependency_processor` module handles two configurations: coordination (Section 4.5.1) and copular sentences (Section 4.5.2)

4.5.1 Coordination

One frequent class of parser errors related to PP-attachment (cf. Section 4.4.1) involve constituents modifying a coordinated phrase which are analyzed as modifying only one of the coordinated elements. E.g. in the Longman entry **casualty** - *someone who is hurt or killed in an accident or war*, the parser fails to detect that the PP *in an accident or war* modifies the constituent *hurt or killed*, not just *killed*. Determining which of two possible parse trees is the correct one is of course difficult - once again, `casualty` may as well mean someone who is killed in an accident or war or someone who is hurt (in any way) and that such a misunderstanding is unlikely in real life is a result of inference mechanisms well beyond what we are able to model.

Our simple attempt to improve the quality of graphs built is to process all pairs of words between which a coordinating dependency holds (e.g. `conj_and, conj_or`, etc.) and copy all edges from each node to the other. This could hardly be called a solution, as it may introduce dependencies incorrectly, but in practice it has proved an improvement. In our current example this step enables us to obtain missing dependencies and thus build the correct `4lang` graph (see Figure 4.8).

4.5.2 Copulars and prepositions

Two further postprocessing steps involve copular constructions containing prepositional phrases. In simple sentences such as *The wombat is under the table*, the parser returns the pair of dependencies `nsubj(is, wombat)` and `prep_under(is, table)`, which we use to generate `prep_under(wombat, table)`. Similarly, when PPs are used to modify a noun, such as in the Longman definition of `influenza`: *an infectious disease that is like a very bad cold*, for which the dependency parser returns, among others, the triplets `rcmod(disease, is)` and `prep_like(is, cold)`, we let a simple rule add the triplet `prep_like(disease, cold)` (see Figure 4.9). In both cases we finish by removing the copular verb in order to simplify our final representation.

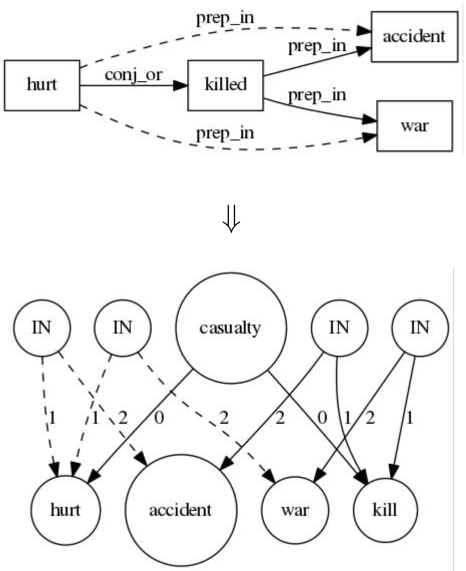


Figure 4.8: Definition graph built from: **casualty** - *someone who is hurt or killed in an accident or war*, with extra dependencies added by the postprocessor

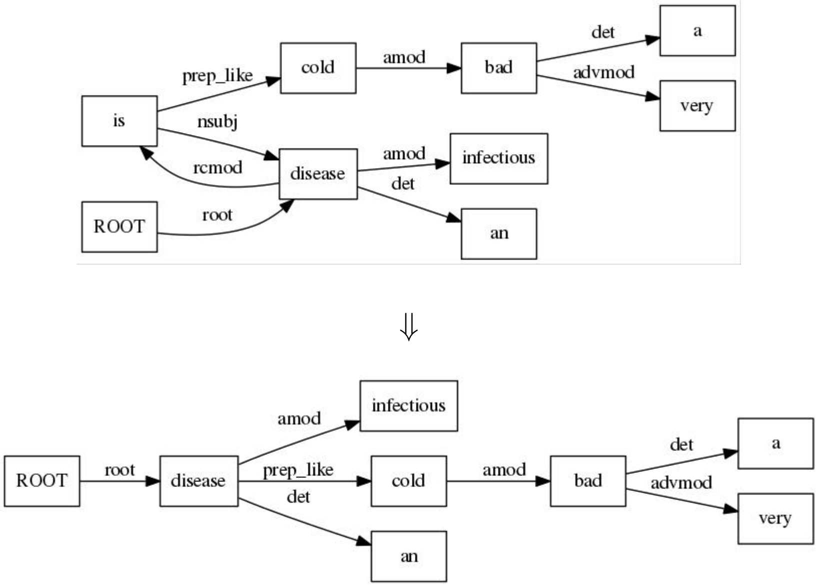


Figure 4.9: Postprocessing the definition *an infectious disease that is like a very bad cold*

4.6 Evaluation

We evaluated our pipeline on small random samples of text by inspecting both the output and the intermediate representations to understand the nature of each error. Our first round of evaluation in Section 4.6.1 involves the complete pipeline and is therefore also influenced by erroneous analyses of the Stanford Parser. To remove this factor, in Section 4.6.2 we also test `dep_to_4lang` on gold standard dependency annotations.

4.6.1 Evaluation on raw text

We performed manual evaluation of the `text_to_4lang` module on a sample from the *UMBC Webbase* corpus (Han et al., 2013), a set of 3 billion English words based on a 2007 webcrawl performed as part of the *Stanford Webbase*[3] project. We used the GNU utility `shuf` to extract a random sample of 50 sentences, which we processed with `text_to_4lang` and examined manually both the final output and the dependencies output by the Stanford Parser in order to gain a full understanding of each anomaly in the graphs created. The sentences in this corpus are quite long (22.1 words/sentence on average), therefore most graphs are affected by multiple issues; we shall now take stock of those that affected more than one sentence in our sample.

Parser errors remain the single most frequent source of error in our final `4lang` graphs: 16 sentences in our sample of 50 were assigned dependencies erroneously. 4 of these cases are related to PP-attachment (see Section 4.4.1). Parser errors are also virtually the only issue that cause incorrect edges to be added to the final graph – nearly all remaining errors will result in missing connections only. The second largest source of errors in this dataset are related to connectives between clauses that our pipeline does not currently process (see Section 4.3). Our sample contains 12 such examples, including 4 relative clauses, 4 pairs of clauses connected by *that*, and a number of other connectives such as *unless*, *regardless*, etc. The output of our pipeline for these sentences often consists of two graphs that are near-perfect representations of the two clauses, but are not connected to each other in any way – an example is shown in Figure 4.10, we shall briefly return to this issue in Section 8.1.

Three more error classes are worth mentioning based on the proportion of graphs affected by them in our sample. 5 representations suffered from recall errors made by the Stanford Coreference Resolution system: in these cases connections of a single concept in the final graph are split among two or more nodes, since our pipeline failed to identify two words as referring to the same entity (Figure 4.11 shows an example). The second type of error also affects 5 sentences, those that are assigned the `vmod` dependency. This relation holds between a noun and a *reduced non-final verbal modifier*, which "is a participial or infinitive form of a verb heading a phrase (which may have some arguments, roughly like a VP). These are used to modify the meaning of an NP or another verb."(DeMarneffe et al., 2006, p.10). This dependency is not processed by `dep_to_4lang`, since it may encode the relation between a verb and either its subject or object; e.g. the example sentences in the Stanford Dependency Manual, *Truffles picked during the spring are tasty* and *Bill tried to shoot, demonstrating his incompetence* will result in the triplets `vmod(truffles,`

[3] `http://dbpubs.stanford.edu:8091/~testbed/doc2/WebBase/`

`picked)` and `vmod(shoot, demonstrating)`, but should be represented in `4lang` by the edges `pick` $\xrightarrow{2}$ `truffle` and `shoot` $\xrightarrow{0}$ `demonstrate`, respectively. When we extend our tools to handle Hungarian input (see Section 4.7), we add to `dep_to_4lang` the capability of differentiating between words based on morphological analysis. English POS-tags are not currently processed, but this feature would make it straightforward to handle the `vmod` dependency using two rules, one for gerunds and one for participle forms.

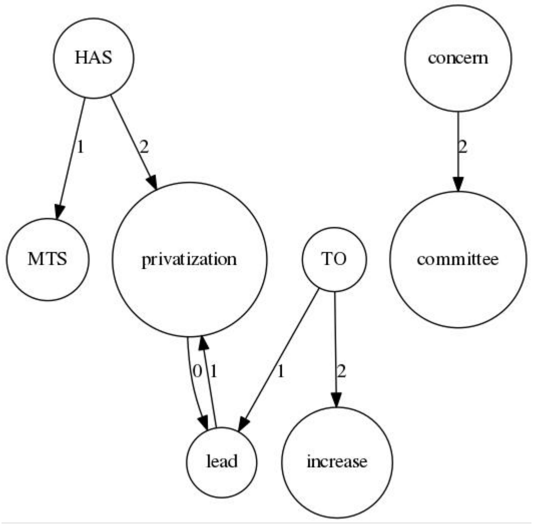


Figure 4.10: `4lang` graph built from the sentence *The Manitoba Action Committee is concerned that the privatization of MTS will lead to rate increases.*. The dependency `ccomp(concerned, lead)` was not processed.

Most representations in our sample suffer from multiple errors. While a quantitative analysis of the quality of these representations is currently not possible, our manual inspection tells us that 16 of the 50 graphs in our sample are either perfect representations of the input sentence (in 4 cases) or are affected by a single minor error only and remain high-quality representations.

4.6.2 Evaluation on gold dependencies

To test our pipeline without interference from parser errors, we also performed manual evaluation of a set of ten sentences[4] that were annotated for the 2008 COLING Workshop on Cross-Framework and Cross-Domain Parser Evaluation (De Marneffe & Manning, 2008b). Since all sentences in this sample have been taken from the Wall Street Journal (WSJ), they were expected to be more complex than the typical sentence in a webcorpus like the one used in our first evaluation. Indeed, the average sentence length in our sample

[4] http://nlp.stanford.edu/software/stanford-dependencies/required-wsj02.Stanford

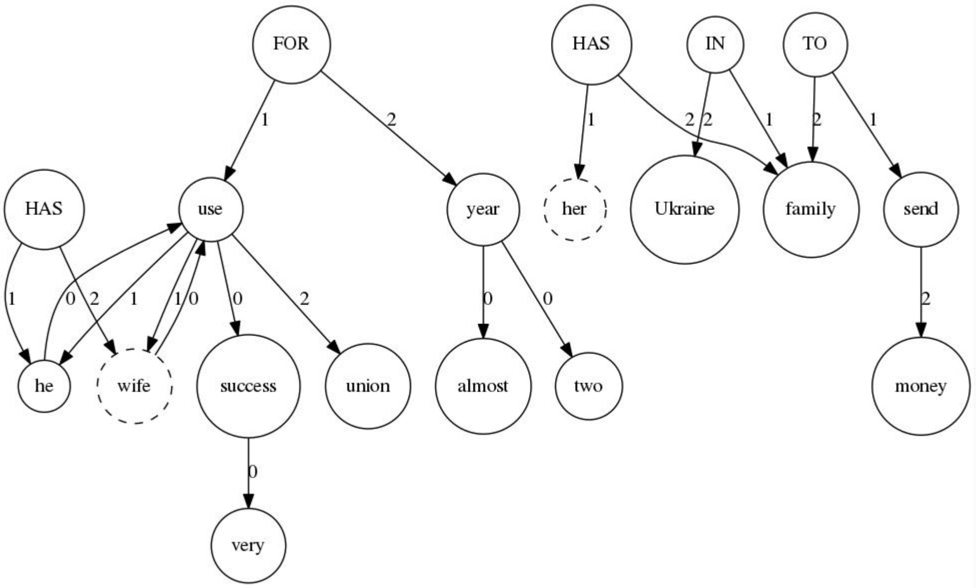


Figure 4.11: `4lang` graph built from the sentence *My wife and I have used Western Union very successfully for almost two years to send money to her family in Ukraine.* Nodes with dashed edges should have been unified based on coreference resolution.

was 27.3 words (compared to 22.1 in the Webcorpus sample). Unsurprisingly, 5 of these 10 sentences were mapped to largely erroneous representations, with 4 graphs containing large unconnected components, each representing parts of complex sentences. Nevertheless, the gold dependency analyses allowed for large good-quality representations, such as the partial representation in Figure 4.12. When errors made by the dependency parser introduce further noise into the full `text_to_4lang` pipeline, resulting `4lang` graphs suffer further in quality. The sentence whose gold parse yielded the subgraph in Figure 4.12 is mapped to a graph with large erroneous components, the largest correct subgraph is shown in Figure 4.13.

4.7 Hungarian

We have created an experimental version of our pipeline for Hungarian, using the NLP library `magyarlanc` for dependency parsing and a mapping to `4lang` graphs that is sensitive to the output of morphological analysis, to account for the rich morphology of Hungarian encoding many relations that a dependency parse cannot capture. We describe the output of `magyarlanc` and the straightforward components of our mapping in Section 4.7.1. In Section 4.7.2 we discuss the use of morphological analysis in our pipeline and in Section 4.7.3 we present some arbitrary postprocessing steps similar to those described in

Sections 4.5.1 and 4.5.2. Finally, in Section 4.7.4 we discuss the performance and main issues of the Hungarian subsystem.

4.7.1 Dependencies

The `magyarlanc` library[5] (Zsibrita et al., 2013) contains a suite of standard NLP tools for Hungarian, which allows us, just like in the case of the Stanford Parser, to process raw text without building our own tools for tokenization, POS-tagging, etc. The dependency parser component of `magyarlanc` is a modified version of the Bohnet parser (Bohnet, 2010) trained on the Szeged Dependency Treebank (Vincze et al., 2010). The output of `magyarlanc` contains a much smaller variety of dependencies than that of the Stanford Parser. Parses of the ca. 4700 entries of the NSzT dataset (to be introduced in Section 5.1) contain nearly 60,000 individual dependencies, 97% of which are covered by the 10 most frequent dependency types (cf. Table 4.2). We shall first discuss dependencies that can be handled straightforwardly in the `dep_to_4lang` framework introduced in Section 4.2.

att	26.0%
punct	16.1%
coord	15.0%
obl	9.6%
root	7.8%
conj	6.6%
mode	5.0%
det	4.7%
obj	3.7%
subj	2.6%

Table 4.2: Most common dependencies in `magyarlanc` output

The dependencies `att,` `mode,` and `pred`, all of which express some form of unary predication, can be mapped to the 0-edge. `subj` and `obj` are treated in the same fashion as the Stanford dependencies `nsubj` and `dobj`. The dependencies `from,` `tfrom,` `locy,` `tlocy,` `to,` and `tto` encode the relationship of a predicate and an adverb or postpositional phrase answering the question ‘from where?’, ‘from when?’, ’where?’, ‘when?’, ‘where to?’, and ‘until when?’, respectively, and are mapped to the binary concepts `from,` `since,` `AT,` `TO`, and `until` (see Table 4.3).

[5] http://www.inf.u-szeged.hu/rgai/magyarlanc

Dependency	Edge
att mode pred	$w_1 \xrightarrow{0} w_2$
subj	$w_1 \xrightarrow{1} w_2$
obj	$w_1 \xrightarrow{2} w_2$
from	$w_1 \xleftarrow{1}$ `FROM` $\xrightarrow{2} w_2$
tfrom	$w_1 \xleftarrow{1}$ `since` $\xrightarrow{2} w_2$
locy tlocy	$w_1 \xleftarrow{1}$ `AT` $\xrightarrow{2} w_2$
to	$w_1 \xleftarrow{1}$ `TO` $\xrightarrow{2} w_2$
tto	$w_1 \xleftarrow{1}$ `until` $\xrightarrow{2} w_2$

Table 4.3: Mapping from `magyarlanc` dependency relations to `4lang` subgraphs

4.7.2 Morphology

Hungarian is a language with rich morphology, and in particular the relationship between a verb and its NP argument is often encoded by marking the noun phrase for one of 17 distinct cases. In English, these relations would typically be expressed by prepositions, which the Stanford Parser can map to dependencies, e.g. the sentence *John climbed under the table* will yield the dependency `prep_under(table, climb)`. The Hungarian parser does not transfer the morphological information to the dependencies, all arguments other than subjects and direct objects will be in the `OBL` relation with the verb. Therefore we updated the `dep_to_4lang` architecture to allow our mappings from dependencies to `4lang` subgraphs to be sensitive to the morphological analysis of the two words between which the dependency holds. The resulting system maps the phrase *a késemért jöttem* the knife-POSS-PERS1-CAU come-PAST-PERS1 ‘I came for my knife’ to `FOR(come, knife)` based on the morphological analysis of *késemért*, performed by `magyarlanc` based on the `morphdb.hu` database (Tron et al., 2005).

This method yields many useful subgraphs, but it also often leaves uncovered the true semantic relationship between verb and argument, since nominal cases can have various interpretations that are connected to their ‘primary’ function only remotely, or not at all. The semantics of Hungarian suffixes *-nak/-nek* (dative case) or *-ban/-ben* (inessive case) exhibit great variation – not unlike that of the English prepositions *for* and *in*, and the ‘default’ semantic relations `FOR` and `IN` are merely one of several factors that must be considered when interpreting a particular phrase. Nevertheless, our mapping from nominal cases to binary relations can serve as a strong baseline, just like interpreting English *for* and *in* as `FOR` and `IN` via the Stanford dependencies `prep_for` and `prep_in`. The mapping from `magyarlanc` dependencies to `4lang` graphs is shown in Table 4.3, nominal cases of `OBL` arguments are mapped to `4lang` binaries according to Table 4.4.

Case	Suffix	Subgraph
sublative	*-ra/-re*	$w_1 \xleftarrow{1} \texttt{ON} \xrightarrow{2} w_2$
superessive	*-on/-en/-ön*	
inessive	*-ban/-ben*	$w_1 \xleftarrow{1} \texttt{IN} \xrightarrow{2} w_2$
illative	*-ba/-be*	
temporal	*-kor*	$w_1 \xleftarrow{1} \texttt{AT} \xrightarrow{2} w_2$
adessive	*-nál/nél*	
elative	*-ból/-ből*	$w_1 \xleftarrow{1} \texttt{FROM} \xrightarrow{2} w_2$
ablative	*-tól/-től*	
delative	*-ról/-ről*	
allative	*-hoz/-hez/-höz*	$w_1 \xleftarrow{1} \texttt{TO} \xrightarrow{2} w_2$
terminative	*-ig*	
causative	*-ért*	$w_1 \xleftarrow{1} \texttt{FOR} \xrightarrow{2} w_2$
instrumental	*-val/-vel*	$w_1 \xleftarrow{1} \texttt{INSTRUMENT} \xrightarrow{2} w_2$

Table 4.4: Mapping nominal cases of `OBL` dependants to `4lang` subgraphs

4.7.3 Postprocessing

Copulars

In the Szeged Dependency Treebank, and consequently, in the output of `magyarlanc`, copular sentences will contain the dependency relation `pred`. Hungarian only requires a copular verb in these constructions when a tense other than the present or a mood other than the indicative needs to be marked (cf. Table 4.5). While the sentence in (1) is analyzed as `subj(Ervin, álmos)`, all remaining sentences will be assigned the dependencies `subj(volt, Ervin)` and `pred(volt, álmos)`. The same copular structures allow the predicate to be a noun phrase (e.g. *Ervin tüzoltó* 'Ervin is a firefighter'). In each of these cases we'd like to eventually obtain the `4lang` edge `Ervin` $\xrightarrow{0}$ `sleepy` (`Ervin` $\xrightarrow{0}$ `firefighter`), which could be achieved in several ways: we might want to detect whether the nominal predicate is a noun or an adjective and add the `att` and subj dependencies accordingly. Both of these solutions would result in a considerable increase the complexity of the dep_to_4lang system and neither would simplify its input: the simplest examples (such as (1) in Table 4.5) would still undergo different treatment. With these considerations in mind we took the simpler approach of mapping all pairs of the form subj(c, x) and pred(c, y) (such that c is a copular verb) to the relation subj(y, x) (see Figure 4.14), which can then be processed by the same rule that handles the simplest copulars (as well as verbal predicates and their subjects.) The transformation must be restricted to cases where c is a copular verb: a sentence such as *Ervin álmos, ami érthető* ‘Ervin is sleepy, which is understandable‘ will be assigned the dependencies subj(álmos, Ervin) and pred(álmos, érthető), which must not be transformed but processed by dep_to_4lang to obtain Ervin $\xrightarrow{0}$ álmos $\xrightarrow{0}$ érthető.

(1)	*Ervin*	*álmos*		
	Ervin	sleepy		
	‘Ervin is sleepy’			
(2)	*Ervin*	*nem*	*álmos*	
	Ervin	not	sleepy	
	‘Ervin is not sleepy’			
(3)	*Ervin*	*álmos*	*volt*	
	Ervin	sleepy	was	
	‘Ervin was sleepy’			
(4)	*Ervin*	*nem*	*volt*	*álmos*
	Ervin	not	was	sleepy
	‘Ervin was not sleepy’			

Table 4.5: Hungarian copular sentences

Coordination

Unlike the Stanford Parser, magyarlanc does not propagate dependencies across coordinated elements. Therefore we introduced a simple postprocessing step where we find words of the sentence governing a coord dependency, then collect all words accessible from any of them via coord or conj dependencies (the latter connects coordinating conjunctions such as *és* ‘and’ to the coordinated elements). Finally, we unify the dependency relations of all coordinated elements – Figure 4.15 shows a simple example[6]

[6] This step introduces erroneous edges in a small fraction of cases: when a sentence contains two or more clauses that are not connected by any conjunction – i.e. no connection is indicated between them – a coord relation is added by magyarlanc to connect the two dependency trees at their root nodes.

4.7.4 Evaluation and issues

As in the case of the English system, we have randomly chosen 20 sentences to manually evaluate `text_to_4lang` on raw Hungarian data, and also tested `dep_to_4lang` on 10 sentences with gold dependency annotation. The source of our first sample is the Hungarian Webcorpus (Halácsy et al., 2004). As before, we shall start by providing some rough numbers regarding the average quality of the 20 `4lang` graphs, then proceed to discuss some of the most typical issues, citing examples from the used sample. 10 of the 20 graphs were correct `4lang` representations, or had only minor errors. An example of a correct transformation can be seen in Figure 4.17. Of the remaining graphs, 4 were mostly correct but had major errors, e.g. 1-2 content words in the sentence had no corresponding node, or several erroneous edges were present in the graph. The remaining 6 graphs had many major issues and can be considered mostly useless.

When investigating the processes that created the more problematic graphs, nearly all errors seem to have been caused by sentences with multiple clauses. When a clause is introduced by a conjunction such as *hogy* 'that' or *ha* 'if', the dependency trees of each graph are connected via these conjunctions only, i.e. the parser does not assign dependencies that hold between words from different clauses. We are able to build good quality subgraphs from each clause, but further steps are required to establish the semantic relationship between them based on the type of conjunction involved – a process that requires case-by-case treatment and would even then be non-trivial. An example from our sample is the sentence in Figure 4.16; here a conditional clause is introduced by a phrase that roughly translates to 'We'd be glad if...'. Even if we disregard the fact that a full analysis of how this phrase affects the semantics of the sentence would require some model of the speaker's desires – we could still interpret the sentence literally by imposing some rule for conditional sentences, e.g. that given a structure of the form A if B, the `CAUSE` relation is to hold between the root nodes of B and A. Such rules could be introduced for several types of conjunctions in the future. A further, smaller issue is caused by the general lack of personal pronouns in sentences: Hungarian is a *pro-drop* language: if a verb is inflected for person, pronouns need not be present to indicate the subject of the verb, e.g. *Eszem.* 'eat-1SG' is the standard way of saying 'I'm eating' as opposed to *?Én eszem* 'I eat-1G' which is only used in special contexts where emphasis is necessary. Currently this means that `4lang` graphs built from these sentences will have no information about who is doing the `eating`, but in the future these cases can be handled by a mechanism that adds a pronoun subject to the graph based on the morphological analysis of the verb. Finally, the lowest quality graphs are caused by very long sentences containing several clauses and causing the parser to make multiple errors.

For our second type of evaluation we used the manually annotated Szeged Dependency Treebank (Vincze et al., 2010), which allowed us to run `dep_to_4lang` on a random sample of 10 error-free dependency structures. Unlike for English, parser errors have not played a significant role in the anomalies that we encountered when testing the full pipeline, therefore our second set of results for Hungarian are very similar to the first. 3 out of 10 sentences were assigned perfect representations and another 3 only showed minor errors. The leading issue, affecting all remaining sentences, is again the relationship among multiple clauses of the same sentence, which will require case-by-case treatment in the future.

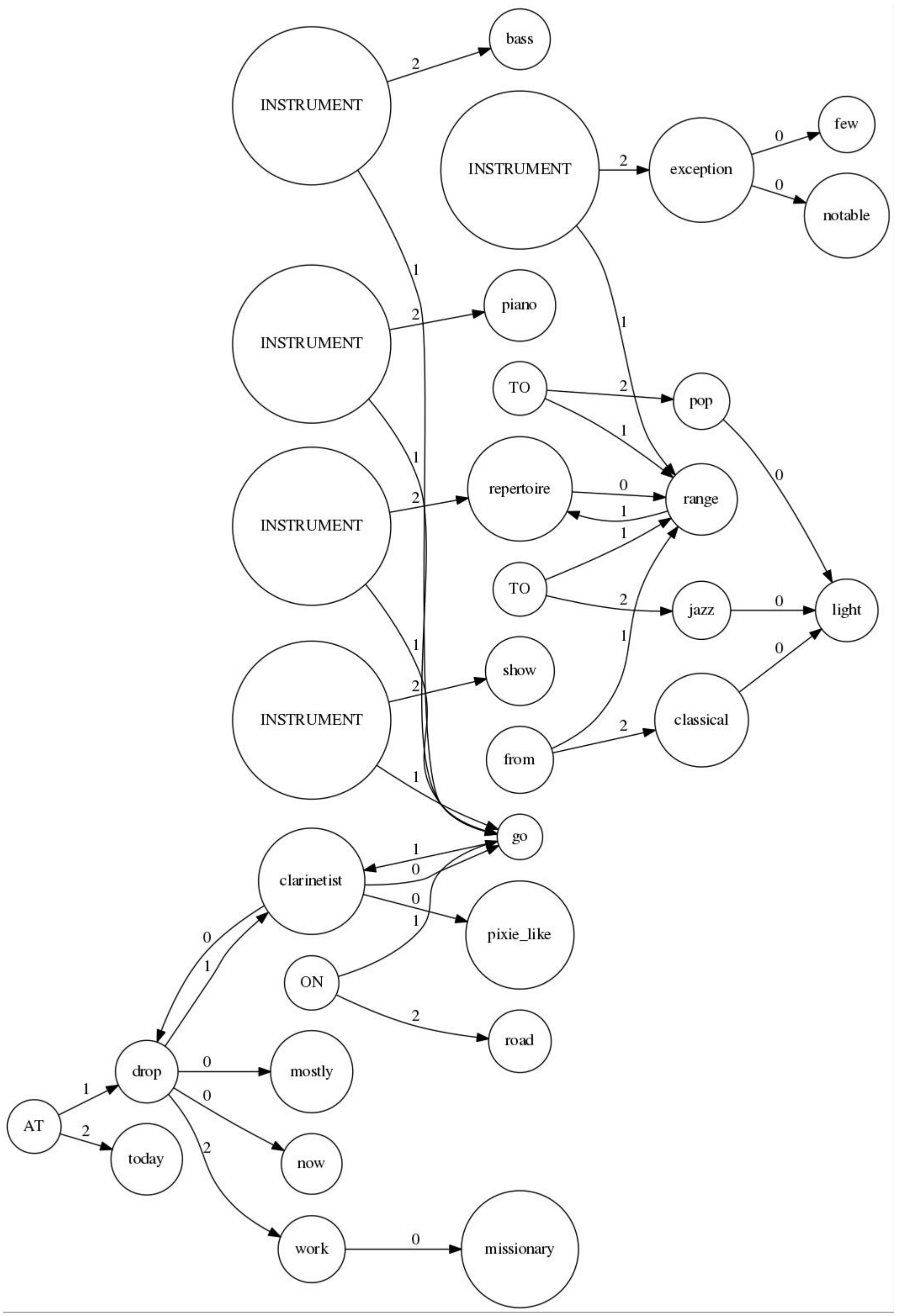


Figure 4.12: Largest correct component of the graph obtained using the perfect parse tree for the WSJ sentence *Today, the pixie-like clarinetist has mostly dropped the missionary work (though a touch of the old Tashi still survives) and now goes on the road with piano, bass, a slide show, and a repertoire that ranges from light classical to light jazz to light pop, with a few notable exceptions.*

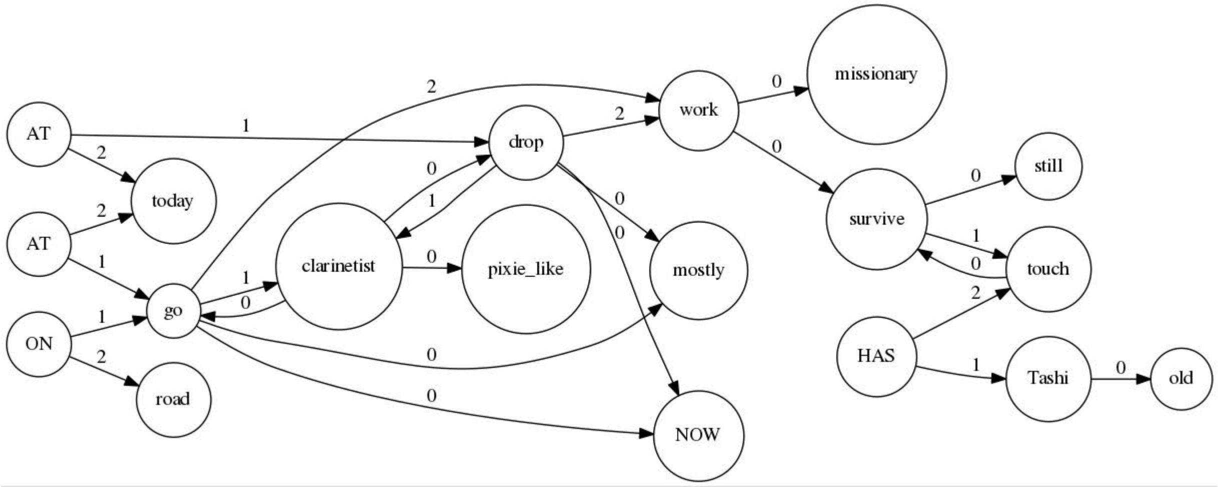


Figure 4.13: Largest correct subgraph of the `text_to_4lang` output for the WSJ sentence *Today, the pixie-like clarinetist has mostly dropped the missionary work (though a touch of the old Tashi still survives) and now goes on the road with piano, bass, a slide show, and a repertoire that ranges from light classical to light jazz to light pop, with a few notable exceptions.*

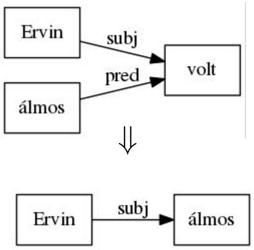


Figure 4.14: Postprocessing dependencies of a copular sentence

Csengő,	*vidám,*	*kellemes*	*kacagás*	*hangzott*	*a*	*magasból*
Ringing	joyful	pleasant	giggle	sound-PST	the	height-ELA

‘Ringing, merry, pleasant laughter sounded from above’

⇓

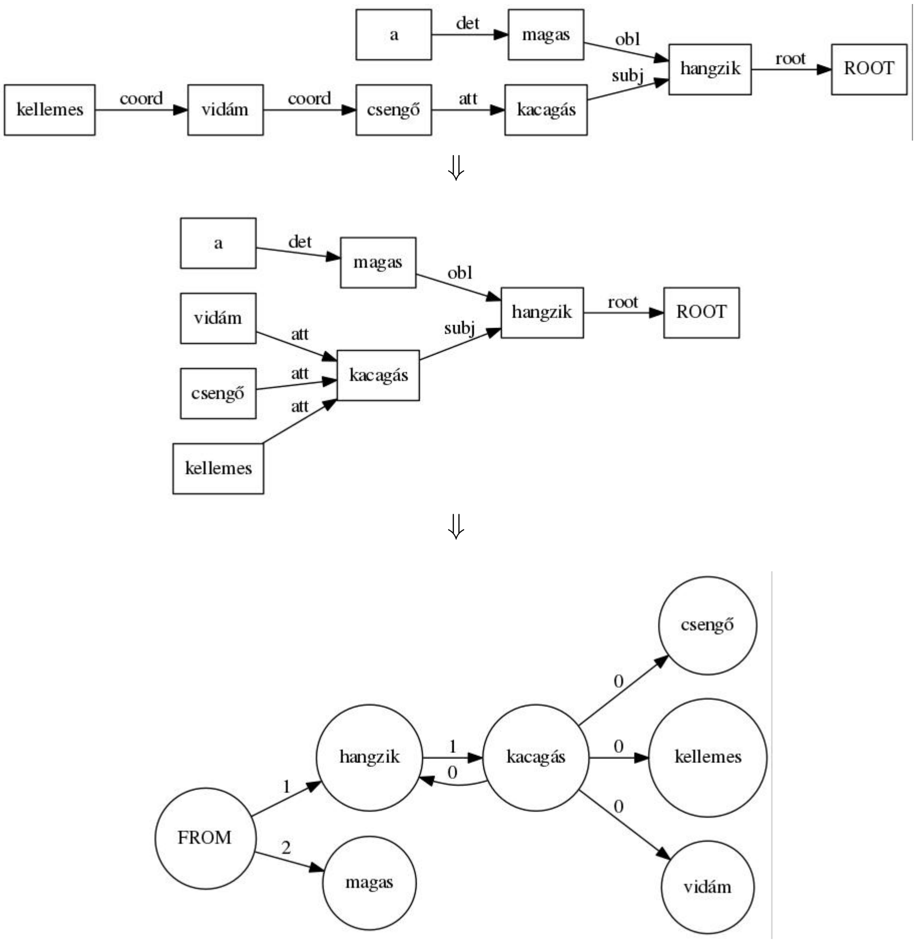


Figure 4.15: Processing a coordinated sentence

Örülnénk,	*ha*	*a*	*konzultációs*	*központok*
rejoice-COND-1PL	if	the	consultation-ATT	center-PL
közötti	*kilométerek*	*nem*	*jelentenének*	
between-ATT	kilometer-PL	not	mean-COND-3PL	
az	*emberek*	*közötti*	*távolságot.*	
the	person-PL	between-ATT	distance-ACC	

‘We’d be glad if the kilometers between consultation centers did not mean distance between people’

Figure 4.16: Subordinating conjunction

1995	*telén*	*vidrafelmérést*	*végeztünk*
1995	winter-POSS-SUP	otter-survey-ACC	conduct-PST-1PL
az	*országos*	*akció*	*keretében.*
the	country-ATT	action	frame-POSS-INE
‘In the winter of 1995 we conducted an otter-survey as part of our national campaign’			

⇓

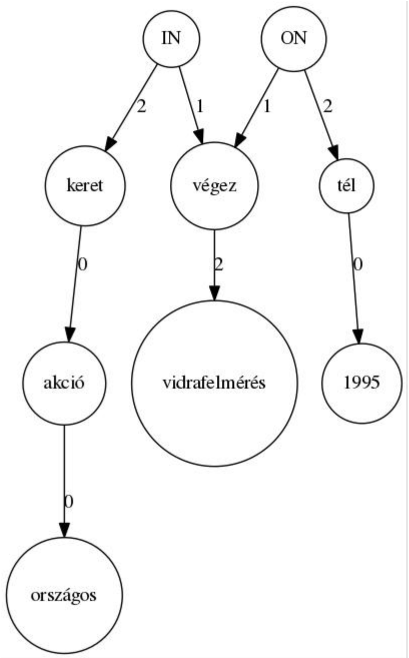


Figure 4.17: Example of perfect `dep_to_4lang` transformation

Chapter 5

Building definition graphs

One application of the text_to_lang module is of particular importance to us. By processing entries in monolingual dictionaries written for humans we can attempt to build definition graphs like those in 4lang for practically any word. This section presents the dict_to_4lang module, which extends the text_to_4lang pipeline with parsers for several major dictionaries (an overview of these is given in Section 5.1) as well as some preprocessing steps specific to the genre of dictionary definitions – these are presented in Section 5.2. Section 5.3 discusses *expansion* of 4lang representations, the process of copying links in definition graphs (both hand-written and built by dict_to_4lang) to 4lang representations created by text_to_4lang. Finally, Section 5.4 points out several remaining issues with definition graphs produced by the dict_to_4lang pipeline. Applications of dict_to_4lang, shall be described in Chapter 6. The entire pipeline is available as part of the 4lang library, implemented by the dict_to_4lang module[1].

5.1 Data sources

We've built parsers for three large dictionaries of English and two of Hungarian. Custom parsers have been built for all five sources and are distributed as part of the 4lang module.

5.1.1 Longman Dictionary of Contemporary English

The Longman Dictionary of Contemporary English (Bullon, 2003) contains ca. 42,000 English headwords. Its definitions are constrained to a small vocabulary, the Longman Defining Vocabulary (LDV, (Boguraev & Briscoe, 1989)). The longman_parser tool processes the xml-formatted data and extracts for each headword a list of its senses, including

[1] https://github.com/kornai/4lang/blob/master/src/dict_to_4lang.py

for each the plain-text definition, the part-of-speech tag, and the full form of the word being defined, if present: e.g. definitions of acronyms will contain the phrase that is abbreviated by the headword. No component of `4lang` currently makes use of this last field, `AAA` will not be replaced by `American Automobile Association`, but this may change in the future.

5.1.2 Collins Cobuild Dictionary

The Collins-COBUILD dictionary (Sinclair, 1987) contains over 84,500 headwords. Its definitions use a vocabulary that is considerably larger than LDOCE, including a large technical vocabulary (e.g. **adularia:** *a white or colourless glassy variety of orthoclase in the form of prismatic crystals*), rare words (**affricare:** *to rub against*), and multiple orthographic forms (**adsuki bean:** *variant spelling of adzuki bean*). Since many definitions are simply pointers to other headwords, the average entry in Collins is much shorter than in LDOCE. Given the technical nature of many entries, the vocabulary used by definitions exhibits a much larger variety: Longman definitions, for the greatest part limited to the LDV, contain less than 9000 English lemmas, not including named entities, numbers, etc., Collins definitions use over 38,000 (these and subsequent figures on vocabulary size are approximated using the `hunmorph` analyzer and the morphological databases `morphdb.en` and `morphdb.hu`).

5.1.3 English Wiktionary

Our third source of English definitions, the English Wiktionary at `http://en.wiktionary.org` is the most comprehensive database, containing over 128,000 headwords and available via public data dumps that are updated weekly. Since wiktionaries are available for many languages using similar – although not standardized – data formats, it has long been a resource for various NLP tasks, among them an effort to extend the `4lang` dictionary to 40 languages (Ács et al., 2013). While for most languages datasets such as Longman and Collins may not be publicly available (e.g. at the time of writing this thesis, both Hungarian dictionaries were only available to the author based on personal requests), wiktionaries currently contain over 100,000 entries for nearly 40 languages, and over 10,000 for a total of 76.

5.1.4 Dictionaries of Hungarian

We've also run the `dict_to_4lang` pipeline on two explanatory dictionaries of Hungarian: volumes 3 and 4 of the *Magyar Nyelv Nagyszótára* (NSzt), containing nearly 5000 headwords starting with the letter *b* (Ittzés, 2011)[2], and over 120,000 entries of the complete *Magyar Értelmező Kéziszótár* (EKsz) (Pusztai, 2003), which has previously been used for NLP research (Miháltz, 2010). Basic figures for all five datasets are presented in Table 5.1.

Dict	headwords	av. def. length	approx. vocab. size
LDOCE	30,126	11.6	9,000
Collins	82,026	13.9	31,000
en.wikt	128,003	8.4	38,000
EKsz	67,515	5.0	33,700
NSzt (b)	4 683	10.7	9 900

Table 5.1: Basic figures for each dataset

5.2 Parsing definitions

5.2.1 Preprocessing

Before passing dictionary entries to the parser, we match them against some simple patterns that are then deleted or changed to simplify the phrase or sentence without loss of information. A structure typical of dictionary definitions are noun phrases with very generic meanings, e.g. *something*, *one*, *a person*, etc. For example, LDOCE defines `buffer` as *someone or something that protects one thing or person from being harmed by another*. The frequency of such structures makes it worthwhile to perform a simple preprocessing step: phrases such as *someone*, *someone who*, *someone*, etc. are removed from definitions in order to simplify them, thus reducing the chance of error in later steps. The above definition of `buffer`, for example, can be reduced to *protects from being harmed*, which can then be parsed to construct the definition graph `protect` $\xleftarrow{1}$ `FROM` $\xrightarrow{2}$ `harm`. A similar step replaced all occurences of the strings *a type of* and *a kind of* with *a*, once again simplifying both the input of the syntactic parser and the final representation without loss of information in definitions such as **lizard**: *a type of reptile that has four legs and a long tail*.

[2] The author gratefully acknowledges editor-in-chief Nóra Ittzés for making an electronic copy available.

5.2.2 Constraining the parser

Since virtually all dictionary definitions of nouns are single noun phrases, we constrain the parser to only allow such analyses for the definitions of all noun headwords. The command-line interface of the Stanford Parser does not support adding constraints on parse trees, but the Java API does; we implemented a small wrapper in `jython` that allowed us to access the classes and functions necessary to enforce this constraint (see Section 7.4.3 for more details). This fixes many incorrect parses, e.g. when a defining noun phrase with the structure in Figure 5.1 could also be parsed as a complete sentence, as in Figure 5.2.

```
(S
  (NP
    (NP (DT the) (NN size))
    (PP (IN of)
      (NP (DT a) (NN radio) (NN wave)))
    (VP (VBN used)
      (S
        (VP (TO to)
          (VP (VB broadcast)
            ( ... )))))))
```

Figure 5.1: Expected parse tree for the definition of **wavelength**: *the size of a radio wave used to broadcast a radio signal*

```
(S
  (NP
    (NP (DT the) (NN size))
    (PP (IN of)
      (NP (DT a) (NN radio) (NN wave))))
  (VP (VBD used)
    (S
      (VP (TO to)
        (VP (VB broadcast)
          ( ... ))))))
```

Figure 5.2: Incorrect parse tree from the Stanford Parser for the definition of **wavelength**: *the size of a radio wave used to broadcast a radio signal*

5.2.3 Building definition graphs

The output of the – possibly constrained – parsing process is passed to the `dep_to_4lang` module introduced in Chapter 4. The `ROOT` dependency in each parse, which was ignored in the general case, is now used to identify the head of the definition, which is typically a hypernym of the word being defined (but see Section 5.4.2 for exceptions). This allows us to connect, via a 0-edge, the node of the concept being defined to the graph built form its definition.

Dict	# graphs	av. nodes
LDOCE	24,799	6.1
Collins	45,311	4.9
en.wikt	120,670	5.4
EKsz	67,397	3.5
NSzt	4,676	6.4

Table 5.2: Graphs built from each dataset

Detecting the hypernym of a headword in its dictionary definition is a simple task that would not in itself require syntactic parsing. A simple algorithm for detecting hypernyms of Hungarian nouns is presented by (Miháltz, 2010), and was used on definitions of EKsz (see Section 5.1) when constructing the Hungarian WordNet. The author has proposed a more generic and somewhat less accurate algorithm. On a small sample of `NSzt` entries, the two algorithms achieved an accuracy of 91 and 98 percent, respectively. Details of this work are presented in a 2013 manuscript which is still under review by the journal *Magyar Nyelv* at the time of submitting this thesis.

5.3 Expanding definition graphs

The `4lang` dictionary contains by design all words of the Longman Defining Vocabulary (LDV, (Boguraev & Briscoe, 1989)). This way, if we use `dict_to_4lang` to define each headword in LDOCE as a graph over nodes corresponding to words in its dictionary definition, these graphs will only contain concepts that are defined in the hand-written `4lang` dictionary. To take advantage of this, we implement an *expansion* step in `4lang`, which adds the definition of each concept to a `4lang` graph by simply adjoining each definition graph to G at the node corresponding to the concept being defined. This can be stated formally as follows:

Definition 1. *Given the set of all concepts C, a* `4lang` *graph G with concept nodes $V(G) = c_1, c_2, \ldots, c_i \in C$, a set of definition graphs D, and a lexicon function $L : C \to D$ such that $\forall c \in C : c \in V(L(c))$, we define the expansion of G as*

$$G^* = G \cup \bigcup_{c_i \in L} L(G)$$

Hand-written definitions in the `4lang` dictionary may also contain pointers to arguments of the definiendum, e.g. `stand` is defined as `upright` $\xleftarrow{0}$ `=AGT` $\xleftarrow{1}$ `ON` $\xrightarrow{2}$ `feet`, indicating that it is the agent of `stand` that is $\xrightarrow{0}$ `upright`, etc. Detecting the thematic role of a verb's arguments can be difficult, yet we handle the majority of cases correctly using a simple step after expansion: all edges containing `=AGT` (`=PAT`) nodes are moved to the machine(s) with a 1-edge (2-edge) pointing to it from the concept being defined. This allows us to create the graph in Figure 5.3 based on the above definition of `stand`. Expansion will affect all nodes of graphs built from LDOCE; when processing generic English text using `text_to_4lang` we may choose to limit expansion to manually built `4lang` definitions, or we can turn to dictionaries built using `dict_to_4lang`, allowing ourselves to add definitions to nearly all nodes. `4lang` modules can be configured to select the approach most suitable for any given application.

5.4 Issues and evaluation

In this section we will describe sources of errors in our pipeline besides those caused by incorrect parser output (see Section 4.4.1). We shall also present the results of manual error analysis conducted on a small sample of graphs in an effort to determine both the average accuracy of our output graphs as well as to identify the key error sources.

5.4.1 Error analysis

To perform manual evaluation of the `dict_to_4lang` pipeline we randomly selected 50 headwords from the Longman Dictionary[3]. In one round of evaluation we grouped the 50 definition graphs by quality, disregarding the process that created them. We found that 31 graphs were high-quality representations: 19 perfectly represented all facts present

[3] The 50 words in our sample, selected randomly using GNU `shuf` were the following: *aircraft, arbour, armful, characteristic, clothesline, contact, contrived, costermonger, cycling, cypress, dandy, efface, excited, fedora, forester, frustrate, gazette, grenade, houseboy, incandescent, invalid, khaki, kohl, lecture, lizard, might, multiplication, nightie, okey-doke, outdid, overwork, popularity, preceding, Presbyterian, punch-drunk, reputed, residency, retaliation, rock-solid, sandpaper, scant, sewing, slurp, transference, T-shirt, underwrite, vivace, well-fed, whatsit, Zen*

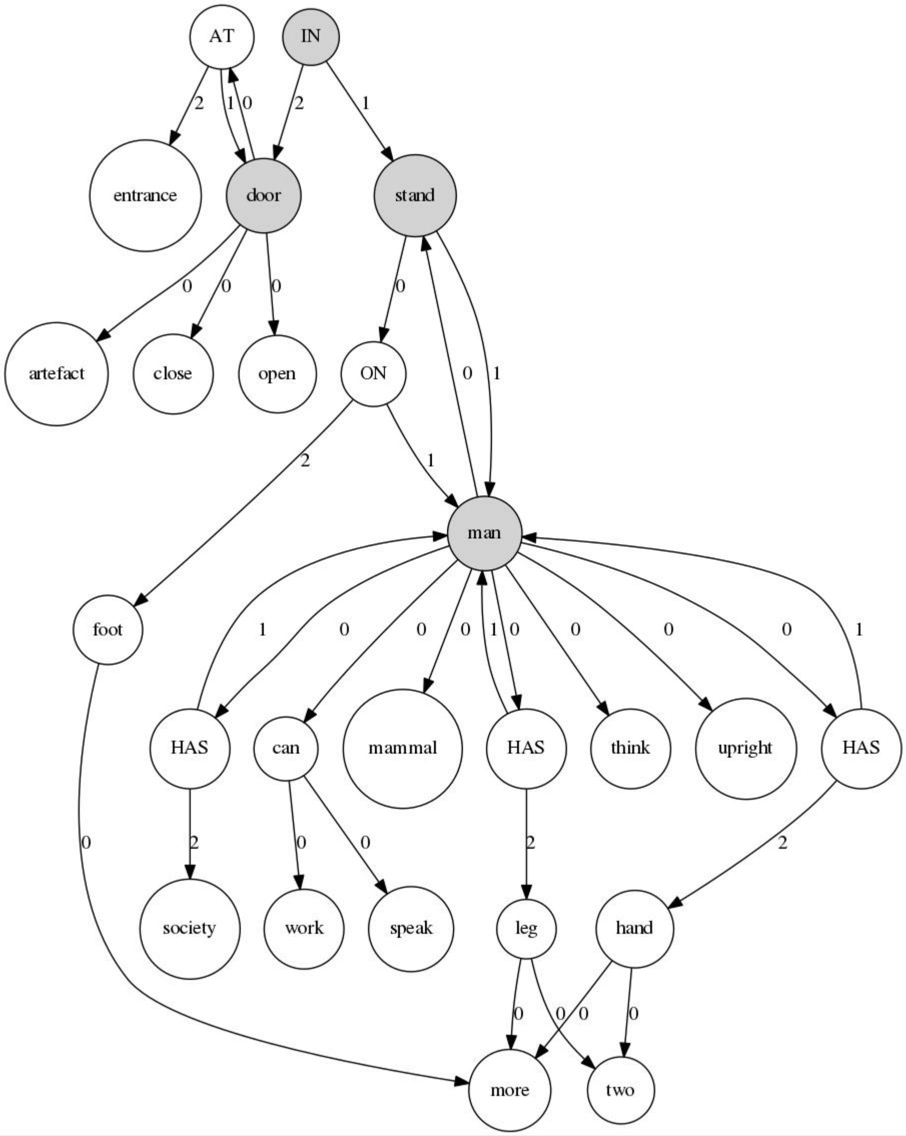


Figure 5.3: Expanded graph for *A man stands in the door*. Nodes of the unexpanded graph are shown in gray

in the dictionary entry (see e.g. Figure 5.4) and another 15 were mostly accurate, with only minor details missing or an incorrect relation present in addition to the correct ones. Of the remaining 19 graphs, 9 still encoded several true relationships, the last 10 were essentially useless. Our sample is too small to conclude that 62% of the graphs we build are of acceptable quality, but these results are nevertheless promising. Our second round of manual inspection was directed at the entire process of building the 50 graphs and aimed to identify the source of errors. Out of the 34 graphs that had errors at all, 8 were clearly a result parser errors (discussed in Section 4.4.1), another 8 contained non-compositional structures that in the future may be handled by *constructions* (see

Section 8.6.2), and 3 were connected to non-standard definitions (see Section 5.4.2). All remaining errors were caused by one-of-a-kind bugs in the pipeline, e.g. preprocessing issues, the occasional overgeneration of relations by the postprocessing of coordinated structures (see Section 4.5.1), etc.

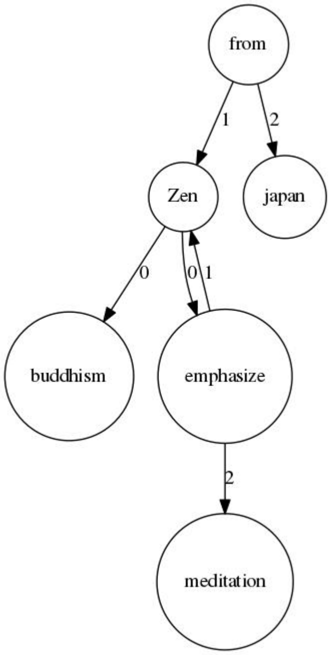


Figure 5.4: Graph constructed from the definition of **Zen**: *a kind of Buddhism from Japan that emphasizes meditation*

5.4.2 Non-standard definitions

Our method for building `4lang` definitions can be successful in the great majority of cases because most dictionary definitions – or at least their first sentences, which is all we make use of – are rarely complex sentences; in most cases they are single phrases describing the concept denoted by the headword – a typical example would be the definition of `koala`: *an Australian animal like a small grey bear with no tail that climbs trees and eats leaves*. It is these kinds of simple definitions that are prevalent in the dictionaries we process and that are handled quite accurately by both the Stanford Parser and our mapping from dependencies to `4lang` relations.

In some cases, definitions use full sentences to explain the meaning of a word in a more straightforward and comprehensible way, e.g.:

- **playback** - *the playback of a tape that you have recorded is when you play it on a machine in order to watch or listen to it*

- **indigenous** - *indigenous people or things have always been in the place where they are, rather than being brought there from somewhere else*
- **ramshackle** - *a ramshackle building or vehicle is in bad condition and in need of repair*

These sentences will result in a higher number of dependency relations, and consequently a denser definition graph; often with erroneous edges. In the special case when the Stanford Parser's output does not contain the `ROOT` relation, i.e. the parser failed to identify any of the words as the root of the sentence, we skip the entry entirely – this affects 0.76% of LDOCE entries, 0.90% of entries in `en.wiktionary`. That such definitions are problematic is also reflected in the fact that earlier editions of the Longman dictionary did not allow them, using the headword in the definition text was forbidden.

5.4.3 Word senses

As discussed in Section 3.2, the `4lang` theory assigns only one definition to each word form, i.e. it does not permit multiple word senses. All usage of a word must be derived from a single concept graph. Explanatory dictionaries like the ones listed in Section 5.1 provide several definitions for each word, of which we always process the first one. This decision is somewhat arbitrary, but produces good results in practice; the first definition typically describes the most common sense of the word, as in the case of `tooth`:

1. one of the hard white objects in your mouth that you use to bite and eat food
2. one of the sharp or pointed parts that sticks out from the edge of a comb or saw

We cannot expect to construct from this entry a generic definition such as `sharp, one_of_many`. Instead, to capture at a later stage that objects other than those in your mouth could be instances of `tooth`, we must turn to the principle that any link in a `4lang` definition can be overridden (see Section 3.2). Not only are we unable to predict the particular subset of links in the definition of `tooth` that will be shared across various uses of the word *tooth*, we *shouldn't* make any such predictions: it is no more than an accident that teeth turned out to be metaphors for small, sharp objects lined up next to one another and not for e.g. small, white, cube-shaped objects.

While in most cases the various senses defined for a word are metaphoric uses of the first, there remain words whose first definition is not generic enough to accommodate all others even if we assume powerful inferencing capabilities. Consider e.g. the definitions of **shower** from LDOCE below:

1. a piece of equipment that you stand under to wash your whole body
2. an act of washing your body while standing under a shower
3. a short period of rain or snow
4. a lot of small, light things falling or going through the air together
5. a party at which presents are given to a woman who is going to get married or have a baby
6. a group of stupid or lazy people
7. to wash your whole body while standing under a shower
8. to give someone a lot of things
9. to scatter a lot of things onto a person or place, or to be scattered in this way

A `4lang` definition generic enough so that one could derive at least the majority of these cases would be most similar to definition #4: showers are occurrences of many things falling, typically through the air. Understanding the word *shower* in the context of e.g. baby showers (#5) would remain a difficult task, including among others that of understanding that `fall` may refer to an object changing place not only physically but also in terms of ownership. In the above LDOCE entry, since we use the first definition to build the `4lang` graph, we lose any chance of recovering any of the meanings #3-6 and #8-9. The lexicographic principle that keeps sense #2 and sense #7 separate simply does not apply in 4lang, which does not distinguish meanings that differ in part of speech alone: the verb and the *nomen actionis* are simply one and the same. We further note that many of the distinctions made here would be made by overt suffixes in other languages, e.g. the Hungarian equivalents of #1 and #2 are *zuhany* and *zuhanyozik*, respectively.

5.4.4 Hungarian

We also conducted manual error analysis on our Hungarian output, in this case choosing 50 random words from the EKsz dictionary[4]. The graphs built by `dict_to_4lang` were of

[4] The 50 words, selected once again using `shuf`, are the following: *állomásparancsnok, állványoz, áttölt, apoteózis, bányatelep, beköt, berukkol, bibliapapír, biplán, bugás, dús, egyidejűleg, emu, exkuzál, font, főmufti, gimnasztika, groteszk, gumósodik, hajkötő, héja, hiánycikk, indikál, írdogál, jobbágyság, kicifráz, közjáték, kukoricamorzsoló, lejön, leszállít, megnyilvánulás, megsző, munkásőr, nagyanyó, nemtelen, összehajtogat, pántlikás, piff-puff, sietség, szemelvény, szét, tetemrehívás, tipográfus, túlkiabálás, vakolat, vízzáró, vöröspecsenye, zajszint, zihál, zsongít.*

very good quality (see Figure 5.5 for an example), with only 10 out of 50 containing major errors. This is partly due to the fact that `NSzt` contains many very simple definitions, e.g. 6 of the 50 headwords in our random sample contained only a list of synonyms as its definition.

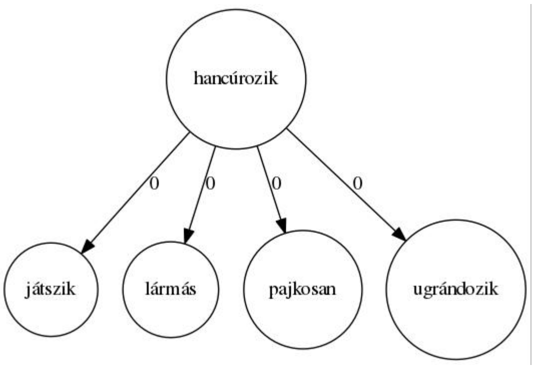


Figure 5.5: `4lang` graph built from the definition of **hancúrozik**: *Pajkosan, lármásan játszik, ugrándozik* 'shrewdly noisily play-PERS3 skip-PERS3'

4 of the 10 significant errors are caused by the same pattern: the analysis of possessive constructions by `magyarlanc` involve assigning the `att` dependency to hold between the possessor and the possessed, e.g. the definition of `piff-puff` (see Figure 5.6) will receive the dependencies `att(hang, kifejezés)` and `att(lövöldözés, hang)`, resulting in the incorrect `4lang` graph in Figure 5.7 instead of the expected one in Figure 5.8. $\texttt{kifejezés} \xrightarrow{0} \texttt{hang} \xrightarrow{0} \texttt{lövöldözés}$ instead of $\texttt{kifejezés} \xleftarrow{2} \texttt{HAS} \xrightarrow{1} \texttt{hang} \xleftarrow{2} \texttt{HAS} \xrightarrow{1} \texttt{lövöldözés}$. These constructions cannot be handled even by taking morphological analysis into account, since possessors are not usually marked (although in some structures they receive the dative suffix *-nak/-nek*, e.g. in embedded possessives like our current example (*hangjának* 'sound-POSS-DAT' is marked by the dative suffix as the possessor of *kifejezésére*). Unless possessive constructions can be identified by `magyarlanc`, we shall require an independent parsing mechanism in the future. The structure of Hungarian noun phrases can be efficiently parsed using the system described in (Recski, 2014), the grammar used there may in the future be incorporated into a `4lang`-internal parser (see Section 8.4).

5.4.5 Comparing dictionaries

Since `dict_to_4lang` currently processes three monolingual dictionaries of English, we obtain three independent definition graphs for most English words. In all applications presented in following chapters we rely on definitions from the Longman dictionary, since its definitions are limited to a vocabulary that is a subset of the `4lang` dictionary, therefore

Lövöldözés	*vagy*	*ütlegelés*	*hangjának*	*kifejezésére*
Shooting	or	thrashing	sound-POSS-DAT	expression-POSS-SUB

‘Used to express the sound of shooting or thrashing’

⇓

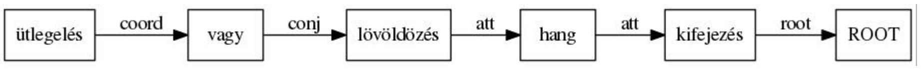


Figure 5.6: Dependency parse of the `EKsz` definition of the (onomatopoeic) term `piff-puff`

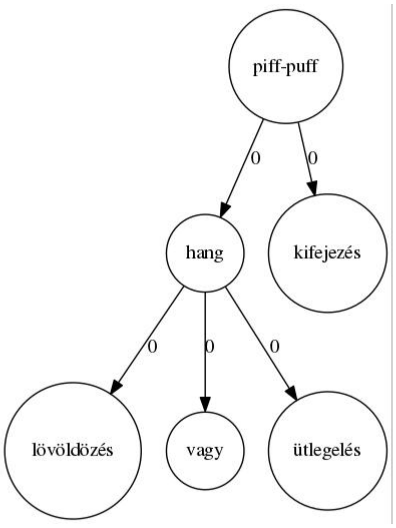


Figure 5.7: Incorrect graph for `piff-puff`

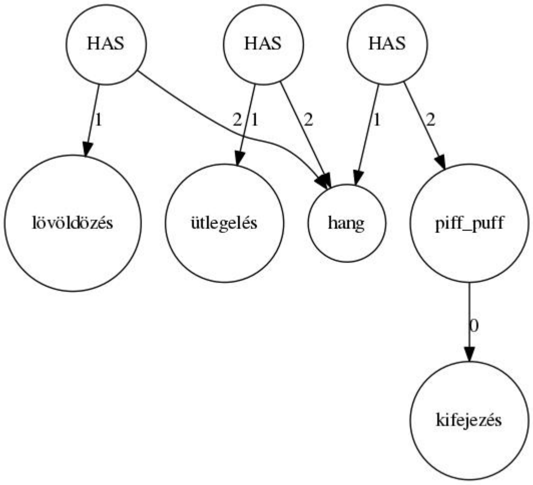


Figure 5.8: Expected graph for `piff-puff`

the expansion step described in Section 5.3 will not introduce additional errors caused by the `dict_to_4lang` pipeline. We also expect Longman definitions to be of a more stable quality than user-generated definitions of `en.wiktionary`. The figures presented in Table 5.1 also suggest that definitions in the Collins dictionary are on average slightly more complex than those in Longman. These tendencies are illustrated by the three graphs built from three definitions of `oak` in Figures 5.9, 5.10, and 5.11. While all current applications use `4lang` graphs built from Longman, in some cases it may be useful to unify multiple graphs to obtain a definition which covers a larger number of facts about the concept at the cost of a potentially larger number of errors. Such a unification of the three graphs for `oak` would yield the graph in Figure 5.12.

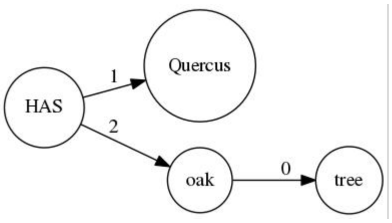


Figure 5.9: `4lang` graph built from the `en.wiktionary` definition of **oak**: *A tree of the genus Quercus.*

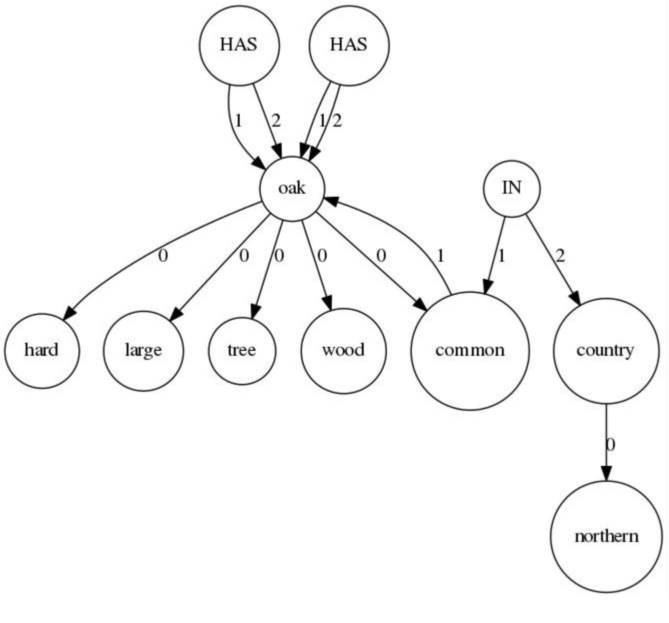


Figure 5.10: `4lang` graph built from the Longman definition of **oak**: *a large tree that is common in northern countries, or the hard wood of this tree*

Although `4lang` concepts are language-independent, `text_to_4lang` and `dict_to_4lang` cannot currently map non-English words to the concepts designated by their English names. The experimental versions of each pipeline for processing Hungarian data create `4lang` graphs whose nodes are associated with Hungarian lemmas. It is therefore

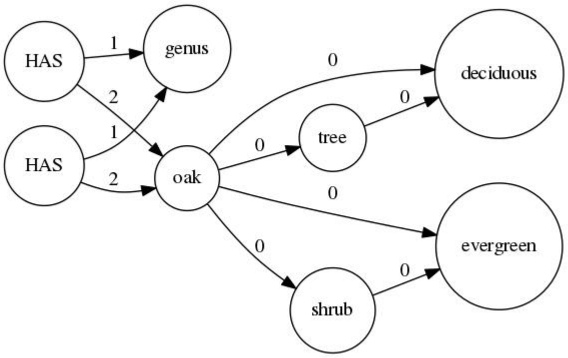


Figure 5.11: `4lang` graph built from the Collins definition of **oak**: *any deciduous or evergreen tree or shrub of the fagaceous genus*

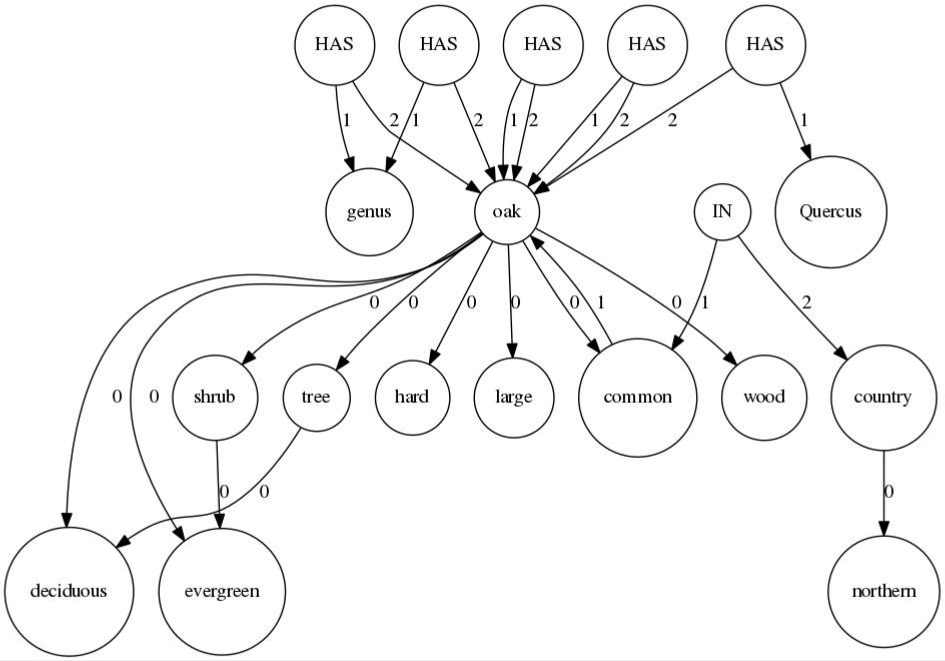


Figure 5.12: Unified `4lang` graph built using three definitions of `oak`

premature to compare graphs built for the same concept using definitions from multiple languages, but comparing Hungarian definition graphs with their English counterparts (cf. Figure 5.13[5]) suggests that unifying nodes across languages may supply additional evidence for facts about a concept.

[5] parts of the definition undetected by `dict_to_4lang` are omitted for clarity

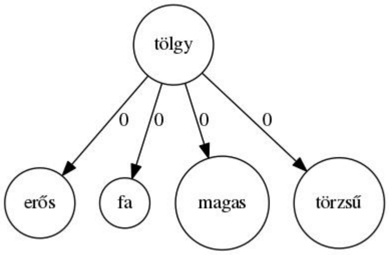


Figure 5.13: `4lang` graph built from the EKSz definition of **tölgy** ‘oak’: *Erős, magas törzsű (...) fa* ’Strong, tall trunk-INAL (...) tree’

Chapter 6

Applications

This chapter presents applications of the `4lang` system. Section 6.1 presents our earliest experiments on measuring semantic similarity between words and sentences using `4lang` graphs, resulting in a system submitted to the *Semantic Textual Similarity* task of the 2015 SemEval conference[1]. Section 6.2 documents the more recent `wordsim` system for measuring similarity of word pairs, which we evaluate on the popular benchmark SimLex-999, achieving significant improvement over the current state of the art. Finally, Section 6.3 presents two early attempts at natural language understanding systems that use *spreading activation* over `4lang` graphs.

The 2015 SemEval system described in Sections 6.1 is a result of joint work with Judit Ács. The open-source system, documented in more detail by (Recski & Ács, 2015), is available at `https://github.com/juditacs/semeval`. The `wordsim` system was created in collaboration with Eszter Iklódi, key components of the ML system were contributed by Katalin Pajkossy. The most detailed description is (Recski, Iklódi, et al., 2016), the code is available under an MIT license at `https://github.com/recski/wordsim`. The systems presented in Section 6.3 were built in cooperation with Dávid Nemeskey and Attila Zséder (2012). This code is no longer functional, although several of its components are still maintained as part of the `pymachine` module used by the `4lang` system (see Section 7.7 for details).

6.1 Semantic similarity of sentences

To demonstrate the use for concept graphs built using `dict_to_4lang`, we participated in two tasks of the 2015 `Semeval` conference: Task 1 - *Paraphrase and Semantic Similarity in Twitter* (Xu et al., 2015) involved detecting paraphrases among tweets (Task 1a) and

[1] `http://alt.qcri.org/semeval2015/`

measuring the semantic similarity between them (Task 1b). Task 2 - *Semantic Textual Similarity* (Agirre et al., 2015) involved measuring the similarity between sentence pairs from a variety of sources. Both tasks require participants to submit systems that will return for pairs of sentences a measure indicating the degree of similarity between their meanings. The connection between these and other tasks in computational semantics such as paraphrase detection and recognizing textual entailment will be briefly discussed in Section 8.2. Since experiments specific to the Twitter dataset were performed by Judit Ács, this thesis will not describe our submissions to Task 1 (the reader is referred to (Recski & Ács, 2015)), in the remainder of this section we shall focus on details of the three configurations submitted to the STS task as well as the experiments with `4lang`-based similarity performed by the author.

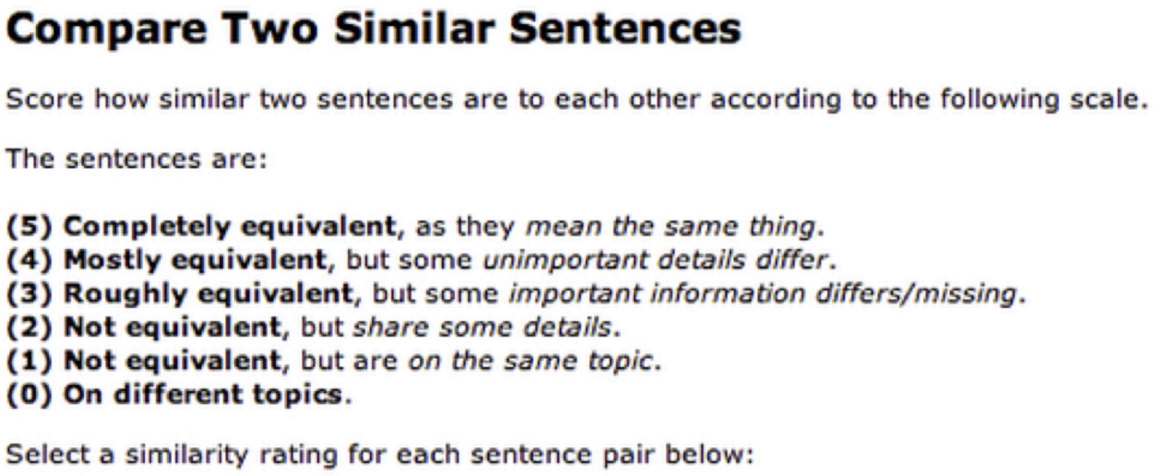
The methods used in state of the art systems to measure sentence similarity rely heavily on word similarity, typically derived from word embeddings (see Section 2.4). We demonstrate that a simple measure of similarity between `4lang` graphs is a competitive measure of word similarity. Our team, `MathLingBudapest`, submitted to Semeval 2015 systems that combine `4lang` similarity with features derived from various word embeddings, lexical resources like WordNet, and surface forms of words.

6.1.1 The STS task

The SemEval conferences, which organize shared tasks in various applications of computational semantics, have featured tracks on Semantic Textual Similarity (STS) every year since 2012. While the datasets used have changed annually, the task has remained unchanged in all evaluations: participating systems are expected to measure the degree of semantic similarity between pairs of sentences. Datasets used in recent years were taken from a variety of sources (news headlines, image captions, answers to questions posted in online forums, answers given by students in classroom tests, etc.). Gold annotation was obtained by crowdsourcing (using Amazon Mechanical Turk), annotators were required to grade sentence pairs on a scale from 0 to 5; Figure 6.1 shows the instructions they were given. Inter-annotator agreement was calculated to ensure the high quality of annotations.

6.1.2 Datasets

In 2015, STS systems were evaluated on a mixed dataset compiled from 5 sources: the `headlines` data contained titles of news articles gathered from several sources. The `images` dataset contained descriptions of images sampled from a set of 1000 images with 10 descriptions each. Half of sentence pairs were descriptions of the same image, the



Compare Two Similar Sentences

Score how similar two sentences are to each other according to the following scale.

The sentences are:

(5) Completely equivalent, as they *mean the same thing.*
(4) Mostly equivalent, but some *unimportant details differ.*
(3) Roughly equivalent, but some *important information differs/missing.*
(2) Not equivalent, but *share some details.*
(1) Not equivalent, but are *on the same topic.*
(0) On different topics.

Select a similarity rating for each sentence pair below:

Figure 6.1: Instructions for annotators of the STS datasets (Agirre et al., 2012, p.3)

other half described different ones. The `answers-student` dataset contains answers given by pupils to an automated tutoring system during a session on basic electronics. Pairs of one-sentence answers were selected based on string similarity. The `answers-forums` dataset contains pairs of responses from the StackExchange Q&A website; some pairs are responses to the same question, others were written in reply to different ones. Finally, the `belief` data contains pairs of user comments on online discussion forums. Pairs were sampled based on string similarity, then annotated and filtered based on inter-annotator agreement. For details on the origins of each dataset, see (Agirre et al., 2015).

6.1.3 Architecture of the `MathLingBudapest` systems

Our framework for measuring semantic similarity of sentence pairs is based on the system of (Han et al., 2013), who were among the top scorers in all STS tasks since 2013 (Kashyap et al., 2014; Han et al., 2015). Their architecture, *Align and Penalize*, involves computing an alignment score between two sentences based on some measure of word similarity. We have chosen to reimplement this system because it allowed us to experiment with various measures of word similarity, including those based on `4lang` graphs built by `dict_to_4lang`, which we shall present in Section 6.1.4. We reimplemented virtually all rules and components described by (Han et al., 2013) for experimentation but will now describe only those that ended up in at least one of the 3 configurations submitted to SemEval.

The core idea behind the *Align and Penalize* architecture is, given two sentences S_1 and S_2 and some measure of word similarity, to align each word of one sentence with some word of the other sentence so that the total similarity of aligned word pairs is maximized.

The mapping need not be one-to-one and is calculated independently for words of S_1, aligning them with words from S_2; and words of S_2, aligning them with words from S_1. The score of an alignment is the sum of the similarities of each word pair, normalized by sentence length, the final score assigned to a pair of sentences is the average of the alignment scores for each sentence.

In our top-scoring 2015 system we used supervised learning to establish the weights with which each source of word similarity contributes to the similarity score assigned to a pair of words. For out-of-vocabulary (OOV) words, i.e. those that are not covered by the component used for measuring word similarity, we rely on string similarity: we measure the Dice- and Jaccard-similarities (Dice, 1945; Jaccard, 1912) over the sets of character n-grams in each word for $n = 1, 2, 3, 4$. Additionally, we use simple rules to detect acronyms and compounds: if a word of one sentence that is a sequence of 2-5 characters (e.g. *ABC*) has a matching sequence of words in the other sentence (e.g. *American Broadcasting Company*), all words of the phrase are aligned with this word and receive an alignment score of 1. If a sentence contains a sequence of two words (e.g. *long term* or *can not*) that appear in the other sentence without a space and with or without a hyphen (e.g. *long-term* or *cannot*), these are also aligned with a score of 1.

The word similarity component can also be influenced by a boost feature based on WordNet (Miller, 1995). Scores are assigned if one word is a hypernym of the other, if one appears frequently in glosses of the other, or if they are derivationally related. For the exact cases covered and a description of how the boost is calculated, the reader is referred to (Han et al., 2013). The role these features play in measuring word similarity will be evaluated in Section 6.2 when we compare various configurations of the `wordsim` system.

The similarity score may be reduced by a variety of penalties, which we only enabled in our submission for Task 1 (Semantic Similarity in Twitter), they haven't improved our results on any other dataset (nor have they proved useful as features for the more recent `wordsim` system, to be described in Section 6.2). For a description of penalties used in the original *Align-and-Penalize* framework, the reader is referred to (Han et al., 2013), while (Recski & Ács, 2015) documents new penalties introduced for use with the Twitter dataset.

6.1.4 4lang-based similarity

The `4lang`-similarity of two words is the similarity between the `4lang` graphs defining them. We developed a measure of graph similarity by testing simple versions directly in our STS systems described in Section 6.1.3. Although this section describes a purely rule-based measure of word similarity, its components were later exposed to the ML-based

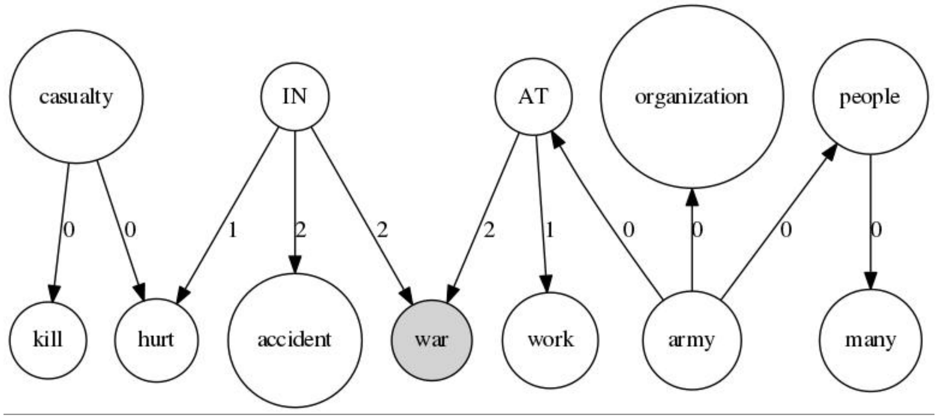


Figure 6.2: Overlap in the definitions of `casualty` (built from LDOCE) and `army` (defined in `4lang`)

`wordsim` system (see Section 6.2), a set of experiments providing more insight on their individual roles in measuring semantic similarity.

To define the similarity of two `4lang` graphs, we start by the intuition that similar concepts will overlap in the elementary configurations they take part in: they might share a 0-neighbor, e.g. `train` $\xrightarrow{0}$ `vehicle` $\xleftarrow{0}$ `car`, or they might be on the same path of 1- and 2-edges, e.g. `park` $\xleftarrow{1}$ `IN` $\xrightarrow{2}$ `town` and `street` $\xleftarrow{1}$ `IN` $\xrightarrow{2}$ `town`. For ease of notation we define the *predicates* of a node as the set of elementary configurations it takes part in. For example, based on the definition graph in Figure 3.3, we say that the predicates of the concept `bird` ($P(\text{bird})$) are {`vertebrate; (HAS, feather); (HAS, wing); (MAKE, egg)`}. Our initial version of graph similarity is the Jaccard similarity of the sets of predicates of each concept, i.e.

$$S(w_1, w_2) = J(P(w_1), P(w_2)) = \frac{|P(w_1) \cap P(w_2)|}{|P(w_1) \cup P(w_2)|}$$

Early experiments lead us to extend the definition of predicates by allowing them to be inherited via paths of 0-edges, e.g. `(HAS, wing)` is considered a predicate of all concepts for which $\xrightarrow{0}$ `bird` holds. We have also experimented with similarity measures that take into account the sets of all nodes accessible from each concept in their respective definition graph ($N(w)$). This proved useful in establishing that two concepts which would otherwise be treated as entirely dissimilar are in fact somewhat related. For example, given the definitions of the concepts `casualty` and `army` in Figure 6.2, the node `war` will allow us to assign nonzero similarity to the pair `(army, casualty)`. We achieved the best results on test data by using the maximum of these two scores as our word similarity measure.

Testing several versions of graph similarity on past years' STS data, we found that if two words w_1 and w_2 are connected by a path of 0-edges, it is best to assign to them a similarity of 1. This proved very efficient for determining semantic similarity of the most common types of sentence pairs in the SemEval datasets. Two descriptions of the same event (common in the *headlines* dataset) or the same picture (in *images*) will often only differ in their choice of words or choice of concreteness. In a dataset from 2014, for example, two descriptions, likely of the same picture, are *A bird holding on to a metal gate* and *A multi-colored bird clings to a wire fence.* Similarly, a pair of news headlines are *Piers Morgan questioned by police* and *Piers Morgan Interviewed by Police. wire* is by no means a synonym for *metal*, nor does being *questioned* mean exactly the same as being *interviewed*, but treating them as perfect synonyms proved to be an efficient strategy for the purpose of assigning similarity scores that correlate highly with human annotators' judgments.

6.1.5 Submissions

For Task 1 we submitted two systems: `twitter-embed` uses a single source of word similarity, a word embedding built from a corpus of word 6-grams from the Rovereto Twitter N-Gram Corpus[2] using the `gensim`[3] package's implementation of the method presented in (Mikolov, Chen, et al., 2013). Our second submission, `twitter-mash` combines similarities based on character ngrams, two word embeddings (built from 5-grams and 6-grams of the Rovereto corpus, respectively) and the `4lang`-based word similarity described in Section 6.1.4. For Task 2 (Semantic Textual Similarity) we were allowed three submissions. The `embedding` system uses a word embedding built from the first 1 billion words of the English Wikipedia using the `word2vec`[4] tool (Mikolov, Chen, et al., 2013). The `machine` system uses the word similarity measure described in Section 6.1.4 (both systems use the character ngram baseline as a fallback for OOVs). Finally, for the `hybrid` submission we combined these two systems and the character-similarity.

Evaluation

Our results on each task are presented in Tables 6.1 and 6.2. In case of Task 1a (Paraphrase Identification) our two systems performed equally in terms of F-score and ranked 30th among 38 systems. On Task 1b the hybrid system performed considerably better than the purely vector-based run, placing 11th out of 28 runs. On Task 2 our hybrid system

[2] `http://clic.cimec.unitn.it/amac/twitter_ngram/`

[3] `http://radimrehurek.com/gensim`

[4] `https://code.google.com/p/word2vec/`

	`embedding`	`hybrid`
Task 1a: *Paraphrase Identification*		
Precision	0.454	0.364
Recall	0.594	0.880
F-score	**0.515**	**0.515**
Task 1b: *Semantic Similarity*		
Pearson	0.229	**0.511**

Table 6.1: Performance of submitted systems on Task 1.

	`embedding`	`machine`	`hybrid`
answers-forums	0.704	0.698	0.723
answers-students	0.700	0.746	0.751
belief	0.733	0.736	0.747
headlines	0.769	0.805	0.804
images	0.804	0.841	0.844
mean Pearson	0.748	0.777	**0.784**

Table 6.2: Performance of submitted systems on Task 2a: *Semantic Similarity.*

ranked 11th among 78 systems, the systems using the word embedding and the `4lang`-based similarity alone (with string similarity as a fallback for OOVs in each case) ranked 22nd and 15th, respectively.

6.1.6 Difficulties

We have obtained from `4lang` graphs a measure of word similarity that we successfully combined with vector-based metrics to create a competitive STS system, but we cannot expect our metric to outperform distributional similarity on its own. Here we discuss some of the more typical issues that we encountered.

Lack of inferencing

Without performing some inference on the concept graphs built from dictionary definitions, the near-synonyms **wizard** - *a man who is supposed to have magic powers* and **magician** - *a man in stories who can use magic* will be assigned a score of only 0.182 by our system; a higher score is not warranted by the knowledge that both concepts refer to men and that both have some connection to magic. In this example the task is as difficult as realizing that the subgraphs $\texttt{X} \xleftarrow{1} \texttt{HAS} \xrightarrow{2} \texttt{power} \xrightarrow{0} \texttt{magic}$ and $\texttt{X} \xleftarrow{1} \texttt{CAN} \xrightarrow{2} \texttt{use} \xrightarrow{2} \texttt{magic}$ refer to roughly the same state-of-affairs. This kind of inference is beyond the system as currently implemented, but well within the capabilities of `4lang`, see (Kornai, in preparation) for a discussion.

OOVs

Another significant source of errors were out-of-vocabulary words (OOVs). Given the sources of input data, named entities (e.g. in `headlines`) and non-standard orthography (e.g. `forums`) are often unknown for both word embeddings and `4lang`. Character similarity can mitigate these effects significantly, but in the future we must reduce OOV-rates of all components, e.g. by training embeddings on larger datasets, building `4lang` definitions from additional resources (e.g. the Urban Dictionary) and by improving the quality of lemmatization.

6.2 Word similarity

The experiments described in Section 6.1 provided many insights about the potential of `4lang` representations to model semantic relatedness of concepts. This section will describe our more recent efforts at measuring the semantic similarity of word pairs, resulting in the hybrid `wordsim` system. The task of word similarity was attractive for several reasons: firstly, any method based on the `4lang` theory and using the representations created either manually or using the `dict_to_4lang` system described in Chapter 5 can provide feedback on the quality of these representations as well as the current shortcomings of the representation itself. Secondly, the word similarity task has been a standard method for evaluating distributional models of semantics (see Setion 2.4), with some models trained explicitly for this task (see Section 6.2.3). This section will present a system using supervised learning over features from multiple models (including both word embeddings and `4lang` representations). We relied on the standard SimLex-999 dataset[5] for training and evaluation, we'll introduce the dataset and summarize previous results in Section 6.2.1. The experimental setup is described in Section 6.2.2, external models used to generate features for our ML system will be listed in Section 6.2.3. Section 6.2.4 introduces the features defined over pairs of `4lang` definition graphs. Our results are presented in Section 6.2.5 along with a brief analysis of common errors. The `wordsim` library is available under an MIT license from `http://www.github.com/recski/wordsim`, the contents of this section are presented in greater detail by (Recski, Iklódi, et al., 2016).

[5] `http://www.cl.cam.ac.uk/~fh295/simlex.html`

6.2.1 Previous work

(Hill et al., 2015) recently proposed the SimLex-999 dataset as a benchmark for systems measuring word similarity. They argue that earlier gold standards measure *association*, not similarity, of word pairs; e.g. the words *cup* and *coffee* receive a high score by annotators in the widely used wordsim353 data (Finkelstein et al., 2002). Hill et al. note that "[a]ssociation and similarity are neither mutually exclusive nor independent" (2015, p.668). Instead of providing any definition of the above distinction, annotators of the SimLex dataset were simply shown a small set of examples and counter-examples. Since its publication in 2015 dozens of models have used the SimLex dataset for evaluation, some of these are listed on the SimLex webpage[6].

Various systems for measuring word similarity are compared using the SimLex dataset by measuring the Spearman correlation between scores assigned to word pairs by each system and the average of scores given by human annotators. Word embeddings are evaluated by several authors by treating the cosine distance of the pair of word vectors as the word similarity score assigned by that embedding to a pair of words. (Hill et al., 2015) report a correlation of 0.41 by an embedding trained on Wikipedia using word2vec (Mikolov, Chen, et al., 2013), (Schwartz et al., 2015) achieve a score of 0.56 using a combination of a standard word2vec-based embedding and the SP model, which encodes the cooccurrence of words in *symmetric patterns* such as *X and Y* or *X as well as Y*. (Banjade et al., 2015) document a set of experiments on the contribution of various models to the task of measuring word similarity. Half a dozen distributional models are combined with simple WordNet-based features indicating whether word pairs are synonymous or antonymous, and with the word similarity algorithm of (Han et al., 2013), which we briefly introduced in Section 6.1.3, and which itself uses WordNet-based features for boosting. By generating features using each of these resources and evaluating ML models trained using 11 different subsets of 10 feature classes, (Banjade et al., 2015) conclude that top performance is achieved when including all of them. This system achieved a Spearman correlation of 0.64, a considerable improvement over the performance of any individual model.

The highest scores on SimLex that we are aware of (other than our own system) is achieved using the Paragram embedding (Wieting et al., 2015), a set of vectors obtained by training pre-existing embeddings on word pairs from the Paraphrase Database (Ganitkevitch et al., 2013). The top correlation of 0.69 is measured when using a 300-dimension embedding created from the same GloVe-vectors that have been introduced in this section (trained on 840 billion tokens). Hyperparameters of this database have been

[6] http://www.cl.cam.ac.uk/~fh295/simlex.html

tuned for maximum performance on SimLex, another version tuned for the WS-353 dataset achieves a correlation of 0.67.

6.2.2 Setup

Our system is trained using several real-valued and binary features generated using various embeddings, WordNet, and 4lang definition graphs. Each class of features will be presented in detail below. We perform support vector regression with an RBF kernel (A. Smola & Vapnik, 1997; A. J. Smola & Schölkopf, 2004) over all features using the numpy library. Models are evaluated using tenfold cross-validation: in each iteration, we train a model on 900 pairs of the SimLex data and evaluate it on the remaining 99 pairs. We calculate the the Spearman correlation scores for each batch of 99 words, the average of these 10 scores is used as the standard figure of merit for any given model. As we introduce the feature classes used in our experiments in the next sections, we shall report these figures for all major configurations, and we conclude by summarizing all results in Section 6.2.5.

6.2.3 External models

Word embeddings

The largest class of features is based on word vector similarity. Each word embedding used in an experiment is represented by a single feature, the cosine similarity of the two vectors corresponding to a pair of words. Three sets of word vectors in our experiments were built using the neural models that have been evaluated on SimLex by (Hill et al., 2015): the SENNA[7] (Collobert & Weston, 2008), and Huang[8] (Huang et al., 2012) embeddings, which contain 50-dimension vectors and were downloaded from the authors' webpages, and word2vec (Mikolov, Chen, et al., 2013) vectors of 300 dimensions, trained on the Google News dataset[9].

We extend this set of models with a GloVe embedding [10] (Pennington et al., 2014) trained on 840 billion tokens of Common Crawl data[11], and also the two word embeddings mentioned in Section 6.2.1 : the 500-dimension SP model[12] (Schwartz et al., 2015) (see

[7] http://ronan.collobert.com/senna/

[8] http://www.socher.org

[9] https://code.google.com/archive/p/word2vec/

[10] http://nlp.stanford.edu/projects/glove/

[11] https://commoncrawl.org/

[12] http://www.cs.huji.ac.il/~roys02/papers/sp_embeddings/sp_embeddings.html

Section 6.2.1) and the 300-dimension Paragram vectors[13] (Wieting et al., 2015), both of which have recently been evaluated on the SimLex dataset yielding state of the art results. The model trained on these 6 features achieves a Spearman correlation of 0.72, the performance of individual embeddings is listed in Table 6.3.

System	Spearman's ρ
Huang	0.14
SENNA	0.27
GloVe	0.40
Word2Vec	0.44
SP	0.50
Paragram	0.68
6 embeddings	**0.72**

Table 6.3: Performance of word embeddings on SimLex

Wordnet

Another class of features are based on the lexical ontology WordNet (Miller, 1995), which we have briefly introduced in Section 2.2.6. WordNet-based metrics proved useful in the Semeval system of (Han et al., 2013), who use these metrics for calculating a boost of word similarity scores. The top system of (Banjade et al., 2015) also relies on a subset of these features. We chose to use four of these metrics as binary features in our system; these indicate whether one word is a direct or two-link hypernym of the other, whether the two are derivationally related, and whether one appears frequently in the glosses of the other, of its direct hypernym, or of its direct hyponyms. Each of the four features improved our system independently, icluding all of them brought the system's performance to 0.73. A model trained on WordNet features alone achieves a correlation of 0.33.

6.2.4 4lang-based features

Based on insights gained from developing a 4lang-based similarity measure for our 2015 STS system (see Section 6.1 for details), we defined multiple features over pairs of 4lang graphs which we predicted would correlate with word similarity. In defining these features we rely on the definition of *predicates* introduced in Section 6.1.4. Two real-valued features correspond to the main components of our earlier, rule-based measure: the Jaccard-similarities of sets of predicates and nodes in definition graphs. Additionally, we introduced

[13] http://ttic.uchicago.edu/~wieting/

feature	definition
`links_jaccard`	$J(P(w_1), P(w_2))$
`nodes_jaccard`	$J(N(w_1), N(w_2))$
`links_contain`	1 if $w_1 \in P(w_2)$ or $w_2 \in P(w_1)$, 0 otherwise
`nodes_contain`	1 if $w_1 \in N(w_2)$ or $w_2 \in N(w_1)$, 0 otherwise
`0_connected`	1 iff w_1 and w_2 are on a path of 0-edges, 0 otherwise

Table 6.4: `4lang` similarity features

three binary features. The `links_contain` feature is true iff either concept is contained in a predicate of the other, `nodes_contain` holds iff either concept is included in the other's definition graph, and `0_connected` is true iff the two nodes are connected by a path of 0-edges in either definition graph. All `4lang`-based features are listed in Table 6.4.

Initial experiments suggested that using these features as the only source of word similarity information result in many “false positives": e.g. pairs of antonyms in SimLex were regularly assigned high similarity scores because the above features are not sensitive to the `4lang` nodes `LACK`, representing negation (`dumb` $\xrightarrow{0}$ `intelligent` $\xrightarrow{0}$ `LACK`), and `BEFORE`, which indicates that something was only true in the past (`forget` $\xrightarrow{0}$ `know` $\xrightarrow{0}$ `BEFORE`),

We therefore proceeded to implement the `is_antonym` feature, a binary set to `true` iff one word is within the scope of, i.e. 0-connected to, an instance of either `LACK` or `BEFORE` in the other word's definition graph. Next, we transform each pair of graphs: all nodes within the scope of `LACK` or `BEFORE` are prefixed by `lack_` and are thus no longer considered identical with their non-negated counterparts when computing each of the features in Table 6.4. An example is shown in Figure 6.3.

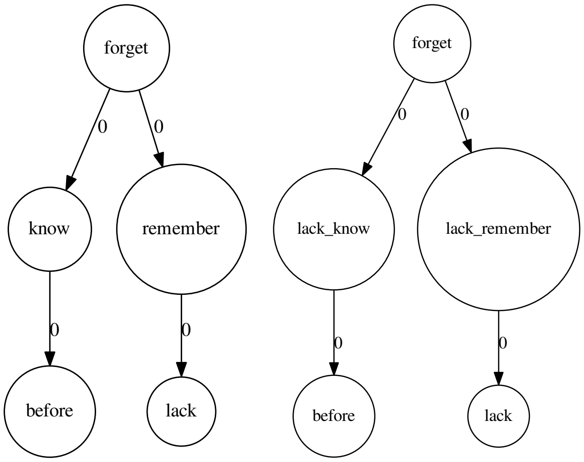


Figure 6.3: `4lang` definition of *forget* and its modified version

A system trained on 4lang-based features only achieves a Pearson correlation in the range of $0.32 - 0.34$ on the SimLex data, scores that were only slightly increased to 0.38 by the above treatment of LACK and BEFORE. While this score is competitive with some word embeddings, it is significantly below the $0.58 - 0.68$ range of the state of the art systems cited in Sections 6.2.1 and 6.2.3. By measuring the individual contribution of each type of 4lang feature to the performance of purely vector-based configurations, we discovered that only two types improve their performance significantly: 0-connected and is_antonym. Adding these two features to the vector-based system brings correlation to 0.75, the model using both 4lang and WordNet achieves our top score of 0.76.

6.2.5 Results

Performance of major wordsim configurations is presented in Table 6.5. The top system using only word embeddings achieves a Spearman correlation of 0.72. WordNet and 4lang features both improve this system, and combining all three feature classes yields our top correlation of 0.76, higher than any previous results that we are aware of. (Hill et al., 2015) report that the average correlation between a human rater and the average of all other raters is 0.78, suggesting that on this benchmark our system has achieved near-human performance.

System	Spearman's ρ
embeddings	0.72
embeddings+wordnet	0.73
embeddings+4lang	0.75
embeddings+wordnet+4lang	**0.76**

Table 6.5: Performance of major configurations on SimLex

In order to gain a better understanding of the shortcomings of our system, we sorted word pairs by the difference between gold similarity values from SimLex and the output of our top-scoring model. Errors made by wordsim are dominated by two distinct groups of word pairs. The largest group consists of word pairs that are nearly or completely synonymous but received low similarity scores from our model, Table 6.6 shows the top examples. The second group contains word pairs that were scored as highly similar by our model but not by human annotators (see Table 6.7 for the top examples). This second error class exemplifies a well-known issue with models of word similarity: (Hill et al., 2015) already observed that similarity of vectors in word embeddings tend to model association (or *relatedness*) rather than the similarity of concepts represented by each word.

word1	word2	output	gold	diff
bubble	*suds*	2.97	8.57	5.59
dense	*dumb*	1.71	7.27	5.56
cop	*sheriff*	3.50	9.05	5.55
alcohol	*gin*	3.43	8.65	5.22
rationalize	*think*	3.50	8.25	4.75

Table 6.6: Top 5 “false negative” errors

word1	word2	output	gold	diff
girl	*maid*	7.72	2.93	-4.79
happiness	*luck*	6.59	2.38	-4.21
crazy	*sick*	7.49	3.57	-3.92
arm	*leg*	6.74	2.88	-3.86
breakfast	*supper*	8.01	4.40	-3.61

Table 6.7: Top 5 “false positive” errors

To better understand the role `4lang` representations play in the performance of our system, we examined definition graphs of top erroneous word pairs. As expected, the `0-connected` feature was `False` for word pairs such as those in Table 6.6. In most cases the missing 0-edges (or 0-paths) could be added to the graphs using simple inference methods of the kind described in Section 3.3. For example, `suds` are defined in LDOCE as *the mass of bubbles formed on the top of water with soap in it*, yielding the `4lang` subgraph `bubble` $\xleftarrow{1}$ `HAS` $\xrightarrow{2}$ `mass` $\xleftarrow{0}$ `suds`. A simple rule stating that a *mass of X* inherits all predicates of `X`, would allow us to infer the edge `suds` $\xrightarrow{0}$ `bubble`

As discussed in Section 3.2 inference over `4lang` graphs should also derive all uses of polysemous words. The `4lang` representation of `dense` is built from its first definition in LDOCE: *made of or containing a lot of things or people that are very close together.* A method that will relate this definition with that of `dumb` is currently out of reach. Better short-term results could be obtained by using all definitions in a dictionary to build `4lang` representations, for `dense` this would include its third definition: *not able to understand things easily.* Other shortcomings of `4lang` representations are of a more technical nature. Currently the lemmatizer mapping words of definitions to concepts fails to map *alcoholic* to `alcohol` in the definition of `gin`: *a strong alcoholic drink made mainly from grain.* Yet other errors could be addressed by rewarding the overlap between two representations, e.g. that the graphs for `cop` and `sheriff` both contain $\xrightarrow{0}$ `officer`.

6.3 Natural language understanding

We now summarize our earliest application of the `4lang` representation, a dialogue system using spreading activation over `4lang`-machines, presented in detail in (Nemeskey et al., 2013). Two systems mimicking the actions of a ticket clerk at a Hungarian railway station (one selling tickets and another responding to timetable inquiries) use Eilenberg-machines – the formal objects behind `4lang` graphs that are viewed as directed graphs of concepts throughout this thesis – to represent user input at all levels of analysis. Words and chunks detected in user input are represented by machines, as are entire utterances after processing. User input is first processed by standard tools: a morphological analyzer (Tron et al., 2005) and an NP chunker (Recski & Varga, 2010). *Constructions* over machines take over in the next step, pairing surface structures with arbitrary actions, in this case filling slots of Attribute Value Matrices (AVMs) with domain-specific fields such as DESTINATION[14]. For example, when encountering *Gödre* ('to Göd'), a noun phrase in sublative case that also contains the name of a Hungarian town, the DESTINATION field can be populated.

Simple rules such as this one are responsible for storing domain-specific knowledge extracted from user input, but a domain-independent activation of machines corresponding to `4lang` concepts governs the actions taken by the system. For each concept found in the input, machines are added to the set of *active* machines and expanded, using either their `4lang` definitions (e.g. in the case of `ticket`) or an external dictionary storing domain-specific information, e.g. that `student` and `pensioner` can be synonyms for `half-price` in the context of train tickets. At every iteration of the activation process, concepts are also activated if all concepts in their definitions are active at the end of the previous iteration. Other interfaces of the system can activate machines and fill AVMs, e.g. the location of the user can activate the concepts `ticket` and `schedule`, and populate the ticket-AVM field SOURCE with the name of the station (which may later be overridden based on user input).

The system was built to respond perfectly to ca. 40 real-life dialogues – transcribed by the author over a 30-minute period at a Budapest railway station and informally referred to as the MÁV-corpus (MÁV is the largest railroad company in Hungary). Our system was never formally evaluated with human users, but was presented to the public, spawning considerable interest (Szedlák, 2012; nyest.hu, 2012). All code is available under an MIT license[15], but the system is no longer actively maintained.

[14] `Construction` objects in the `pymachine` module – a dependency of `4lang`– are not introduced in this thesis, but Section 8.4 will briefly mention some more applications. AVM filling is performed by subtypes of the `Operator` class, also not documented here.

[15] `http://www.github.com/kornai/pymachine/`

Chapter 7

System architecture

This chapter describes the main building blocks of the 4lang system. The most up-to-date version of this documentation is available under https://github.com/kornai/4lang/tree/master/doc. Besides introducing the main modules dep_to_4lang (Section 7.3) and dict_to_4lang (Section 7.4), which were introduced in Chapters 4 and 5 repsectively, this chapter also describes auxiliary components such as the Lemmatizer and Lexicon classes (Sections 7.6 and 7.5) as well as some modules of the pymachine library used by 4lang (Section 7.7). Section 7.2 lists the external dependencies of the 4lang module along with brief instructions on how to obtain and install them. The purpose of the short overview in Section 7.1 is to make this chapter accessible on its own, those who have read Chapters 3 through 6 of this thesis may safely skip it. Finally, Section 7.9 gives detailed instructions on how to customize each 4lang tool using configuration files.

7.1 Overview

The 4lang library provides tools to build and manipulate directed graphs of concepts that represent the meaning of words, phrases and sentences. 4lang can be used to

- build concept graphs from plain text (text_to_4lang)
- build concept graphs from dictionary definitions (dict_to_4lang)
- measure semantic similarity of concept graphs

Both text_to_4lang and dict_to_4lang rely on the Stanford CoreNLP (English) and the magyarlanc (Hungarian) toolchains for generating dependency relations from text, which are in turn processed by the dep_to_4lang module.

The top-level file 4lang contains a manually built concept dictionary, mapping ca. 3000 words to 4lang-style definition graphs. Graphs are specified using a simple human-readable format, partially documented in (Kornai et al., 2015) (a more complete description is forthcoming). Definitions in the 4lang dictionary can be processed using the definition_parser module of the pymachine library (see Section 7.7).

The text_to_4lang module takes as its input raw text, passes it to the Stanford CoreNLP package for dependency parsing and coreference resolution, than calls the dep_to_4lang module to convert the output into interconnected Machine instances. The dict_to_4lang tool builds graphs from dictionary definitions by extending the pipeline with parsers for several machine-readable monolingual dictionaries and some genre-specific preprocessing steps.

7.2 Requirements

7.2.1 pymachine

The pymachine library is responsible for implementing machines, graphs of machines, and some more miscellaneous tools for manipulating machines. The library is documented in Section 7.7. The library can be downloaded from http://www.github.com/kornai/pymachine and installed by running python setup.py install from the pymachine directory.

7.2.2 hunmorph and hundisambig

The lemmatizer class in 4lang, documented in Section 7.6 uses a combination of tools, two of which are the hunmorph open-source library for morphological analysis and the hundisambig tool for morphological disambiguation. The source code for both can be downloaded from http://mokk.bme.hu/en/resources/hunmorph/, the pre-built models for English and Hungarian, morphdb.en and morphdb.hu, are also made available. Alternatively, pre-compiled binaries for both hunmorph and hundisambig are available at http://people.mokk.bme.hu/~recski/4lang/huntools_binaries.tgz, they can be expected to work on most UNIX-based systems. The archive should be extracted in the 4lang working directory, which will create the huntools_binaries directory. If binaries need to be recompiled, they should also be copied to this directory, or the value of the parameter hunmorph_path must be changed in default.cfg to point to an alternative directory.

7.2.3 Stanford Parser and CoreNLP

4lang runs the Stanford Parser in two separate ways. When parsing dictionary definitions, the stanford_wrapper module launches the Jython-based module stanford_parser.py, which can communicate directly with the Stanford Parser API to enforce constraints on the parse trees (see Section 5.2.2 for details). These modules require the presence of the Stanford Dependency Parser, which can be obtained from http://nlp.stanford.edu/software/lex-parser.shtml#Download and the Jython tool, available from http://www.jython.org/downloads.html. After downloading and installing these tools, the ‘stanford’ and ‘corenlp’ sections of the default configuration file ‘conf/default.cfg’ must be updated so that the relevant fields point to existing installations of each tool and the englishRNN.ser.gz model (details on the config file will be given in Section 7.9).

The text_to_4lang tool, on the other hand, runs parsing as well as coreference resolution using the Stanford CoreNLP package. To save the overhead of loading multiple models each time text_to_4lang is run, CoreNLP is run using the corenlp-server tool, which takes care of downloading CoreNLP, then launching it and keeping it running in the background, allowing text_to_4lang to pass requests to it continuously. The corenlp-server tool can be downloaded from https://github.com/kowey/corenlp-server, then instructions in its README should be followed to launch the server.

7.3 dep_to_4lang

The core module for building 4lang graphs from text is the dep_to_4lang module which processes the output of dependency parsers. The text_to_4lang module only contains glue code for feeding raw text to Stanford CoreNLP and passing the output to dep_to_4lang. The dict_to_4lang module, which parses and preprocesses dictionary definitions before passing them to CoreNLP, will be described in the next section.

The dep_to_4lang module processes for each sentence the output of a dependency parser, i.e. a list of relations (or *triplets*) of the form $R(w_1, w_2)$, and optionally a list of coreferences, i.e. indications that a group of words in the sentence all refer to the same entity (this is currently available for English, using the Stanford Coreference Resolution system from the CoreNLP library). The configuration passed to the DepTo4lang class upon initialization must point to a file containing a map from dependencies to 4lang edges and/or binary relations. For English the default map is the dep_to_4lang.txt file in the project’s root directory.

The core method of the dep_to_4lang module is
DepTo4lang.get_machines_from_deps_and_corefs, which expects as its parameter not

just a list of dependencies but also the output of coreference resolution, which is called by `text_to_4lang` but not by `dict_to_4lang`. This function will ultimately return a map from surface word forms to `Machine` instances. To create machines, the function requires the dependencies to also contain each word's lemma - for Hungarian data these are extracted from the output of `magyarlanc` by `magyarlanc_wrapper`, for English data the `Lemmatizer` module is called (see Section 7.6). Dependency triplets are iterated over, Machines are instantiated for each lemma, and the `apply_dep` function is called for each triple of `(relation, machine1, machine2)`.

The `apply_dep` function matches such triplets against `Dependency` instances that have been created by parsing the `dep_to_4lang.txt` file containing the mapping from dependency relations to `4lang` configurations. In order to handle morphological features in Hungarian data, these patterns may make reference to the `MSD` labels of words which have also been extracted from the `magyarlanc` output. In case of a match, `Operators` associated with the dependency are run on the machines to enforce the specific configurations[1].

```
from collections import defaultdict
import json
import logging
import os
import re
import sys
import traceback

from pymachine.operators import AppendOperator, AppendToNewBinaryOperator,
    AppendToBinaryFromLexiconOperator  # nopep8

from dependency_processor import DependencyProcessor
from lemmatizer import Lemmatizer
from lexicon import Lexicon
from utils import ensure_dir, get_cfg, print_4lang_graphs

class DepTo4lang():

    dep_regex = re.compile("([a-z_-]*)\((.*?)-([0-9]*)'*, (.*?)-([0-9]*)'*\)")

    def __init__(self, cfg):
        self.cfg = cfg
        self.lang = self.cfg.get("deps", "lang")
        self.out_fn = self.cfg.get("machine", "definitions_binary_out")
        ensure_dir(os.path.dirname(self.out_fn))
        self.dependency_processor = DependencyProcessor(self.cfg)
        dep_map_fn = cfg.get("deps", "dep_map")
        self.read_dep_map(dep_map_fn)
```

[1] We do not document the `Operator` class, which is used to define complex actions over `Machines` that may be sensitive to some input data. In its current state the codebase makes no more use of them as it does of `Machines`: they are elaborate structures performing one or two very simple tasks; in this case, adding edges between machines. They do however play a significant role in the experimental system presented in Section 6.3 and will likely play a crucial part in `4lang`-based parsing (see Section 8.4).

```
        self.undefined = set()
        self.lemmatizer = Lemmatizer(cfg)
        self.lexicon_fn = self.cfg.get("machine", "definitions_binary")
        self.lexicon = Lexicon.load_from_binary(self.lexicon_fn)
        self.word2lemma = {}

    def read_dep_map(self, dep_map_fn):
        self.dependencies = defaultdict(list)
        for line in file(dep_map_fn):
            l = line.strip()
            if not l or l.startswith('#'):
                continue
            dep = Dependency.create_from_line(l)
            self.dependencies[dep.name].append(dep)

    def apply_dep(self, dep, machine1, machine2):
        dep_type = dep['type']
        msd1 = dep['gov'].get('msd')
        msd2 = dep['dep'].get('msd')
        if dep_type not in self.dependencies:
            if dep_type not in self.undefined:
                self.undefined.add(dep_type)
                logging.warning(
                    'skipping dependency not in dep_to_4lang map: {0}'.format(
                        dep_type))
            return False  # not that anyone cares
        for dep in self.dependencies[dep_type]:
            dep.apply(msd1, msd2, machine1, machine2)

    def dep_to_4lang(self):
        dict_fn = self.cfg.get("dict", "output_file")
        logging.info('reading dependencies from {0}...'.format(dict_fn))
        longman = json.load(open(dict_fn))
        for c, (word, entry) in enumerate(longman.iteritems()):
            if c % 1000 == 0:
                logging.info("added {0}...".format(c))
            try:
                if entry["to_filter"]:
                    continue
                if not entry['senses']:
                    # TODO these are words that only have pointers to an MWE
                    # that they are part of.
                    continue
                definition = entry['senses'][0]['definition']
                if definition is None:
                    continue
                deps = definition['deps']
                if not deps:
                    # TODO see previous comment
                    continue
                machine = self.get_dep_definition(word, deps)
                if machine is None:
                    continue

                # logging.info('adding: {0}'.format(word))
                # logging.info('ext_lex_keys: {0}'.format(
                    # self.lexicon.ext_lexicon.keys()))
                self.lexicon.add(word, machine)
            except Exception:
                logging.error(u"exception caused by: '{0}'".format(word))
                # logging.error(
                #     u'skipping "{0}" because of an exception:'.format(
                #         word))
                # logging.info("entry: {0}".format(entry))
                traceback.print_exc()
                sys.exit(-1)
                continue

        logging.info('added {0}, done!'.format(c + 1))

    def print_graphs(self):
        print_4lang_graphs(
            self.lexicon.ext_lexicon,
            self.cfg.get('machine', 'graph_dir'))

    def save_machines(self):
        self.lexicon.save_to_binary(self.out_fn)

    @staticmethod
    def parse_dependency(string):
        dep_match = DepTo4lang.dep_regex.match(string)
        if not dep_match:
            raise Exception('cannot parse dependency: {0}'.format(string))
        dep, word1, id1, word2, id2 = dep_match.groups()
        return dep, (word1, id1), (word2, id2)

    def get_root_lemmas(self, deps):
        return [
            d['dep'].setdefault(
                'lemma', self.lemmatizer.lemmatize(d['dep']['word'], uppercase=True))
            for d in deps if d['type'] == 'root']  # TODO

    def get_dep_definition(self, word, deps):
        deps = self.dependency_processor.process_dependencies(deps)
        root_lemmas = self.get_root_lemmas(deps)
        if not root_lemmas:
            logging.warning(
                u'no root dependency, skipping word "{0}"'.format(word))
            return None

        word2machine = self.get_machines_from_deps_and_corefs(
            [deps], [], process_deps=False)

        if word in word2machine:
            return word2machine[word]

        root_machines = filter(None, map(word2machine.get, root_lemmas))
        if not root_machines:
            logging.info("failed to find root machine")
            logging.info('root lemmas: {0}'.format(root_lemmas))
            logging.info('word2machine: {0}'.format(word2machine))
            sys.exit(-1)

        word_machine = self.lexicon.get_machine(word, new_machine=True)

        for root_machine in root_machines:
            word_machine.unify(root_machine)
            word_machine.append(root_machine, 0)
        return word_machine

    def get_machines_from_deps_and_corefs(
            self, dep_lists, corefs, process_deps=True):
        if process_deps:
            dep_lists = map(
                self.dependency_processor.process_dependencies, dep_lists)
        coref_index = defaultdict(dict)
        for (word, sen_no), mentions in corefs:
            for m_word, m_sen_no in mentions:
                coref_index[m_word][m_sen_no-1] = word

        # logging.info('coref index: {0}'.format(coref_index))

        word2machine = {}
        for deps in dep_lists:
            for dep in deps:
                for t in (dep['gov'], dep['dep']):
                    self.word2lemma[t['word']] = t.setdefault(
                        'lemma', self.lemmatizer.lemmatize(t['word'], uppercase=True))

        for i, deps in enumerate(dep_lists):
            try:
                for dep in deps:
                    word1 = dep['gov']['word']
                    word2 = dep['dep']['word']
                    # logging.info('dep: {0}, w1: {1}, w2: {2}'.format(
                    #     repr(dep), repr(word1), repr(word2)))
                    c_word1 = coref_index[word1].get(i, word1)
                    c_word2 = coref_index[word2].get(i, word2)

                    """
                    if c_word1 != word1:
                        logging.warning(
                            "unifying '{0}' with canonical '{1}'".format(
                                word1, c_word1))
                    if c_word2 != word2:
                        logging.warning(
                            "unifying '{0}' with canonical '{1}'".format(
                                word2, c_word2))
                    """
                    lemma1 = self.word2lemma[c_word1]
                    lemma2 = self.word2lemma[c_word2]

                    # TODO
                    # lemma1 = lemma1.replace('/', '_PER_')
                    # lemma2 = lemma2.replace('/', '_PER_')
                    # logging.info(
                    #      'lemma1: {0}, lemma2: {1}'.format(
                    #          repr(lemma1), repr(lemma2)))

                    for lemma in (lemma1, lemma2):
                        if lemma not in word2machine:
                            word2machine[lemma] = self.lexicon.get_machine(
                                lemma, new_machine=True)

                    self.apply_dep(
                        dep, word2machine[lemma1], word2machine[lemma2])

            except:
                logging.error(u"failure on dep: {0}({1}, {2})".format(
                    dep, word1, word2))
                traceback.print_exc()
                raise Exception("adding dependencies failed")

        return word2machine

class Dependency():
    def __init__(self, name, patt1, patt2, operators=[]):
        self.name = name
        self.patt1 = re.compile(patt1) if patt1 else None
        self.patt2 = re.compile(patt2) if patt2 else None
        self.operators = operators

    @staticmethod
    def create_from_line(line):
        rel, reverse = None, False
        # logging.debug('parsing line: {}'.format(line))
        fields = line.split('\t')
        if len(fields) == 2:
            dep, edges = fields
        elif len(fields) == 3:
            dep, edges, rel = fields
            if rel[0] == '!':
                rel = rel[1:]
                reverse = True
        else:
            raise Exception('lines must have 2 or 3 fields: {}'.format(
                fields))

        if ',' in dep:
            dep, patt1, patt2 = dep.split(',')
        else:
            patt1, patt2 = None, None

        edge1, edge2 = map(lambda s: int(s) if s not in ('-', '?') else None,
                           edges.split(','))

        if (dep.startswith('prep_') or
                dep.startswith('prepc_')) and rel is None:
            # logging.info('adding new rel from: {0}'.format(dep))
            rel = dep.split('_', 1)[1].upper()

        # Universal Dependencies
        if ((dep.startswith('acl:') and not dep.startswith('acl:relcl')) or
                dep.startswith('advcl:') or
                dep.startswith('nmod:')) and rel is None:
            logging.info('adding new rel from: {0}'.format(dep))
            rel = dep.split(':', 1)[1].upper()

        return Dependency(dep, patt1, patt2, Dependency.get_standard_operators(
            edge1, edge2, rel, reverse))

    @staticmethod
    def get_standard_operators(edge1, edge2, rel, reverse):
        operators = []
        if edge1 is not None:  # it can be zero, don't check for truth value!
            operators.append(AppendOperator(0, 1, part=edge1))
        if edge2 is not None:
            operators.append(AppendOperator(1, 0, part=edge2))
        if rel:
            operators.append(
                AppendToNewBinaryOperator(rel, 0, 1, reverse=reverse))

        return operators

    def match(self, msd1, msd2):
        for patt, msd in ((self.patt1, msd1), (self.patt2, msd2)):
            if patt is not None and msd is not None and not patt.match(msd):
                return False
        return True

    def apply(self, msd1, msd2, machine1, machine2):
        logging.debug(
            'trying {0} on {1} and {2}...'.format(self.name, msd1, msd2))
        if self.match(msd1, msd2):
            logging.debug('MATCH!')
            for operator in self.operators:
                operator.act((machine1, machine2))

def main():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s : " +
        "%(module)s (%(lineno)s) - %(levelname)s - %(message)s")
    cfg_file = sys.argv[1] if len(sys.argv) > 1 else None
    cfg = get_cfg(cfg_file)
    dep_to_4lang = DepTo4lang(cfg)
    dep_to_4lang.dep_to_4lang()
    dep_to_4lang.save_machines()
    dep_to_4lang.print_graphs()

if __name__ == "__main__":
    main()
```

7.4 dict_to_4lang

The dict_to_4lang module implements the pipeline that builds 4lang graphs from dictionary entries by connecting a variety of dictionary parsers, a module for preprocessing dictionary entries (EntryPreprocessor), and a custom wrapper for the Stanford Parser (stanford_parser.py) written in Jython that allows adding custom constraints to the parsing process. The output from dependency parsers is passed by dict_to_4lang to dep_to_4lang, the resulting graph of 4lang concepts is used to construct the definition graph for each headword in the dictionary, which are then saved using the Lexicon class (see Section 7.5).

```
from __future__ import with_statement
from collections import defaultdict
import json
import logging
import os
import sys
import threading
import time
import traceback

from dep_to_4lang import DepTo4lang
from entry_preprocessor import EntryPreprocessor
from lexicon import Lexicon
from longman_parser import LongmanParser
from wiktionary_parser import WiktParser
from stanford_wrapper import StanfordWrapper
from utils import batches, ensure_dir, get_cfg
from collins_parser import CollinsParser
from eksz_parser import EkszParser
from nszt_parser import NSzTParser
from magyarlanc_wrapper import Magyarlanc

assert Lexicon  # silence pyflakes (Lexicon must be imported for cPickle)

ONE_BY_ONE = False  # run threads after one another (to avoid memory issues)


class DictTo4lang():
    def __init__(self, cfg):
        self.dictionary = {}
        self.cfg = cfg
        self.output_fn = self.cfg.get('dict', 'output_file')
        ensure_dir(os.path.dirname(self.output_fn))
        self.tmp_dir = self.cfg.get('data', 'tmp_dir')
        ensure_dir(self.tmp_dir)
        self.graph_dir = self.cfg.get('machine', 'graph_dir')
        ensure_dir(self.graph_dir)
        self.get_parser_and_lang()
        self.machine_wrapper = None
    def get_parser_and_lang(self):
        input_type = self.cfg.get('dict', 'input_type')
        logging.info('input type: {0}'.format(input_type))
        if input_type == 'wiktionary':
            self.parser = WiktParser()
            self.lang = 'eng'
        elif input_type == 'longman':
            self.parser = LongmanParser()
            self.lang = 'eng'
        elif input_type == 'collins':
            self.parser = CollinsParser()
            self.lang = 'eng'
        elif input_type == 'eksz':
            self.parser = EkszParser()
            self.lang = 'hun'
        elif input_type == 'nszt':
            self.parser = NSzTParser()
            self.lang = 'hun'
        else:
            raise Exception('unknown input format: {0}'.format(input_type))

    def parse_dict(self):
        input_file = self.cfg.get('dict', 'input_file')
        self.raw_dict = defaultdict(dict)
        for entry in self.parser.parse_file(input_file):
            if 'senses' not in entry or entry['senses'] == []:
                continue  # todo
            self.unify(self.raw_dict[entry['hw']], entry)

    def unify(self, entry1, entry2):
        if entry1 == {}:
            entry1.update(entry2)
        elif entry1['hw'] != entry2['hw']:
            raise Exception(
                "cannot unify entries with different headwords: " +
                "{0} vs. {1}".format(entry1['hw'], entry2['hw']))

        # print 'entry1: ' + repr(entry1)
        # print 'entry2: ' + repr(entry2)
        entry1['senses'] += entry2['senses']

    def process_entries(self, words):
        entry_preprocessor = EntryPreprocessor(self.cfg)
        entries = map(entry_preprocessor.preprocess_entry,
                      (self.raw_dict[word] for word in words))

        if self.lang == 'eng':
            stanford_wrapper = StanfordWrapper(self.cfg)
            entries = stanford_wrapper.parse_sentences(
                entries, definitions=True)
        elif self.lang == 'hun':
            magyarlanc_wrapper = Magyarlanc(self.cfg)
            entries = magyarlanc_wrapper.parse_entries(entries)
        else:
            print 'incorrect lang'
        for entry in entries:
            if entry['to_filter']:
                continue
            word = entry['hw']
            for sense in entry['senses']:
                definition = sense['definition']
                if definition is None:
                    continue

            if word in self.dictionary:
                logging.warning(
                    "entries with identical headwords:\n{0}\n{1}".format(
                        entry, self.dictionary[word]))

                self.unify(self.dictionary[word], entry)
            else:
                self.dictionary[word] = entry

    def process_entries_thread(self, i, words):
        try:
            self.process_entries(words)
        except:
            self.thread_states[i] = False
            traceback.print_exc()
        else:
            self.thread_states[i] = True

    def run(self, no_threads=1):
        logging.info('parsing xml...')
        self.parse_dict()
        # print "\n".join(["\n".join(["{0}\t{1}".format(
        #                         w, d['definition']) for d in s['senses']])
        #                    for w, s in self.raw_dict.items()])
        # print self.raw_dict
        # sys.exit(-1)
        entries_per_thread = (len(self.raw_dict) / no_threads) + 1
        self.thread_states = {}
        # may turn out to be less then "no_threads" with small input
        started_threads = 0
        if ONE_BY_ONE:
            logging.warning('running threads one by one!')
        for i, batch in enumerate(batches(self.raw_dict.keys(),
                                  entries_per_thread)):

            if ONE_BY_ONE:
                logging.warning('running batch #{0}'.format(i))
                self.process_entries_thread(i, batch)
            else:
                t = threading.Thread(
                    target=self.process_entries_thread, args=(i, batch))
                t.start()
            started_threads += 1
        logging.info("started {0} threads".format(started_threads))
        while True:
            if len(self.thread_states) < started_threads:
                time.sleep(1)
                continue
            elif all(self.thread_states.values()):
                logging.info(
                    "{0} threads finished successfully".format(no_threads))
                break
            else:
                raise Exception("some threads failed")

    def read_dict(self):
        logging.info(
            'loading dict_to_4lang intermediate state from {0}'.format(
                self.output_fn))
        with open(self.output_fn, 'r') as dict_file:
            self.dictionary = json.load(dict_file)
        logging.info('done!')

    def print_dict(self, stream=None):
        if stream is None:
            with open(self.output_fn, 'w') as out:
                json.dump(self.dictionary, out)
        else:
            json.dump(self.dictionary, stream)


def main():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s : " +
        "%(module)s (%(lineno)s) - %(levelname)s - %(message)s")
    cfg_file = sys.argv[1] if len(sys.argv) > 1 else None
    no_threads = int(sys.argv[2]) if len(sys.argv) > 2 else 1
    cfg = get_cfg(cfg_file)

    dict_to_4lang = DictTo4lang(cfg)
    dict_to_4lang.run(no_threads)
    dict_to_4lang.print_dict()

    dep_to_4lang = DepTo4lang(cfg)
    dep_to_4lang.dep_to_4lang()
    dep_to_4lang.save_machines()
    dep_to_4lang.print_graphs()


if __name__ == '__main__':
    main()
```

7.4.1 Parsing dictionaries

`dict_to_4lang` supports 5 input data formats:

- an XML version of the Longman Dictionary of Contemporary English

- a typographer's tape version of the Collins COBUILD Dictionary from the ACL/DCI dataset (https://catalog.ldc.upenn.edu/LDC93T1)
- XML dumps of the English Wiktionary (https://dumps.wikimedia.org/enwiktionary/)
- an XML version of the *Magyar Nyelv Nagyszótára* (Hungarian)
- a preprocessed XML format of the *Magyar Értelmező Kéziszótár*. (Hungarian)

These datasets are processed by the modules `longman_parser, collins_parser, wiktionary_parser, nszt_parser,` and `eksz_parser`, respectively, three of which (`longman_parser, wiktionary_parser, eksz_parser`) are subclasses of the `xml_parser` module. Each parser extracts a dictionary containing a list of definitions for each headword, each with part-of-speech tag (where available), and possibly other data which is not currently used by `dict_to_4lang`. Parsers also perform format-specific preprocessing if necessary (e.g. replacing abbreviated forms of frequent words with their full form in Hungarian definitions). If run as standalone applications, all five parsers will print their output in human-readable format, useful for testing.

`xml_parser`

Methods common to the three XML-based formats are defined in the abstract superclass `XMLParser`:

```python
import re

class XMLParser():

    @staticmethod
    def section_pattern(tag):
        """Create (section) regex object."""
        pattern_string = "<{0}>(.*?)</{0}>".format(tag)
        return re.compile(pattern_string, re.S)  # S: . can be newline

    @staticmethod
    def tag_pattern(tag):
        """Create (tag) regex object."""
        pattern_string = "</?{0}>".format(tag)
        return re.compile(pattern_string, re.S)

    @staticmethod
    def iter_sections(tag, text):
        """Return list of tags in text."""
        return XMLParser.section_pattern(tag).findall(text)

    @staticmethod
    def get_section(tag, text):
        """Return the first group of tag in text."""
        match_obj = XMLParser.section_pattern(tag).search(text)
        return None if match_obj is None else match_obj.group(1)

    @staticmethod
    def remove_sections(tag, text):
        """Remove (section) tags from text."""
        return XMLParser.section_pattern(tag).sub("", text)

    @staticmethod
    def remove_tags(tag, text):
        """Remove (tag) tags from text."""
        return XMLParser.tag_pattern(tag).sub("", text)

    @staticmethod
    def parse_xml(data):
        raise NotImplementedError

    @classmethod
    def parse_file(cls, fn):
        """Open, read and decode the input file,
        then give it to the main parser class' 'parse_xml' method."""
        return cls.parse_xml(open(fn).read().decode('utf-8'))
```

`longman_parser`

Methods specific to the Longman dictionary are defined by the `LongmanParser` class:

```
#!/usr/bin/env python
# Module for reading Longman XML and producing JSON output

from collections import defaultdict
import json
import re
import sys

from xml_parser import XMLParser

assert json  # silence pyflakes

class LongmanParser(XMLParser):

    @staticmethod
    def add_suffixes(text):
        return re.sub("<SUFFIX>(.*?)</SUFFIX>", "\\1", text)

    @staticmethod
    def remove_extra_whitespace(text):
        if text is None:
            return None
        return " ".join(text.split()).strip()

    @staticmethod
    def clean_definition(definition):
        if definition is None:
            return definition
        for tag in ("TEXT", "NonDV", "REFHWD", "FULLFORM", "PRON",
                    "PronCodes", "ABBR"):
            definition = LongmanParser.remove_tags(tag, definition)
        for tag in ("REFSENSENUM", "REFHOMNUM", "GLOSS"):
            definition = LongmanParser.remove_sections(tag, definition)
        definition = LongmanParser.remove_extra_whitespace(definition)
        definition = LongmanParser.add_suffixes(definition)
        return definition

    @staticmethod
    def parse_sense(text):
        definition = LongmanParser.clean_definition(
            LongmanParser.get_section("DEF", text))
        full_form = LongmanParser.get_section("FULLFORM", text)
        return {"full_form": full_form, "definition": definition}

    @staticmethod
    def get_headword(entry_text):
        """Return the first group of "HWD" in entry_text"""
        return LongmanParser.remove_extra_whitespace(
            LongmanParser.get_section("HWD", entry_text))

    @staticmethod
    def get_pos(entry_text):
        return LongmanParser.remove_extra_whitespace(
            LongmanParser.get_section("POS", entry_text))

    @staticmethod
    def parse_entry(entry_text):
        """ """
        entry = {
            "hw": LongmanParser.get_headword(entry_text),
            "senses": map(
                LongmanParser.parse_sense,
                LongmanParser.iter_sections("Sense", entry_text)),
        }

        pos = LongmanParser.get_pos(entry_text)
        for sense in entry['senses']:
            sense['pos'] = pos

        hom_num = LongmanParser.get_section('HOMNUM', entry_text)
        if hom_num is not None:
            entry['hom_num'] = hom_num.strip()

        return entry

    @staticmethod
    def parse_xml(xml_text):
        """Give items of generator of "Entry" strings in xml_text to
        'parse_entry' method one by one."""
        for raw_entry in LongmanParser.iter_sections("Entry", xml_text):
            yield LongmanParser.parse_entry(raw_entry)
    @staticmethod
    def print_defs(longman_obj):
        for entry in longman_obj:
            for sense in entry['senses']:
                print u"{0}\t{1}".format(
                    entry['hw'], sense['definition']).encode("utf-8")

    @staticmethod
    def print_sorted_defs(longman_obj):
        index = defaultdict(list)
        for e in longman_obj:
            index[e['hw']].append(e)
        for hw in sorted(index.iterkeys()):
            for entry in index[hw]:
                for sense in entry['senses']:
                    print u"{0}\t{1}".format(
                        hw, sense['definition']).encode("utf-8")


if __name__ == "__main__":
    LongmanParser.print_sorted_defs(LongmanParser.parse_file(sys.argv[1]))
```

wiktionary_parser

Functions required to parse database dumps of the English Wiktionary (available at https://dumps.wikimedia.org/enwiki/) are defined by the WiktionaryParser class:

```
# simple parser for English Wiktionary
from HTMLParser import HTMLParser
import re
import sys

from xml_parser import XMLParser

class WiktParser(XMLParser):

    html_parser = HTMLParser()

    header_regex = re.compile("^=+([^=]*?)=+$", re.M)
    lang_section_regex = re.compile('==English==$.*', re.M | re.S)
    defs_section_regex = re.compile("^=+[^=$]*?=+$[^=]*?^#.*?^=", re.M | re.S)
    def_regex = re.compile("^#([^#:\*].*)", re.M)
    double_curly_regex = re.compile("{{.*?}}")
    replacements = [(re.compile(pattern), subst) for pattern, subst in [
        ("\[\[(.*?)\|(.*?)\]\]", "\\2")]]
    patterns_to_remove = [re.compile(pattern) for pattern in [
        "\[\[", "\]\]", "<ref>.*</ref>", "''", "''"]]

    pos_name_map = {  # entries with categories not listed shall be omitted
        'noun': 'n', 'proper noun': 'n', 'verb': 'v', 'adjective': 'adj',
        'adverb': 'adv', 'initialism': 'n', 'pronoun': 'n',
        'abbreviation': 'n', 'numeral': 'num', 'interjection': 'interj',
        'definitions': 'n',  # this means the POS is unknown
        'preposition': 'prp', 'conjunction': 'conj', 'acronym': 'n',
        'cardinal numeral': 'num', 'cardinal number': 'num', 'number': 'num',
        'article': 'art', 'particle': 'part', 'determiner': 'det', }

    @staticmethod
    def get_pages(text):
        return WiktParser.iter_sections('page', text)

    @staticmethod
    def get_pos(section):
        header = WiktParser.header_regex.match(section).group(1).lower()
        if header not in WiktParser.pos_name_map:
            # sys.stderr.write(header+'\n')
            return False
        return WiktParser.pos_name_map[header]

    @staticmethod
    def parse_definition(definition):
        d = definition.strip()
        # semi-colons usually separate two definitions on the same line
        d = d.split(';')[0]
        d = WiktParser.html_parser.unescape(d)
        d = WiktParser.double_curly_regex.sub('', d)
        for pattern, subst in WiktParser.replacements:
            d = pattern.sub(subst, d)
        for pattern in WiktParser.patterns_to_remove:
            d = pattern.sub("", d)

        # if a definition is longer than 300 characters, that's probably a bug
        # and it will cause memory errors when parsing
        d = d[:300]

        return d.strip()

    @staticmethod
    def get_definitions(section):
        raw_definitions = WiktParser.def_regex.findall(section)
        parsed_definitions = map(WiktParser.parse_definition, raw_definitions)
        kept_definitions = filter(None, parsed_definitions)
        return kept_definitions

    @staticmethod
    def parse_page(page):
        headword = WiktParser.get_section('title', page)
        if ":" in headword:
            return None

        lang_section = WiktParser.lang_section_regex.search(page)
        if lang_section is None:
            return None

        defs_section = WiktParser.defs_section_regex.search(
            lang_section.group())

        if defs_section is None:
            return None
        pos = WiktParser.get_pos(defs_section.group())
        if pos is False:
            return None

        definitions = WiktParser.get_definitions(defs_section.group())

        if not definitions:
            return None

        return {
            "hw": headword,
            "senses": [{
                "full_form": headword,
                "pos": pos,
                "definition": definition}
                for definition in definitions]}

    @staticmethod
    def parse_xml(xml):
        for page in WiktParser.get_pages(xml):
            parsed_page = WiktParser.parse_page(page)
            if parsed_page is not None:
                yield parsed_page


def test():
    xml = sys.stdin.read()
    for entry in WiktParser.parse_xml(xml):
        print entry

if __name__ == "__main__":
    test()
```

collins_parser

The `CollinsParser` class, contributed by Attila Bolevácz, parses the typographer's tape format of the 1979 edition of the Collins English Dictionary, fixed by Mark Liberman and made available by LDC as pasrt of the LDC/ACI collection (LDC93T1)[2]

```
import logging
import sys
import re


class CollinsParser():
    @staticmethod
    def print_definitions(definitions):
        for section in definitions:
            for sense in section['senses']:
                print "{0}\t{1}\t{2}".format(
```

[2] https://catalog.ldc.upenn.edu/LDC93T1

```
                    section['hw'], sense['pos'], sense['definition'])

    @staticmethod
    def parse_file(input_file):
        for section in re.split('#[hH]', CollinsParser.get_text(input_file)):
            try:
                yield CollinsParser.parse_entry(section)
            except:
                logging.warning("parse failed on section: {0}".format(section))

    @staticmethod
    def pattern_obj(pattern):
        return re.compile(pattern, re.S)

    @staticmethod
    def get_text(input_file):
        text = open(input_file).read().decode('utf-8')
        if text[:2] == '#h' or text[:2] == '#H':
            return text[2:]
        else:
            return text

    @staticmethod
    def parse_entry(entry):
        """Delete unnecessary marks
        and return entry in appropriate format."""
        if not entry.strip():
            return None
        from_ = ['@=', '\?&', '@!', ' esp.']
        to = ['-', '&', '!', ' especially']
        for f, t in zip(from_, to):
            entry = re.sub(f, t, entry)
        for pattern in ['\n', '@n']:
            entry = re.sub(pattern, " ", entry)
        alternate_forms = CollinsParser.get_alternate_forms(entry)
        for pattern in ['#\+', '@\.', '\?!',
                        'or #3[^ ]+']:  # '#3' another spelling
            entry = re.sub(pattern, "", entry)
        for pattern in ['#5\(.*?\)', '#5\[.*?\]']:
            entry = re.sub(pattern, '#5', entry)
        hw, description = CollinsParser.get_hw(entry)
        return {
            'hw': hw,
            'senses': CollinsParser.get_senses(description),
            'alternate_forms': alternate_forms}

    @staticmethod
    def get_alternate_forms(entry):
        forms = re.findall('#3(.*?)#[56]', entry)
        return [
            form.replace('#+', '').replace('@.', '').replace('#4', '').strip()
            for form in forms]

    @staticmethod
    def get_pos(entry):
        # first #6 except #6or
        match = re.search('#6(?!or)(.+?)[. ]', entry, re.S)
        if match:
            return match.group(1)
        else:
            return 'unknown'

    @staticmethod
    def get_hw(entry):
        """Return headword."""
        match = re.search('(.+?)#[56](.+)', entry, re.S)
        hw = match.group(1).replace('#4', '').strip()
        description = match.group(2)
        return hw, description

    @staticmethod
    def get_senses(entry):
        """Return sense(s)."""
        if '#1$D' in entry:
            return CollinsParser.del_pronunciation(
                CollinsParser.get_multiple_senses(entry))
        else:
            return CollinsParser.del_pronunciation(
                CollinsParser.get_mono_sense(entry))

    @staticmethod
    def del_pronunciation(lst_of_senses):
        for sense in lst_of_senses:
            if sense['definition'][0] == '(':
                re.sub('\(.*?\)', '', sense['definition'], count=1)
#         print 'without pronunciation: ' + repr(lst_of_senses)
        return lst_of_senses

    @staticmethod
    def get_mono_sense(description):
        def_and_pos = CollinsParser.separate_def_and_pos(description)
        definition = def_and_pos[1]
        if not definition:
            return []
        pos = def_and_pos[0]
        return [{'definition': definition,
                 'pos': pos}]
    pos_and_def_patt = re.compile(
        '(.*)#6(n|adj|vb|tr|adv|intr|abbrev|pl|interj|prep|prefix|determiner|pron|conj|
            suffix)\.(.*)')  # nopep8

    @staticmethod
    def separate_def_and_pos(description):
        """Return a tuple of pos and definition of a sense"""
        pos_and_def = CollinsParser.pos_and_def_patt.search(description)
        if pos_and_def:
            pos, definition = pos_and_def.group(2), pos_and_def.group(
                1) + pos_and_def.group(3)
        else:
            pos, definition = 'unknown', description
        definition = definition.strip('.,').strip().replace(
            '#5', '').replace('#4', '').strip('.')
        unnecessary = ['^#6[^ ]*', '#1a', '#6']
        for patt in unnecessary:
            definition = re.sub(patt, '', definition).strip()
        definition = re.sub('@m.*', '', definition).strip('.').strip()
        return pos, definition

    @staticmethod
    def get_multiple_senses(description):
        lst = []
        def_part = ''  # This corrects unnecessary splitting
        pos_for_multiple_senses = 'unknown'
        for sense in unicode.split(description, '#1$D'):
            if def_part:
                sense = def_part + sense
            def_and_pos = CollinsParser.separate_def_and_pos(sense)
            definition = def_and_pos[1]
            if not definition:
                def_part = sense
                continue
            else:
                def_part = ''
            pos = def_and_pos[0]
            if pos == 'unknown':
                pos = pos_for_multiple_senses
            else:
                pos_for_multiple_senses = pos
            lst.append({'definition': definition, 'pos': pos})
        return lst

if __name__ == "__main__":
    CollinsParser.print_definitions(CollinsParser.parse_file(sys.argv[1]))
```

`nszt_parser`

The `NSZTParser` class processes an XML format of a single volume of *A Magyar Nyelv Nagyszótára*, made available to the author by editor-in-chief Nóra Ittzés:

```
#!usr/bin/python
# -*- coding: utf-8 -*-

import sys
import re
# import json
import textwrap


class NSzTParser():
    @staticmethod
    def print_definitions(definitions):
#       with open('magyar_out.json', 'w') as out:
```

```
#           json.dump(None, out)
#       for section in definitions:
#           if section != None:
##                print section
#               with open('magyar_out.json', 'a') as out:
#                   json.dump(section, out)
        for section in definitions:
#            if section is not None:
#               print 'start'
            print
#            print "section: " + str(section)
            print section['hw'].encode('utf-8')
            if 'redirect' in section:
                print textwrap.fill(
                    'redirect: ' + section['redirect'],
                    initial_indent='    ',
                    subsequent_indent='        ').encode('utf-8')
            if 'senses' in section:
                for sense in section['senses']:
                    if 'latin' in sense:
                        print textwrap.fill(
                        'latin: ' + sense['latin'],
                        initial_indent='    ',
                        subsequent_indent='        ').encode('utf-8')
                    print textwrap.fill(
                        sense['definition'],
                        initial_indent='    ',
                        subsequent_indent='        ').encode('utf-8')
#               print
#               print 'end'

    @staticmethod
    def parse_file(input_file):
#        for line in iter(open(input_file)):
        for entry in re.finditer('<entry.+?<lemma>.+?</lemma>.*?</entry',
            # avoid entries with empty lemmas
            open(input_file).read().decode('utf-8').strip()):
                yield NSzTParser.parse_entry(entry.group(0))


    @staticmethod
    def parse_entry(entry):
#       print 'type of entry: ' + str(type(entry))
#        if entry[:6] == '<entry':
#            entry_dict = {'hw': NSzTParser.get_hw(entry),
#                          'senses': NSzTParser.get_senses(entry)}
#        else:
#            entry_dict = None
#        if entry[:8] == '<entryxr':
#            entry_dict['redirect'] = NSzTParser.get_xr(entry)
#        return entry_dict

        entry_dict = {'hw': NSzTParser.get_hw(entry)}
        if entry[:8] == '<entryxr':
            entry_dict['redirect'] = NSzTParser.get_xr(entry)
        else:
            entry_dict['senses'] = NSzTParser.get_senses(entry)
# xr?
        return entry_dict

    @staticmethod
    def get_hw(entry):
        hw = re.search('<lemma>(.+?)</lemma>', entry, re.S).group(1)
        tags = ['<hom>[1-9]</hom>', '</?deduced>', '</?reflex>']
        for tag in tags:
            hw = re.sub(tag, '', hw)
        return hw

    @staticmethod
    def get_senses(entry):
        hw = NSzTParser.get_hw(entry)
        if hw[0] == '-' or hw[-1] == '-':  # elotag/utotag
            return [{'definition': NSzTParser.clean_definition(re.search(
                '<def>(.+?)</def>', entry).group(1))}]

        raw_sense_list = re.findall(
            '<mainsens>.*?<def>(.*?)</def>.*?</mainsens>', entry)
        modified_sense_list = []
        for sense in raw_sense_list:
            if sense != '<same/>':
                modified_sense_list.append(
                    {'definition': NSzTParser.clean_definition(sense)})
                if '<tr>' in sense:
                    modified_sense_list[-1]['latin'] = NSzTParser.get_latin(
                        sense)
        return modified_sense_list

    @staticmethod
    def get_xr(entry):
        redirect = re.search('<xr>(.+?)</xr>', entry).group(1)
        return re.sub('<hom>[1-9]</hom>', '', redirect)

    @staticmethod
    def get_latin(sense):
        latin = re.search('<tr>(.+?)</tr>', sense).group(1)
        latin = re.sub('</?sub>', '', latin)
        return latin

    @staticmethod
    def clean_definition(definition):
        tags = ['gloss', 'mention', 'syn', 'tr>.+?</tr', 'hom>[1-9]</hom',
            'sub', 'syn special="no"', 'mean']
        for tag in tags:
            definition = re.sub('</?' + tag + '>', ' ', definition)

        definition = ' ' + definition
        before = ['</?hint>', '<syn special="semicolon">',
            '<syn special="comma">', '<syn special="ill">',
            '<syn special="v">', ' es\.', ' gyakr\.', ' haszn\.', ' ill\.',
            ' kapcs\.', u' k\xf6l\.', ' rendsz\.', ' ritk\.', ' v\.',
            'vonatk\.', u'\xe1llt\.', 'vm', 'vki', 'mn ', 'fn ', 'pl.',
            u'\xfan.', ' {2,}', ' ,']
        after = ['', '; ', ', ', ' illetve ', ' vagy ', ' esetleg', ' gyakran',
            u'haszn\xe1lt', ' illetve', ' kapcsolatos', u'k\xfcl\xf6n\xf6sen',
            ' rendszerint', u'ritk\xe1bban', ' vagy', u'vonatkoz\xf3',
            u'\xe1ltal\xe1ban', ' valam', ' valaki', u'mell\xe9kn\xe9v ', u'f\u0151n\
                xe9v ',
            u'p\xe9ld\xe1ul', u'\xfagynevezett', ' ', ',']
        # places of last two items are important
        for b, a in zip(before, after):
            definition = re.sub(b, a, definition)

#        definition = re.sub('</?hint>', '', definition)
#        definition = re.sub('<syn special="semicolon">', '; ', definition)
#        definition = re.sub('<syn special="comma">', ', ', definition)
#        definition = re.sub('<syn special="ill">', ' illetve ', definition)
#        definition = re.sub(' {2,}', ' ', definition)
#        definition = re.sub(' ,', ',', definition)
        return definition.strip()

    @staticmethod
    def sub(string, pattern, repl):
        pass


if __name__ == "__main__":
    for input_file in sys.argv[1:]:
        NSzTParser.print_definitions(NSzTParser.parse_file(input_file))
```

eksz_parser

Finally, the `EKSZParser` class processes an interim format of *Magyar Értelmező Kéziszótár*, created by Márton Miháltz:

```
#!/usr/bin/env python
# Module for reading Longman XML and producing JSON output

import json
import sys

from xml_parser import XMLParser

assert json  # silence pyflakes

u'\xe1rv\xedzt\u0171r\u0151 t\xfck\xf6rf\xfar\xf3g\xe9p\n'

class EkszParser(XMLParser):
    abbreviations = [
        (u"mo.-i", u'magyarorsz\xe1gi'),
        (u"Mo.-on", u'Magyarorsz\xe1gon'),
        (u"Mo.-hoz", u'Magyarorsz\xe1ghoz'),
        (u"Mo.", u'Magyarorsz\xe1g'),
        (u"bp.-i", u"budapesti"),
    (u"vonatk.", u"vonatkoz\xf3"),
    (u"vki", u"valaki"),
    (u"Vki", u"Valaki"),
    (u"vmi", u"valami"),
    (u"Vmi", u"Valami"),
    (u"vhol", u"valahol"),
    (u"Vhol", u"Valahol"),
    (u"vhonnan", u"valahonnan"),
    (u"Vhonnan", u"Valahonnan"),
    (u"vmely", u"valamely"),
    (u"Vmely", u"Valamely"),
    (u"vmilyen", u"valamilyen"),
    (u"Vmilyen", u"Valamilyen"),
    (u"kapcs.", u"kapcsolatos"),
    (u"kif-", u"kifejez\xe9s"),
    (u"haszn.", u"haszn\xe1lt"),
    (u".,", u";"),
    (u".;", u";"),
    (u"..", u"."),
    (u".a.", u"a."),  # a single line in the data
    (u"sz.-", u"sz\xe1zad"),
    (u"sz\xf3haszn-\xe1ban", u"sz\xf3haszn\xe1lat\xe1ban"),
    (u" (Na2SO4.10H2O)", u""),
    (u" (CaSO4.2H2O)", u""),
    (u" MgSO4.7H2O", u""),
    (u" KAlSO42.12H2O", u""),
    (u"jan.", u"janu\xe1r"),
    (u"J\xfan.-", u"J\xfanius"),
    (u"j\xfan.-", u"j\xfanius"),
    (u"aug.", u"augusztus"),
    (u"szept.-", u"szeptember"),
    (u"okt.-", u"okt\xf3ber"),
    (u"dec. ", u"december "),
    (u"h.:", u"hogy:"),
    (u" h. ", u" helyett "),
    (u" film.a ", u" film a "),
]

@staticmethod
def parse_headword(sense):
    hw = EkszParser.get_section("LEMMA", sense)
    hom_num = int(EkszParser.get_section("HOM", hw))
    hw = EkszParser.remove_sections("HOM", hw)
    return hw, hom_num

@staticmethod
def clean_definition(d):
    for a, b in EkszParser.abbreviations:
        d = d.replace(a, b)
    return d

@staticmethod
def parse_sense(sense):
    hw, hom_num = EkszParser.parse_headword(sense)
    definition = EkszParser.get_section('DEF', sense)
```

```
        definition = EkszParser.clean_definition(definition)
        pos = EkszParser.get_section('POS', sense)
        return hw, hom_num, pos, definition

    @staticmethod
    def get_entries(xml_text):
        completed_hws = set()
        curr_hw = None
        curr_pos = None
        curr_senses = []
        for sense in EkszParser.iter_sections("SENSE", xml_text):
            hw, hom_num, pos, definition = EkszParser.parse_sense(sense)
            if hom_num > 1:
                continue  # temporary solution
            if curr_hw is None:  # first line
                curr_hw = hw
            elif curr_hw != hw:
                if hw in completed_hws:
                    sys.stderr.write(
                        "INPUT NOT SORTED BY HW: {0}\n".format(hw).encode(
                            'utf-8'))
                    sys.exit(-1)
                else:
                    completed_hws.add(hw)

                yield {
                    "hw": curr_hw,
                    "pos": curr_pos if curr_pos is not None else pos,
                    "senses": curr_senses}

                curr_pos = pos  # we'll use the pos of the first occurence
                curr_hw = hw
                curr_senses = []

            curr_senses.append({"definition": definition})
        yield {
            "hw": curr_hw,
            "pos": curr_pos if curr_pos is not None else pos,
            "senses": curr_senses}

    @staticmethod
    def parse_xml(xml_text):
        """Give items of generator of "Entry" strings in xml_text to
        'parse_entry' method one by one."""
        for entry in EkszParser.get_entries(xml_text):
            yield entry

    @staticmethod
    def print_defs(eksz_obj):
        for entry in eksz_obj:
            for sense in entry['senses']:
                print u"{0}\t{1}".format(
                    entry['hw'], sense['definition']).encode("utf-8")
```

```
if __name__ == "__main__":
    # EkszParser.print_defs(EkszParser.parse_file(sys.argv[1]))
    with open(sys.argv[2], 'w') as out_file:
        json.dump(list(EkszParser.parse_file(sys.argv[1])), out_file)
```

7.4.2 Preprocessing entries

The output from parsing dictionary data is passed to the `EntryPreprocessor` module, which performs various steps that clean and simplify data before it is passed to external syntactic parsers. This module defines a list of regex patterns to be removed or replaced in definitions, and each pattern can be associated with one or more flags that are added to the entry if a replacement took place. It is therefore straightforward to define, given a new datasource, rules that will e.g. remove the string *of person* from a definition and simultaneously add the flag `person` to the entry being processed. The preprocessor also performs sentence tokenization (via `nltk.punkt`) and by default keeps only the first sentence of the first definition for each headword (but see Section 7.9 on how to change this).

```
from collections import defaultdict
import logging
import re

from unidecode import unidecode

from hunmisc.xstring.encoding import encode_to_proszeky
import nltk.data

assert logging, unidecode  # silence pyflakes


class EntryPreprocessor():
    word_replacement_pairs = [
        (re.compile(patt, re.UNICODE), repl) for patt, repl in [
            (u'/', u'_PER_'), (u'\?', u'_Q_'), (u'\.', u'_P_'),
            (u'\(', u'_LRB_'), (u'\)', u'_RRB_')]]
    def_replacement_pairs = [
        (re.compile(patt, re.UNICODE), repl, flags) for patt, repl, flags in [
            (u'([^,]) etc', u'\\1, etc', ()),  # comma before etc.
            (u'someone or something that is ', u'', ()),
            (u'someone or something that ', u'', ()),
            (u'someone who is ', u'', ('person',)),
            (u'someone who ', u'', ('person',)),
            (u'someone whose job is ', u'', ('person',)),
            (u'^someone ', u'', ('person',)),
            (u'( *)a kind of ', u'\\1a ', ()),
            (u'( *)a type of ', u'\\1a ', ()),
            (u'=', u'', ()),
        ]]

    @staticmethod
```

```
def clean_headword(word):
    clean = encode_to_proszeky(word)
    # clean = unidecode(clean)  #will map different words together!
    for pattern, replacement in EntryPreprocessor.word_replacement_pairs:  # nopep8
        clean = pattern.sub(replacement, clean)
    return clean

def __init__(self, cfg):
    self.cfg = cfg
    for package in ('stopwords', 'punkt'):
        nltk.download(package, quiet=True)
    self.sent_detector = nltk.data.load('tokenizers/punkt/english.pickle')
    self.word_counter = defaultdict(int)

def preprocess_word(self, orig_word, orig_definition=None):
    word = EntryPreprocessor.clean_headword(orig_word)
    return word, set()

def preprocess_definition(self, orig_definition, word):
    all_flags = set()
    if orig_definition is None:
        return orig_definition, all_flags
    definition = self.sent_detector.tokenize(orig_definition)[0]
    for pattern, replacement, flags in EntryPreprocessor.def_replacement_pairs:  #
        nopep8
        if pattern.search(definition):
            all_flags |= set(flags)
        definition = pattern.sub(replacement, definition)

    return definition, all_flags

def preprocess_entry(self, entry):
    if self.cfg.getboolean('filter', 'first_only'):
        entry['senses'] = entry['senses'][:1]
    entry['to_filter'] = self.to_filter(entry['hw'])
    if entry['to_filter']:
        return entry
    entry['hw'], entry['word_flags'] = self.preprocess_word(entry['hw'])
    entry['word_flags'] = sorted(list(entry['word_flags']))
    for sense in entry['senses']:
        sense['definition'], sense['flags'] = self.preprocess_definition(
            sense['definition'], entry['hw'])
        sense['flags'] = sorted(list(sense['flags']))

    return entry

def to_filter(self, word, definition=None):
    if ' ' in word and not self.cfg.getboolean(
            'filter', 'keep_multiword'):
        return True
    if "'" in word and not self.cfg.getboolean(
            'filter', 'keep_apostrophes'):
        return True
    return False
```

7.4.3 Parsing definitions

Definitions returned by `EntryPreprocessor` are passed to one of two external tools for dependency parsing: the Stanford Parser for English definitions and the `magyarlanc` tool for Hungarian, both accessed via the python wrappers `stanford_wrapper.py` and `magyarlanc_wrapper.py`. Both wrappers use the `Subprocess` module to launch external tools; `magyarlanc` is launched directly and the Stanford Parser is used via a Jython wrapper.

Parser wrappers

Since the `dict_to_4lang` module requires access to the Stanford Parser's API (see below for details), a wrapper (`stanford_parser.py`) was written in Jython, a Java implementation of the Python interpreter that allows direct access to Java classes from Python code. Access to the Stanford Parser API is necessary to pass custom *constraints* to the parser before processing sentences, limiting the types of possible parse trees. Currently this feature is used to enforce that dictionary definitions of nouns get parsed as noun phrases (NPs). When using the `parse_definitions` function for parsing, part-of-speech tags for each entry are passed to the `get_constraints` function, which returns a list of `ParserConstraint` instances – currently a list of length 0 or 1 (more ParserConstraints can be created from regex `Patterns`).

```
import json
import logging
import math
import os
import sys
from tempfile import NamedTemporaryFile

parser = sys.argv[1]
sys.path.append(parser)
sys.path.append(os.path.join(os.path.dirname(parser), 'ejml-0.23.jar'))

from edu.stanford.nlp.process import Morphology, PTBTokenizer, WordTokenFactory
from edu.stanford.nlp.parser.common import ParserConstraint
from edu.stanford.nlp.parser.lexparser import Options
from edu.stanford.nlp.parser.lexparser import LexicalizedParser
from edu.stanford.nlp.ling import Sentence
from edu.stanford.nlp.trees import PennTreebankLanguagePack

from java.io import StringReader
from java.util.regex import Pattern

class StanfordParser:

    @staticmethod
    def get_constraints(sentence, pos):
        constraints = []
        length = len(sentence)
        if pos == 'n':
            constraints.append(
                ParserConstraint(0, length, Pattern.compile("NP.*")))
        return constraints

    def __init__(self, parser_file,
                 parser_options=['-maxLength', '80',
                                 '-retainTmpSubcategories']):

        """@param parser_file: path to the serialised parser model
            (e.g. englishPCFG.ser.gz)
        @param parser_options: options
        """

        assert os.path.exists(parser_file)
        options = Options()
        options.setOptions(parser_options)
        self.lp = LexicalizedParser.getParserFromFile(parser_file, options)
        tlp = PennTreebankLanguagePack()
        self.gsf = tlp.grammaticalStructureFactory()
        self.lemmer = Morphology()
        self.word_token_factory = WordTokenFactory()
        self.parser_query = None

    def tokenize(self, text):
        reader = StringReader(text)
        tokeniser = PTBTokenizer(reader, self.word_token_factory, None)
        tokens = tokeniser.tokenize()
        return tokens

    def get_parse(self, sentence):
        tokens = [unicode(x) for x in self.tokenize(sentence)]
        parse = self.lp.apply(Sentence.toWordList(tokens))
        return parse

    def get_grammatical_structure(self, parse):
        return self.gsf.newGrammaticalStructure(parse)

    def get_kbest(self, query, k=3):
        for candidate_tree in query.getKBestPCFGParses(k):
            parse = candidate_tree.object()
            prob = math.e ** candidate_tree.score()
            yield prob, parse

    def parse(self, sentence):
        return self.parse_with_constraints(sentence, None)

    def parse_with_constraints(self, sentence, constraints):
        # logging.debug("getting query...")
        query = self.lp.parserQuery()
        if constraints is not None:
            query.setConstraints(constraints)
        # logging.debug("tokenizing...")
        toks = self.tokenize(sentence)
        # logging.debug("running parse...")
        query.parse(toks)
        # logging.debug("getting best...")
        parse = query.getBestParse()
        # logging.debug("getting gs...")
        gs = self.get_grammatical_structure(parse)
        # dependencies = gs.typedDependenciesCollapsed()
        dependencies = gs.typedDependenciesCCprocessed()
        return parse, gs, dependencies

    def parse_sens(self, in_file, out_file, log=False):
        logging.debug("reading input...")
        with open(in_file) as in_obj:
            sens = json.load(in_obj)
        parsed_sens = []
        if log:
            log_file = NamedTemporaryFile(dir="/tmp", delete=False)
        for c, sentence in enumerate(sens):
            if log and c % 100 == 0:
                log_file.write("parsed {0} sentences\n".format(c))
                log_file.flush()
            parse, _, dependencies = self.parse(sentence)

            dep_strings = map(unicode, dependencies)
            parsed_sens.append({
                'sen': sentence,
                'deps': dep_strings})

        with open(out_file, 'w') as out:
            json.dump(parsed_sens, out)

    def parse_definitions(self, in_file, out_file):
        with open(in_file) as in_obj:
            logging.info("loading input...")
            entries = json.load(in_obj)
            logging.info("done!")
        with NamedTemporaryFile(dir="/tmp", delete=False) as log_file:
            logging.info('logging to {0}'.format(log_file.name))
            for c, entry in enumerate(entries):
                # log_file.write(
                #     'entry: {0}\n'.format(entry['hw']).encode('utf-8'))
                # log_file.flush()
                if c % 100 == 0:
                    log_file.write("parsed {0} entries\n".format(c))
                    log_file.flush()
                for sense in entry['senses']:
                    sentence = sense['definition']
                    if sentence is None:
                        continue
                    # sentence += '.'  # fixes some parses and ruins others
                    pos = sense['pos']
                    constraints = StanfordParser.get_constraints(sentence, pos)
                    try:
                        parse, _, dependencies = self.parse_with_constraints(
                            sentence, constraints)
                    except:
                        sys.stderr.write(
                            u'parse failed on sentence: {0}'.format(
                                sentence).encode('utf-8'))
                        dep_strings = []
                    else:
                        dep_strings = map(unicode, dependencies)

                    sense['definition'] = {
                        'sen': sentence,
                        'deps': dep_strings}

        with open(out_file, 'w') as out:
            json.dump(entries, out)

def test():
    logging.warning("running test, not main!")
    parser = StanfordParser(sys.argv[2])

    # dv_model = parser.lp.reranker.getModel()
    # print dv_model

    # sentence = 'the size of a radio wave used to broadcast a radio signal'
    sentence = 'a man whose job is to persuade people to buy his company\'s \
        products.'
    pos = 'n'
    parse, gs, dependencies = parser.parse_with_constraints(
        sentence, StanfordParser.get_constraints(sentence, pos))

    print type(parse), type(gs)
    print parse.pennPrint()
    print "\n".join(map(str, dependencies))

def main():
    parser_file, in_file, out_file, is_defs, loglevel = sys.argv[2:7]
    logging.basicConfig(
        level=int(loglevel),
        format="%(asctime)s : " +
        "%(module)s (%(lineno)s) - %(levelname)s - %(message)s")
    logging.info("initializing parser...")
    parser = StanfordParser(parser_file)
    logging.info("done!")
    if int(is_defs):
        parser.parse_definitions(in_file, out_file)
    else:
        parser.parse_sens(in_file, out_file)

if __name__ == "__main__":
    main()
    # test()
```

The Jython module `stanford_parser.py` is not to be confused with the python module `stanford_wrapper.py`: the latter can be imported by any Python application and will launch a Jython session running the former.

```
from ConfigParser import ConfigParser
import json
import logging
import os
import requests
import subprocess
from subprocess import Popen, PIPE
import sys
from tempfile import NamedTemporaryFile

from utils import ensure_dir

class StanfordWrapper():

    http_request_headers = {
        'Content-type': 'application/json', 'Accept': 'text/plain'}

    class ParserError(Exception):
        pass

    def __init__(self, cfg, is_server=False):
        self.cfg = cfg
        remote = self.cfg.getboolean('stanford', 'remote')
        if is_server or not remote:
            self.get_stanford_paths()
            if is_server:
                # used as server
                self.start_parser()
                self.parse_sentences = self.parse_sentences_server
            else:
                # standalone, using jython
                self.get_jython_paths()
                self.parse_sentences = self.parse_sentences_local
        else:
            # used as client
            self.server_url = self.cfg.get('stanford', 'url')
            self.parse_sentences = self.parse_sentences_remote

    def get_stanford_paths(self):
        self.stanford_dir = self.cfg.get('stanford', 'dir')
        parser_fn = self.cfg.get('stanford', 'parser')
        self.model_fn = self.cfg.get('stanford', 'model')
        self.parser_path = os.path.join(self.stanford_dir, parser_fn)
        self.model_path = os.path.join(self.stanford_dir, self.model_fn)
        if not (os.path.exists(self.parser_path) and
                os.path.exists(self.model_path)):
            raise Exception("cannot find parser and model files!")

    def get_jython_paths(self):
        self.jython_path = self.cfg.get('stanford', 'jython')
        if not os.path.exists(self.jython_path):
            raise Exception("cannot find jython executable!")

        self.jython_module = os.path.join(
            os.path.dirname(__file__), "stanford_parser.py")

        self.tmp_dir = self.cfg.get('data', 'tmp_dir')
        ensure_dir(self.tmp_dir)

    def start_parser(self):
        command = [
            'java', '-mx1500m', '-cp', '{0}/*:'.format(self.stanford_dir),
            'edu.stanford.nlp.parser.lexparser.LexicalizedParser',
            '-outputFormat', 'typedDependenciesCollapsed',
            '-sentences', 'newline',
            'edu/stanford/nlp/models/lexparser/{0}'.format(self.model_fn),
            '-']

        logging.info(
            "starting stanford parser with this command: {0}".format(
                ' '.join(command)))

        self.parser_process = Popen(command, stdin=PIPE, stdout=PIPE)

    def parse_sentences_server(self, sens, definitions=False):
        parsed_sens = []
        for c, sentence in enumerate(sens):
            parsed_sens.append({'sen': sentence, 'deps': []})
            # logging.info('writing to stdin...')
            self.parser_process.stdin.write(sentence+'\n')
            self.parser_process.stdin.flush()
            # logging.info('reading from stdout...')
            line = self.parser_process.stdout.readline().strip()
            while line:
                # logging.info('read this: {0}'.format(repr(line)))
                if line == '':
                    break
                parsed_sens[-1]['deps'].append(line.strip())
                line = self.parser_process.stdout.readline().strip()

        # logging.info('returning parsed sens')
        return parsed_sens

    def create_input_file(self, sentences, token):
        sen_file = NamedTemporaryFile(
            dir=self.tmp_dir, prefix=token, delete=False)
        for sen in sentences:
            # need to add a period so the Stanford Parser knows where
            # sentence boundaries are. There should be a smarter way...
            sen_file.write(
                u"{0}\n".format(sen['sen']).encode('utf-8'))

        return sen_file.name

    def run_parser(self, in_file, out_file, definitions):
        return_code = subprocess.call([
            self.jython_path, self.jython_module, self.parser_path,
            self.model_path, in_file, out_file, str(int(definitions)),
            str(logging.getLogger(__name__).getEffectiveLevel())])
        return return_code == 0

    def parse_sentences_old(self, sentences):
        """sentences should be a list of dictionaries, each with a "sen" key
        whose value will be parsed, a "deps" key whose value is a list for
        collecting dependencies, and a "pos" key that may map to constraints on
        the parse"""
        with NamedTemporaryFile(dir=self.tmp_dir, delete=False) as in_file:
            json.dump(sentences, in_file)
            in_file_name = in_file.name
        with NamedTemporaryFile(dir=self.tmp_dir, delete=False) as out_file:
            success = self.run_parser(in_file_name, out_file.name)
            if not success:
                logging.critical(
                    "jython returned non-zero exit code, aborting")
                raise StanfordWrapper.ParserError()
            parsed_sentences = json.load(out_file)
        sentences.update(parsed_sentences)
        return True

    def parse_sentences_remote(self, entries, definitions=False):
        req = requests.get(
            self.server_url, data=json.dumps(entries),
            headers=StanfordWrapper.http_request_headers)

        return json.loads(req.text)

    def parse_sentences_local(self, entries, definitions=False):
        with NamedTemporaryFile(dir=self.tmp_dir, delete=False) as in_file:
            json.dump(entries, in_file)
            in_file_name = in_file.name
        logging.info("dumped input to {0}".format(in_file_name))

        with NamedTemporaryFile(dir=self.tmp_dir, delete=False) as out_file:
            out_file_name = out_file.name
            logging.info("writing parses to {0}".format(out_file_name))
            success = self.run_parser(in_file_name, out_file_name, definitions)

        if not success:
            logging.critical(
                "jython returned non-zero exit code, aborting")
            raise StanfordWrapper.ParserError()

        logging.debug("reading output...")
        with open(out_file_name) as out_file:
            new_entries = json.load(out_file)

        return new_entries

def main_flask(wrapper):
    from flask import Flask, request, Response
    app = Flask(__name__)

    @app.route("/")
    def hello():
        sens = request.get_json()
        # logging.info('got this: {0}'.format(sens))
        parsed_sens = wrapper.parse_sentences(sens)
        # logging.info('returning response...')
        # logging.info('returning this: {0}'.format(parsed_sens))
        return Response(json.dumps(parsed_sens), mimetype='application/json')

    app.run()

TEST_DATA = [
    ("rawhide", "leather that is in its natural state", "n"),
    ("playback", "the playback of a tape that you have recorded is when you play it on a
        machine in order to watch or listen to it", "n"),  # nopep8
    ("playhouse", "a theatre - used in the name of theatres", "n"),
    ("extent", "used to say how true something is or how great an effect or change is", "
        n"),  # nopep8
    ("indigenous", "indigenous people or things have always been in the place where they
        are, rather than being brought there from somewhere else", "n"),  # nopep8
    ("off-street", "places for parking that are not on public streets", "n"),
    ("half-caste", "a very offensive word for someone whose parents are of different
        races.", "n"),  # nopep8
    ("concordant", "being in agreement or having the same regular pattern", "n"),  #
        nopep8
    ("groundsman", "a man whose job is to take care of a large garden or sports field", "
        n")  # nopep8
]
def test(wrapper):

    entries = [{"hw": w,
                "senses": [{
                    "definition": d, "pos": "a" if n else 'a', "flags": []}]}
               for w, d, n in TEST_DATA]
    entries += [{
        "hw": "wombat",
        "senses": [{
            "definition": "an Australian animal like a small bear whose babies\
                live in a pocket of skin on its body",
            "pos": "n",
            "flags": []}]}]

    parsed_entries = wrapper.parse_sentences(
        entries, definitions=True)
    print json.dumps(parsed_entries)

def main():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s : " +
        "%(module)s (%(lineno)s) - %(levelname)s - %(message)s")

    cfg_file = 'conf/default.cfg' if len(sys.argv) < 2 else sys.argv[1]
    cfg = ConfigParser()
    cfg.read([cfg_file])

    wrapper = StanfordWrapper(cfg)
    test(wrapper)


if __name__ == '__main__':
    main()
```

The `magyarlanc` library is run directly as a subprocess launched by the `Magyarlanc` class, which also processes the parser's output to obtain dependencies as well as morphological information.

```
import logging
import os
import subprocess
from StringIO import StringIO
import sys
from tempfile import NamedTemporaryFile
import traceback

from hunmisc.corpustools.tsv_tools import sentence_iterator, get_dependencies


class Magyarlanc():
    def __init__(self, cfg):
        self.jarpath = cfg.get('magyarlanc', 'jar')
        self.magyarlanc_dir = cfg.get('magyarlanc', 'dir')
        self.tmp_dir = cfg.get('data', 'tmp_dir')

    def dump_entries(self, entries):
        logging.info('dumping to file...')
        with NamedTemporaryFile(dir=self.tmp_dir, delete=False) as in_file:
            for e in entries:
                definition = e['senses'][0]['definition']
                definition = definition.replace('i. e.', 'i.e.')  # TODO
                in_file.write(u"{0}\n".format(definition).encode('utf-8'))
                in_file_name = in_file.name
        logging.info("dumped input to {0}".format(in_file_name))
        return in_file_name

    def dump_text(self, text):
        logging.info('dumping to file...')
        with NamedTemporaryFile(dir=self.tmp_dir, delete=False) as in_file:
            t = text.replace('i. e.', 'i.e.')  # TODO
            in_file.write(t.encode('utf-8'))
            in_file_name = in_file.name
        logging.info("dumped input to {0}".format(in_file_name))
        return in_file_name

    def run_parser(self, in_file_name):
        os.chdir(self.magyarlanc_dir)
        with NamedTemporaryFile(dir=self.tmp_dir, delete=False) as out_file:
            return_code = subprocess.call([
                'java', '-Xmx2G', '-jar', self.jarpath,
                '-mode', 'depparse', '-input', in_file_name,
                '-output', out_file.name])
            if return_code == 0:
                return out_file.name
            return None

    @staticmethod
    def lines_to_deps(lines):
        text_str = u"\n".join((u"".join(sen) for sen in list(lines)))
        tsv_stream = StringIO(text_str)
        return map(get_dependencies, sentence_iterator(tsv_stream))

    def add_deps(self, entry, lines):
        deps = Magyarlanc.lines_to_deps([lines])[0]
        entry['senses'][0]['definition'] = {
            "sen": entry['senses'][0]['definition'],
            "deps": deps}

    def parse_text(self, text):
        in_file_name = self.dump_text(text)
        raw_parses = self.parse_file(in_file_name)
        deps = Magyarlanc.lines_to_deps(raw_parses)
        return deps, []

    def parse_entries(self, entries):
        in_file_name = self.dump_entries(entries)
        raw_parses = self.parse_file(in_file_name)
        for count, parse in enumerate(raw_parses):
            try:
                self.add_deps(entries[count], parse)
            except:
                logging.error("count: {0}".format(count))
                logging.error("last entry: {0}".format(entries[count-1]))
                logging.error(u"failed with: {0}".format(parse))
                traceback.print_exc()
                sys.exit(-1)
        return entries

    def parse_file(self, in_file_name):
        logging.info('parser input: {0}'.format(in_file_name))
        out_file_name = self.run_parser(in_file_name)
        logging.info('parser output: {0}'.format(out_file_name))
        if out_file_name is None:
            logging.error('parser failed')
            sys.exit(-1)
        count = 0
        curr_lines = []
        for line in open(out_file_name):
            if line == '\n':
                yield curr_lines
                curr_lines = []
                count += 1
            else:
                curr_lines.append(line.decode('utf-8'))


def test():
    import sys
    from utils import get_cfg
    cfg = get_cfg(sys.argv[1])
    m = Magyarlanc(cfg)
    test_sens = ["valamely asztalon vagy padon az ablakra illesztett keret"]
    # test_sens = ["Egy", "Hat", "Nyolc"]
    for sen in test_sens:
        for line in m.tag(sen):
            print line

if __name__ == '__main__':
    test()
```

7.4.4 Processing dependencies

The language-specific postprocessing of dependencies described in Section 4.5 takes place in the `dependency_processor` module. The `DependencyProcessor` class defines one or more functions for each of the processing steps described, these functions take as their input instances of either the `Dependencies` or the `NewDependencies` class. The `Dependencies` class is deprecated, new functions should support the `NewDependencies` class.

```
from collections import defaultdict
from copy import deepcopy
import logging
import re

class Dependencies():
    dep_regex = re.compile("(.*?)\((.*?)-([0-9]*)'*, (.*?)-([0-9]*)'*\)")

    @staticmethod
    def parse_dependency(string):
        dep_match = Dependencies.dep_regex.match(string)
        if not dep_match:
            raise Exception('cannot parse dependency: {0}'.format(string))
        dep, word1, id1, word2, id2 = dep_match.groups()
        return dep, (word1, id1), (word2, id2)

    @staticmethod
    def create_from_strings(dep_strings):
        dep_list = map(Dependencies.parse_dependency, dep_strings)
        return Dependencies(dep_list)

    def __init__(self, dep_list):
        self.dep_list = dep_list
        self.index_dependencies(dep_list)

    def index_dependencies(self, deps):
        self.index = defaultdict(lambda: (defaultdict(set), defaultdict(set)))
        deps = [(dep, tuple(w1), tuple(w2)) for dep, w1, w2 in deps]
        for triple in deps:
            self.add(triple)

    def remove(self, (dep, word1, word2)):
        self.index[word1][0][dep].remove(word2)
        self.index[word2][1][dep].remove(word1)

    def add(self, (dep, word1, word2)):
        self.index[word1][0][dep].add(word2)
        self.index[word2][1][dep].add(word1)

    def get_dep_list(self, exclude=[]):
        dep_list = []
        for word1, (dependants, _) in self.index.iteritems():
            for dep, words in dependants.iteritems():
                if any(dep.startswith(patt) for patt in exclude):
                    continue
                for word2 in words:
                    dep_list.append((dep, word1, word2))
        return dep_list

    def get_root(self):
        root_words = self.index[(u'ROOT', u'0')][0]['root']
        if len(root_words) != 1:
            logging.warning('no unique root element: {0}'.format(root_words))
            return None
        return iter(root_words).next()

    def merge(self, word1, word2, exclude=[]):
        for dep, w1, w2 in self.get_dep_list(exclude=exclude):
            if w1 in (word1, word2) and w2 in (word1, word2):
                pass
            elif w1 == word1:
                self.add((dep, word2, w2))
            elif w1 == word2:
                self.add((dep, word1, w2))
            elif w2 == word1:
                self.add((dep, w1, word2))
            elif w2 == word2:
                self.add((dep, w1, word1))
            else:
                pass


class NewDependencies():

    @staticmethod
    def create_from_old_deps(old_deps):
        deps = []
        for d_type, gov, dep in old_deps.get_dep_list():
            deps.append({
                "type": d_type,
                "gov": {
                    "word": gov[0],
                    "id": gov[1]},
                "dep": {
                    "word": dep[0],
                    "id": dep[1]}})
        return NewDependencies(deps)
    def __init__(self, deps):
        self.deps = deps
        self.indexed = False
        self.index()

    def index(self):
        self.tok_index = defaultdict(lambda: [None, [], []])
        self.dep_index = defaultdict(list)
        for d in self.deps:
            self.tok_index[d['gov']['id']][0] = d['gov']
            self.tok_index[d['dep']['id']][0] = d['dep']
            self.tok_index[d['gov']['id']][1].append(d)
            self.tok_index[d['dep']['id']][2].append(d)
            self.dep_index[d['type']].append(d)

        self.indexed = True

    def add(self, d_type, gov, dep):
        self.deps.append({"type": d_type, "gov": gov, "dep": dep})
        self.indexed = False

    def remove_tok(self, i):
        self.deps = [
            d for d in self.deps
            if d['gov']['id'] != i and d['dep']['id'] != i]
        self.indexed = False

    def remove_type(self, d_type):
        self.deps = [d for d in self.deps if d['type'] != d_type]
        self.indexed = False

class DependencyProcessor():
    copulars = set([
        "'s", 'are', 'be', 'been', 'being', 'is', 's', 'was', 'were'])

    def __init__(self, cfg):
        self.cfg = cfg
        self.lang = self.cfg.get("deps", "lang")

    def process_coordination_stanford(self, deps):
        for word1, word_deps in deepcopy(deps.index.items()):
            for i in (0, 1):
                for dep, words in word_deps[i].iteritems():
                    if dep.startswith('conj_'):
                        for word2 in words:
                            deps.merge(word1, word2, exclude=['conj_'])
                    elif dep.startswith('conj:'):
                        for word2 in words:
                            deps.merge(word1, word2, exclude=['conj:'])
        return deps

    def process_coordinated_root(self, deps):
        root_word = deps.get_root()
        for i in (0, 1):
            for dep, words in deepcopy(deps.index[root_word][i]).iteritems():
                if dep.startswith('conj_'):
                    for word in words:
                        deps.merge(word, root_word, exclude=['conj_'])
                elif dep.startswith('conj:'):
                    for word in words:
                        deps.merge(word, root_word, exclude=['conj:'])
        return deps

    def process_rcmods(self, deps):
        # rcmods = [
        #     (w1, w2) for w1, (dependants, _) in deps.index.iteritems()
        #     for dep, words in dependants.iteritems()
        #     for w2 in words if dep == 'rcmod']
        return deps

    def process_negation(self, deps):
        for dep in deps.get_dep_list():
            dtype, w1, w2 = dep
            if dtype == 'neg' and w2[0] != 'not':
                deps.remove(dep)
                deps.add((dtype, w1, ('not', w2[1])))
        return deps

    def process_copulars(self, deps):
        # nsubj(is, x), prep_P(is, y) -> prep_P(x, y)
        # rcmod(x, is), prep_P(is, y) -> prep_P(x, y)
        copulars = [(word, w_id) for word, w_id in deps.index.iterkeys()
                    if word in DependencyProcessor.copulars]
        new_deps = []
        for cop in copulars:
            if 'nsubj' in deps.index[cop][0]:
                for dep, words in deps.index[cop][0].iteritems():
                    if dep.startswith('prep_'):
                        for word2 in words:
                            new_deps += [
                                (dep, word3, word2)
                                for word3 in deps.index[cop][0]['nsubj']]
            if 'rcmod' in deps.index[cop][1]:
                for dep, words in deps.index[cop][0].iteritems():
                    if dep.startswith('prep_'):
                        for word2 in words:
                            new_deps += [
                                (dep, word3, word2)
                                for word3 in deps.index[cop][1]['rcmod']]
        for new_dep in new_deps:
            # logging.info('adding new dep: {0}'.format(new_dep))
            deps.add(new_dep)
        return deps

    def remove_copulars(self, deps):
        for dep, word1, word2 in deps.get_dep_list():
            if (word1[0] in DependencyProcessor.copulars or
                    word2[0] in DependencyProcessor.copulars):
                deps.remove((dep, word1, word2))
        return deps

    def process_conjunction_magyarlanc(self, deps):
        # for all conj(x, conj), for all D(conj, y) create D(x, y)
        # where conj in (hogy, de)
        # get conj dependants of conj relations
        conjs = set((
            d['dep']['id']
            for d in deps.dep_index['conj']
            if d['dep']['lemma'] in ('hogy', 'de')))
        # then for each of these:
        for conj in conjs:
            # get all their governors
            govs = [
                d['gov']
                for d in deps.tok_index[conj][2] if d['type'] == 'conj']
            # then for all dependents of hogy,
            for dep in deps.tok_index[conj][1]:
                # copy each dependency to each governor
                for gov in govs:
                    deps.add(dep['type'], gov, dep['dep'])

            deps.remove_tok(conj)
        deps.index()
        return deps

    def process_copulars_magyarlanc(self, deps):
        # mapping all pairs of the form nsubj(x, c) and pred(c, y)
        # (such that c is a copular verb) to the relation subj(x, y)
        pred_gov_cop_ids = [
            d['gov']['id'] for d in deps.dep_index['pred']
            if d['gov']['lemma'] == 'van']
        for gov_id in pred_gov_cop_ids:
            subj_deps = [d['dep'] for d in deps.tok_index[gov_id][1]]
            for subj in subj_deps:
                preds = [
                    d['dep'] for d in deps.tok_index[gov_id][1]
                    if d['type'] == 'pred']
                for pred in preds:
                    deps.add("subj", subj, pred)
            deps.remove_tok(gov_id)
        deps.index()
        return deps

    def process_coordination_magyarlanc(self, deps):
        # get governors of coord relations
        govs = set((d['gov']['id'] for d in deps.dep_index['coord']))
        # then for each of these:
        for gov in govs:
            # get dep-neighbours of each of these
            coord = [
                d['dep']['id'] for d in deps.tok_index[gov][1]
                if d['type'] in ('coord', 'conj')]
            # print 'coord:', [deps.tok_index[c][0]['lemma'] for c in coord]
            coord += [
                d['gov']['id'] for d in deps.tok_index[gov][2]
                if d['type'] in ('coord', 'conj')]
            # print 'coord:', [deps.tok_index[c][0]['lemma'] for c in coord]
            # and unify their relations
            # logging.info('unifying these:')
            # for c in coord:
            #     logging.info(u"{0}".format(
            #         deps.tok_index[c][0]['word']))
            gov_tok = deps.tok_index['gov'][0]
            if gov_tok is None or gov_tok['msd'][0] != 'C':
                # if the gov is not a conjunction, then it must take part
                # in the unification
                coord.append(gov)
            else:
                # otherwise it should be removed
                deps.remove_tok(gov)

            deps = self.unify_dependencies(
                coord, deps, exclude=set(['coord', 'punct']))

        # we reindex in the end only!
        deps.index()
        return deps

    def unify_dependencies(self, tokens, deps, exclude):
        for tok1 in tokens:
            for tok2 in tokens:
                if tok2 == tok1:
                    continue
                for dep in deps.tok_index[tok1][1]:
                    if dep['type'] in exclude:
                        continue
                    # logging.info('copying: {0}'.format(dep))
                    deps.add(dep['type'], deps.tok_index[tok2][0], dep['dep'])
                for dep in deps.tok_index[tok1][2]:
                    if dep['type'] in exclude:
                        continue
                    # logging.info('copying: {0}'.format(dep))
                    deps.add(dep['type'], dep['gov'], deps.tok_index[tok2][0])
        return deps

    def process_dependencies(self, deps):
        if self.lang == 'en':
            return self.process_stanford_dependencies(deps)
        elif self.lang == 'hu':
            return self.process_magyarlanc_dependencies(deps)
        else:
            raise Exception('unsupported language: {0}'.format(self.lang))

    def process_magyarlanc_dependencies(self, deps):
        deps = NewDependencies(deps)
        deps.remove_type('punct')
        deps.index()
        deps = self.process_conjunction_magyarlanc(deps)
        deps = self.process_copulars_magyarlanc(deps)
        deps = self.process_coordination_magyarlanc(deps)
        return deps.deps

    def process_stanford_dependencies(self, dep_strings):
        try:  # TODO
            deps = Dependencies.create_from_strings(dep_strings)
        except TypeError:
            deps = Dependencies(dep_strings)
        deps = self.process_copulars(deps)
        deps = self.remove_copulars(deps)
        deps = self.process_rcmods(deps)
        deps = self.process_negation(deps)
        # deps = self.process_coordinated_root(deps)
        deps = self.process_coordination_stanford(deps)

        return NewDependencies.create_from_old_deps(deps).deps
```

7.5 The `Lexicon` class

The `Lexicon` class stores `4lang` definitions for words, separating the manually written ones in the `4lang` dictionary from those built by the `dict_to_4lang` module. When invoked from the command line, `Lexicon.py` processes the `4lang` dictionary (using the `definition_parser` module of the `pymachine` library) and saves the resulting `Lexicon` instance in pickle format. `dict_to_4lang` loads the lexicon built from `4lang`, adds definitions built from dictionaries, and saves the output. All other applications can load any of the pickle files to use the corresponding `Lexicon` instance. Applications typically use the `get_machine` function to obtain the `4lang` definition graph for some word. By default, `get_machine` first searches for definitions of a word in `4lang`, then among words for which graphs have been built automatically, and finally falls back to creating a new `Machine` instance with no definition (i.e. no connections to other `Machines`). The `expand` function implements expansion of definitions (see Section 5.3), adding links to all nodes in a definition taken from their own definitions. Stopwords are omitted by default, the user can specify other words that are to be skipped. Expansion does not affect definition graphs stored in the lexicon.

```
import copy
import cPickle
import json
import logging
import sys

from nltk.corpus import stopwords as nltk_stopwords
from pymachine.definition_parser import read as read_defs
from pymachine.machine import Machine
from pymachine.control import ConceptControl
from pymachine.utils import MachineGraph, MachineTraverser

from utils import get_cfg

import networkx as nx
import csv


class Lexicon():
    """A mapping from lemmas to machines"""

    @staticmethod
    def build_from_4lang(cfg):
        fn = cfg.get("machine", "definitions")
        plural_fn = cfg.get("machine", "plurals")
        primitive_fn = cfg.get("machine", "primitives")
        primitives = set(
            [line.decode('utf-8').strip() for line in open(primitive_fn)])
        logging.info('parsing 4lang definitions...')
        pn_index = 1 if cfg.get("deps", "lang") == 'hu' else 0
        definitions = read_defs(
            file(fn), plural_fn, pn_index, three_parts=True)
        logging.info('parsed {0} entries, done!'.format(len(definitions)))
        lexicon = Lexicon.create_from_dict(definitions, primitives, cfg)
        return lexicon

    @staticmethod
    def load_from_binary(file_name):
        logging.info('loading lexicon from {0}...'.format(file_name))
        data = cPickle.load(file(file_name))
        machines_dump, ext_machines_dump = map(
            lambda s: json.loads(data[s]), ("def", "ext"))
        cfg, primitives = data['cfg'], data['prim']
        lexicon = Lexicon.create_from_dumps(machines_dump, ext_machines_dump,
                                            primitives, cfg)
        logging.info('done!')
        return lexicon

    def save_to_binary(self, file_name):
        logging.info('saving lexicon to {0}...'.format(file_name))
        data = {
            "def": json.dumps(Lexicon.dump_machines(self.lexicon)),
            "ext": json.dumps(Lexicon.dump_machines(self.ext_lexicon)),
            "prim": self.primitives,
            "cfg": self.cfg}

        with open(file_name, 'w') as out_file:
            cPickle.dump(data, out_file)
        logging.info('done!')

    @staticmethod
    def create_from_dumps(machines_dump, ext_machines_dump, primitives, cfg):
        """builds the lexicon from dumps created by Lexicon.dump_machines"""
        lexicon = Lexicon(cfg)
        lexicon.primitives = primitives
        for word, dumped_def_graph in machines_dump.iteritems():
            new_machine = Machine(word, ConceptControl())
            lexicon.add_def_graph(word, new_machine, dumped_def_graph)
            lexicon.add(word, new_machine, external=False)

        for word, dumped_def_graph in ext_machines_dump.iteritems():
            new_machine = Machine(word, ConceptControl())
            lexicon.add_def_graph(word, new_machine, dumped_def_graph)
            lexicon.add(word, new_machine, external=True)

        return lexicon

    def add_def_graph(self, word, word_machine, dumped_def_graph,
                      allow_new_base=False, allow_new_ext=False):
        node2machine = {}
        graph = MachineGraph.from_dict(dumped_def_graph)
        for node in graph.nodes_iter():
            pn = "_".join(node.split('_')[:-1])
            if pn == word:
                node2machine[node] = word_machine
            else:
                if not pn:
                    logging.warning(u"empty pn in node: {0}, word: {1}".format(
                        node, word))
                node2machine[node] = self.get_machine(pn, new_machine=True)

        for node1, adjacency in graph.adjacency_iter():
            machine1 = node2machine[node1]
            for node2, edges in adjacency.iteritems():
                machine2 = node2machine[node2]
                for i, attributes in edges.iteritems():
                    part_index = attributes['color']
                    machine1.append(machine2, part_index)

    @staticmethod
    def dump_definition_graph(machine, seen=set()):
        graph = MachineGraph.create_from_machines([machine])
        return graph.to_dict()

    @staticmethod
    def dump_machines(machines):
        """processes a map of lemmas to machines and dumps them to lists
        of strings, for serialization"""
        dump = {}
        for word, machine_set in machines.iteritems():
            if len(machine_set) > 1:
                raise Exception("cannot dump lexicon with ambiguous \
                    printname: '{0}'".format(word))
            machine = next(iter(machine_set))

            # logging.info('dumping this: {0}'.format(
            #     MachineGraph.create_from_machines([machine]).to_dot()))

            dump[word] = Lexicon.dump_definition_graph(machine)
        return dump

    @staticmethod
    def create_from_dict(word2machine, primitives, cfg):
        lexicon = Lexicon(cfg)
        lexicon.lexicon = dict(word2machine)
        lexicon.primitives = primitives
        return lexicon

    def __init__(self, cfg):
        self.cfg = cfg
        self.lexicon = {}
        self.ext_lexicon = {}
        self.oov_lexicon = {}
        self._known_words = None
        self.expanded = set()
        self.expanded_lexicon = {}
        self.stopwords = set(nltk_stopwords.words('english'))
        self.stopwords.add('as')  # TODO
        self.stopwords.add('root')  # TODO
        self.full_graph = None
        self.shortest_path_dict = None

    def get_words(self):
        return set(self.lexicon.keys()).union(set(self.ext_lexicon.keys()))

    def known_words(self):
        if self._known_words is None:
            self._known_words = self.get_words()
        return self._known_words

    def add(self, printname, machine, external=True, oov=False):
        if printname in self.oov_lexicon:
            assert oov is False
            del self.oov_lexicon[printname]
        lexicon = self.oov_lexicon if oov else (
            self.ext_lexicon if external else self.lexicon)

        self._add(printname, machine, lexicon)

    def _add(self, printname, machine, lexicon):
        if printname in lexicon:
            raise Exception("duplicate word in lexicon: '{0}'".format(lexicon))
        lexicon[printname] = set([machine])

    def get_expanded_definition(self, printname):
        machine = self.expanded_lexicon.get(printname)
        if machine is not None:
            return machine

        machine = copy.deepcopy(self.get_machine(printname))
        self.expand_definition(machine)
        self.expanded_lexicon[printname] = machine
        return machine
```

```
def get_machine(self, printname, new_machine=False, allow_new_base=False,
                allow_new_ext=False, allow_new_oov=True):
    """returns the lowest level (base < ext < oov) existing machine
    for the printname. If none exist, creates a new machine in the lowest
    level allowed by the allow_* flags. Will always create new machines
    for uppercase printnames"""

    # returns a new machine without adding it to any lexicon
    if new_machine:
        return Machine(printname, ConceptControl())

    # TODO
    if not printname:
        return self.get_machine("_empty_")

    if printname.isupper():
        return self.get_machine(printname, new_machine=True)

    machines = self.lexicon.get(
        printname, self.ext_lexicon.get(
            printname, self.oov_lexicon.get(printname, set())))
    if len(machines) == 0:
        # logging.info(
        #     u'creating new machine for unknown word: "{0}"'.format(
        #         printname))
        new_machine = Machine(printname, ConceptControl())
        if allow_new_base:
            self.add(printname, new_machine, external=False)
        elif allow_new_ext:
            self.add(printname, new_machine)
        elif allow_new_oov:
            self.add(printname, new_machine, oov=True)
        else:
            return None

        return self.get_machine(printname)
    else:
        if len(machines) > 1:
            debug_str = u'ambiguous printname: {0}, machines: {1}'.format(
                printname,
                [lex.get(printname, set([]))
                 for lex in (self.lexicon, self.ext_lexicon,
                             self.oov_lexicon)])
            raise Exception(debug_str)

        return next(iter(machines))

def expand_definition(self, machine, stopwords=[]):
    def_machines = dict(
        [(pn, m) for pn, m in [
            (m2.printname(), m2) for m2 in MachineTraverser.get_nodes(
                machine, names_only=False, keep_upper=True)]
         if pn != machine.printname()])
    self.expand(def_machines, stopwords=stopwords)
```

```
def expand(self, words_to_machines, stopwords=[], cached=False):
    if len(stopwords) == 0:
        stopwords = self.stopwords
    for lemma, machine in words_to_machines.iteritems():
        if (
                (not cached or lemma not in self.expanded) and
                lemma in self.known_words() and lemma not in stopwords):

            # deepcopy so that the version in the lexicon keeps its links
            definition = self.get_machine(lemma)
            copied_def = copy.deepcopy(definition)

            """
            for parent, i in list(definition.parents):
                copied_parent = copy.deepcopy(parent)
                for m in list(copied_parent.partitions[i]):
                    if m.printname() == lemma:
                        copied_parent.remove(m, i)
                        break
                else:
                    raise Exception()
                    # "can't find {0} in partition {1} of {2}: {3}".format(
                    # ))
                copied_parent.append(copied_def, i)
            """

            case_machines = [
                m for m in MachineTraverser.get_nodes(
                    copied_def, names_only=False, keep_upper=True)
                if m.printname().startswith('=')]

            machine.unify(copied_def, exclude_0_case=True)

            for cm in case_machines:
                if cm.printname() == "=AGT":
                    if machine.partitions[1]:
                        machine.partitions[1][0].unify(cm)
                if cm.printname() == "=PAT":
                    if machine.partitions[2]:
                        machine.partitions[2][0].unify(cm)

            self.expanded.add(lemma)

def get_full_graph(self):
    if not self.full_graph == None:
        return self.full_graph
    allwords = set()
    allwords.update(self.lexicon.keys(), self.ext_lexicon.keys(), self.oov_lexicon.
        keys())
    self.full_graph = nx.MultiDiGraph()

    # TODO: only for debugging
    until = 10
    for i, word in enumerate(allwords):
        # TODO: only for debugging
```

```
            # if word not in ['dumb', 'intelligent', 'stupid']:
            #     continue
            # if i > until:
            #     break

            machine = self.get_machine(word)
            MG = MachineGraph.create_from_machines([machine], str_graph=True)
            G = MG.G

            # TODO: to print out all graphs
            # try:
            #     fn = os.path.join('/home/eszter/projects/4lang/data/graphs/allwords', u
                "{0}.dot".format(word)).encode('utf-8')
            #     with open(fn, 'w') as dot_obj:
            #         dot_obj.write(MG.to_dot_str_graph().encode('utf-8'))
            # except:
            #     print "EXCEPTION: " + word

            # TODO: words to test have nodes
            # if 'other' in G.nodes() and 'car' in G.nodes():
            #     print word
            #
            # if word == 'merry-go-round' or word == 'Klaxon':
            #     print G.edges()

            self.full_graph.add_edges_from(G.edges(data=True))
            # TODO: needed??
            # self.full_graph.add_nodes_from(G.nodes())

            # TODO: only for debugging
            # MG.G = self.full_graph
            # fn = os.path.join('/home/eszter/projects/4lang/test/graphs/full_graph', u
                "{0}.dot".format(i)).encode('utf-8')
            # with open(fn, 'w') as dot_obj:
            #     dot_obj.write(MG.to_dot_str_graph().encode('utf-8'))

        return self.full_graph

    def get_shortest_path(self, word1, word2, file):
        if self.shortest_path_dict == None:
            self.shortest_path_dict = dict()
            with open(file, 'r') as f:
                reader = csv.reader(f, delimiter="\t")
                d = list(reader)
                for path in d:
                    key = path[0] + "_" + path[-1]
                    self.shortest_path_dict[key] = len(path)
        key = word1 + "_" + word2
        if key in self.shortest_path_dict.keys():
            return self.shortest_path_dict[key]
        else:
            return 0

if __name__ == "__main__":
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s : " +
        "%(module)s (%(lineno)s) - %(levelname)s - %(message)s")
    cfg_file = sys.argv[1] if len(sys.argv) > 1 else None
    cfg = get_cfg(cfg_file)
    lexicon = Lexicon.build_from_4lang(cfg)
    lexicon.save_to_binary(cfg.get("machine", "definitions_binary"))
```

7.6 The `Lemmatizer` class

The `Lemmatizer` combines various external tools in trying to map words to `4lang` concepts. For each word processed, the `lemmatize` function invokes the `hunmorph` morphological analyzer (using wrappers around `ocamorph` and `hundisambig` from the `hunmisc` library), as well as the Porter stemmer. `lemmatize` caches the results of each analysis step, storing for each word form it encounters the stem (according to the Porter stemmer), the list of possible morphological analyses (according to `ocamorph`) and the analysis chosen by `hundisambig`. In using all these to select the lemma to be returned, the `lemmatize` function supports several strategies for different applications.

If no flags are passed, `lemmatize` returns the output of `hundisambig`. The option `defined` can be used to pass the list of all lemmas from which `lemmatize` should try to return one (e.g. the list of all concepts defined) – if specified, `lemmatize` will return the word itself if it is defined, then try the lemma from *hundisambig*, and then go through all other lemmas proposed by `ocamorph`. If no match is found, the stemmed form is tried as a last resort before returning None. If the flag `stemmed_first` is set to True, `lemmatize` will run the above process on the stem first and only return to the original word form if no defined lemma is found. If `defined` is left unspecified and `stem_first` is set to true at the same time, `lemmatize` will act as a plain Porter stemmer, and a warning is issued. By default, `Lemmatizer` loads on startup a cache file of previously analyzed words. To save a new cache file (or overwrite an old one), the program using `Lemmatizer` must call its `write_cache` function.

```
import logging
import os
import sys

from nltk.corpus import stopwords as nltk_stopwords
from hunmisc.utils.huntool_wrapper import Hundisambig, Ocamorph, OcamorphAnalyzer,
    MorphAnalyzer  # nopep8
from stemming.porter2 import stem as porter_stem

from utils import get_cfg

class Lemmatizer():
    def __init__(self, cfg):
        self.cfg = cfg
        self.analyzer, self.morph_analyzer = self.get_analyzer()

        self.stopwords = set(nltk_stopwords.words('english'))
        self.stopwords.add('as')  # TODO
        self.stopwords.add('root')  # TODO

        self.read_cache()
        self.oov = set()

    def clear_cache(self):
        self.cache = {}
        self.oov = set()

    def _analyze(self, word):
        stem = porter_stem(word)
        lemma = list(self.analyzer.analyze(
            [[word]]))[0][0][1].split('||')[0].split('<')[0]

        cand_krs = self.morph_analyzer.analyze([[word]]).next().next()
        candidates = [cand.split('||')[0].split('<')[0] for cand in cand_krs]

        self.cache[word] = (stem, lemma, candidates)

    def _lemmatize_with_stopwords(self, word, uppercase):
        if word == 'have':
            return 'HAS'
        elif not uppercase:
            return word
        elif word in self.stopwords:
            return word.upper()
        else:
            return word

    def lemmatize(self, word, defined=None, stem_first=False, uppercase=False,
                  debug=False):
        # if 'defined' is provided, will refuse to return lemmas not in it

        # if the word is defined, we just return it
        if defined is not None and word in defined:
            return self._lemmatize_with_stopwords(word, uppercase)

        # if the word is not in our cache, we run all analyses
        if word not in self.cache:
            self._analyze(word)

        stem, lemma, candidates = self.cache[word]

        # if stem_first flag is on, we rerun lemmatize on the stem
        # and return the result unless it doesn't exist
        if stem_first:
            if defined is None:
                logging.warning("stem_first=True and defined=None, \
                                'lemmatize' is now a blind Porter stemmer")
            stemmed_lemma = self.lemmatize(
                stem, defined=defined, stem_first=False, uppercase=uppercase)
            if stemmed_lemma is not None:
                return self._lemmatize_with_stopwords(stemmed_lemma, uppercase)

        # we return the lemma unless it's not in defined
        if defined is None or lemma in defined:
            return self._lemmatize_with_stopwords(lemma, uppercase)

        # we go over the other candidates as a last resort
        for cand in candidates:
            if cand in defined:
                return self._lemmatize_with_stopwords(cand, uppercase)

        # last resort is the porter stem:
        if stem in defined:
            return self._lemmatize_with_stopwords(stem, uppercase)

        # if that doesn't work either, we return None
        return None

    def get_analyzer(self):
        hunmorph_path = self.cfg.get('lemmatizer', 'hunmorph_path')
        ocamorph_fn = os.path.join(hunmorph_path, "ocamorph")
        morphdb_model_fn = os.path.join(hunmorph_path, "morphdb_en.bin")
        hundisambig_fn = os.path.join(hunmorph_path, "hundisambig")
        hunpos_model_fn = os.path.join(hunmorph_path, "en_wsj.model")

        logging.warning('loading hunmorph using binaries in {0}'.format(hunmorph_path))
        for fn in (ocamorph_fn, morphdb_model_fn, hundisambig_fn,
                   hunpos_model_fn):
            if not os.path.exists(fn):
                raise Exception("can't find hunmorph resource: {0}".format(fn))

        ocamorph = Ocamorph(ocamorph_fn, morphdb_model_fn)
        ocamorph_analyzer = OcamorphAnalyzer(ocamorph)
        hundisambig = Hundisambig(hundisambig_fn, hunpos_model_fn)
        morph_analyzer = MorphAnalyzer(ocamorph, hundisambig)

        return morph_analyzer, ocamorph_analyzer

    def read_cache(self):
        self.clear_cache()
        cache_fn = self.cfg.get('lemmatizer', 'cache_file')
        if not os.path.exists(cache_fn):
            return
        logging.info('reading hunmorph cache...')
        with open(cache_fn) as f_obj:
            for line in f_obj:
                try:
                    fields = line.decode('utf-8').strip().split('\t')
                except (ValueError, UnicodeDecodeError), e:
                    raise Exception('error parsing line in tok2lemma file: \
                        {0}\n{1}'.format(e, line))
                word, stem, lemma = fields[:3]
                candidates = fields[3:]

                self.cache[word] = (stem, lemma, candidates)

        logging.info('done!')

    def write_cache(self):
        cache_fn = self.cfg.get('lemmatizer', 'cache_file')
        logging.info('writing hunmorph cache...')
        with open(cache_fn, 'w') as f_obj:
            for word, (stem, lemma, candidates) in self.cache.iteritems():
                f_obj.write(u"{0}\t{1}\t{2}\t{3}\n".format(
                    word, stem, lemma, "\t".join(candidates)).encode('utf-8'))

        logging.info('done!')

def main():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s : " +
        "%(module)s (%(lineno)s) - %(levelname)s - %(message)s")
    cfg_file = sys.argv[1] if len(sys.argv) > 1 else None
    cfg = get_cfg(cfg_file)
    lemmatizer = Lemmatizer(cfg)
    while True:
        word = raw_input('> ')
        print lemmatizer.lemmatize(word)

if __name__ == "__main__":
    main()
```

7.7 The `pymachine` library

Concept graphs built by `4lang` are encoded using the external library `pymachine` (`http://www.github.com/kornai/pymachine`), which implements Eilenberg machines via the `Machine` class. Currently `4lang` uses these objects simply as graph nodes, not as Eilenberg machines. `pymachine.utils` provides, among others, the `MachineGraph` class for building, manipulating, (de)serializing and visualizing graphs of `Machine`s. This class relies on the open-source library `networkx` as its backend for encoding directed graphs. The `pymachine.definition_parser` module provides a parser for the format used by the `4lang` dictionary, generation is currently not supported, i.e. graphs created with `4lang` cannot be saved in this format. `pymachine` also contains several modules that form the codebase of the system described in (Nemeskey et al., 2013), these are not used by the `4lang` library.

7.8 The similarity module

All systems for measuring word similarity or textual similarity, described in Chapter 6, rely on 4lang's similarity module to return similarity scores for pairs of English words. Main functions are exposed by the WordSimilarity class, which performs lemmatization, accesses Lexicons, and generates scores using one of several strategies, depending on the application at hand:

```
from collections import defaultdict
from ConfigParser import ConfigParser
import logging
import math
import sys

from gensim.models import Word2Vec
from nltk.corpus import stopwords as nltk_stopwords
from scipy.stats.stats import pearsonr

from pymachine.utils import average, harmonic_mean, jaccard, min_jaccard, MachineGraph,
    MachineTraverser, my_max  # nopep8
from pymachine.wrapper import Wrapper as MachineWrapper

from lemmatizer import Lemmatizer
from lexicon import Lexicon
from text_to_4lang import TextTo4lang
from utils import ensure_dir, get_cfg, print_text_graph, print_4lang_graph
from sim_feats import SimFeatures, MachineInfo

assert jaccard, min_jaccard  # silence pyflakes


class WordSimilarity():
    def __init__(self, cfg, cfg_section='word_sim'):
        self.batch = cfg.getboolean(cfg_section, 'batch')

        logging.warning("fourlangpath is {0}".format(
            cfg.get(cfg_section, 'fourlangpath')))
        self.cfg = cfg
        self.graph_dir = cfg.get(cfg_section, "graph_dir")
        ensure_dir(self.graph_dir)
        self.lemmatizer = Lemmatizer(cfg)
        self.lexicon_fn = self.cfg.get(cfg_section, "definitions_binary")
        self.lexicon = Lexicon.load_from_binary(self.lexicon_fn)
        self.defined_words = self.lexicon.get_words()
        self.word_sim_cache = {}
        self.lemma_sim_cache = {}
        self.links_nodes_cache = {}
        self.stopwords = set(nltk_stopwords.words('english'))
        self.sim_feats = SimFeatures(cfg, cfg_section, self.lexicon)
        self.expand = cfg.getboolean(cfg_section, "expand")
        logging.info("expand is {0}".format(self.expand))

    def log(self, string):
```

```
    if not self.batch:
        logging.info(string)

def sim_type_to_function(self, sim_type):
    return lambda w1, w2: self.word_similarities(w1, w2)[sim_type]

def machine_similarities(self, machine1, machine2, machine1_expand, machine2_expand):
    pn1, pn2 = machine1.printname(), machine2.printname()
    self.log(u'machine1: {0}, machine2: {1}'.format(pn1, pn2))

    links1, nodes1 = self.get_links_nodes(machine1)
    links2, nodes2 = self.get_links_nodes(machine2)
    links1_expand, nodes1_expand = self.get_links_nodes(machine1_expand)
    links2_expand, nodes2_expand = self.get_links_nodes(machine2_expand)

    self.log('links1: {0}, links2: {1}'.format(links1, links2))
    self.log('nodes1: {0}, nodes2: {1}'.format(nodes1, nodes2))
    self.log('links1_expand: {0}, links2_expand: {1}'.format(links1_expand,
        links2_expand))
    self.log('nodes1_expand: {0}, nodes2_expand: {1}'.format(nodes1_expand,
        nodes2_expand))

    sims = self.sim_feats.get_all_features(MachineInfo(machine1_expand, nodes1,
        nodes1_expand, links1, links1_expand),
                                           MachineInfo(machine2_expand, nodes2,
                                               nodes2_expand, links2, links2_expand))

    # TODO: we should use this way, but so far it didn't prove to be better
    # if sims['is_antonym'] == 1:
    #     sims['shortest_path'] = 0

    return sims

def lemma_similarities(self, lemma1, lemma2):
    if (lemma1, lemma2) in self.lemma_sim_cache:
        return self.lemma_sim_cache[(lemma1, lemma2)]

    if lemma1 == lemma2:
        lemma_sims = self.sim_feats.one_similarities()

    machine1, machine2 = map(
            self.lexicon.get_machine, (lemma1, lemma2))
    machine1_expand, machine2_expand = map(
            self.lexicon.get_expanded_definition, (lemma1, lemma2))

    if not self.batch:
        for w, m in ((lemma1, machine1), (lemma2, machine2)):
            print_4lang_graph(w, m, self.graph_dir)
        for w, m in ((lemma1, machine1_expand), (lemma2, machine2_expand)):
            print_4lang_graph(w, m, self.graph_dir + "_expand")

    lemma_sims = self.machine_similarities(machine1, machine2, machine1_expand,
        machine2_expand)

    self.lemma_sim_cache[(lemma1, lemma2)] = lemma_sims
```

```
        self.lemma_sim_cache[(lemma2, lemma1)] = lemma_sims
        return lemma_sims

    def word_similarities(self, word1, word2):
        if (word1, word2) in self.word_sim_cache:
            return self.word_sim_cache[(word1, word2)]
        lemma1, lemma2 = [self.lemmatizer.lemmatize(
            word, defined=self.defined_words, stem_first=True, uppercase=True)
            for word in (word1, word2)]
        # self.log(u'lemmas: {0}, {1}'.format(lemma1, lemma2))
        if lemma1 is None or lemma2 is None:
            if lemma1 is None:
                logging.debug("OOV: {0}".format(word1))
            if lemma2 is None:
                logging.debug("OOV: {0}".format(word2))

            word_sims = self.sim_feats.zero_similarities()
        else:
            word_sims = self.lemma_similarities(lemma1, lemma2)
        self.word_sim_cache[(word1, word2)] = word_sims
        self.word_sim_cache[(word2, word1)] = word_sims
        return word_sims

    def get_links_nodes(self, machine, use_cache=True):
        if use_cache and machine in self.links_nodes_cache:
            return self.links_nodes_cache[machine]
        self.seen_for_links = set()
        links, nodes = self._get_links_and_nodes(machine, depth=0)
        links, nodes = set(links), set(nodes)
        links.add(machine.printname())
        nodes.add(machine.printname())
        self.links_nodes_cache[machine] = (links, nodes)
        return links, nodes

    def _get_links_and_nodes(self, machine, depth, exclude_links=False):
        name = machine.printname()
        if name.isupper() or name == '=AGT':
            links, nodes = [], []
        elif exclude_links:
            links, nodes = [], [name]
        else:
            links, nodes = [name], [name]

        # logging.info("{0}{1},{2}".format(depth*"    ", links, nodes))
        is_negated = False
        is_before = False
        if machine in self.seen_for_links or depth > 5:
            return [], []
        self.seen_for_links.add(machine)
        for i, part in enumerate(machine.partitions):
            for hypernym in part:
                h_name = hypernym.printname()
                # logging.info("{0}h: {1}".format(depth*"    ", h_name))
                if h_name in ("lack", "not", "before"):
                    is_negated = True
                    continue

                c_links, c_nodes = self._get_links_and_nodes(
                    hypernym, depth=depth+1, exclude_links=i != 0)

                if not h_name.isupper():
                    links += c_links
                nodes += c_nodes

        if not exclude_links:
            links += self.get_binary_links(machine)
        if is_negated:
            add_lack = lambda link: "lack_{0}".format(link) if isinstance(link, unicode)
                else ("lack_{0}".format(link[0]), link[1])  # nopep8
            links = map(add_lack, links)
            nodes = map(add_lack, nodes)

        return links, nodes

    def get_binary_links(self, machine):
        for parent, partition in machine.parents:
            parent_pn = parent.printname()
            # if not parent_pn.isupper() or partition == 0:
            if partition == 0:
                # haven't seen it yet but possible
                continue
            elif partition == 1:
                links = set([(parent_pn, other.printname())
                             for other in parent.partitions[2]])
            elif partition == 2:
                links = set([(other.printname(), parent_pn)
                             for other in parent.partitions[1]])
            else:
                raise Exception(
                    'machine {0} has more than 3 partitions!'.format(machine))
            for link in links:
                yield link

    def contains(self, links, machine):
        pn = machine.printname()
        for link in links:
            if link == pn or (pn in link and isinstance(link, tuple)):
                self.log('link "{0}" is/contains name "{1}"'.format(link, pn))
                return True
        else:
            return False

class GraphSimilarity():
    @staticmethod
    def graph_similarity(graph1, graph2):
        return jaccard(graph1.edges, graph2.edges)

    @staticmethod
    def old_graph_similarity(graph1, graph2):
        sim1, ev1 = GraphSimilarity.supported_score(graph1, graph2)
        sim2, ev2 = GraphSimilarity.supported_score(graph2, graph1)
        if sim1 + sim2 > 0:
            pass
            # logging.info('evidence sets: {0}, {1}'.format(ev2, ev2))
        return harmonic_mean((sim1, sim2))

    @staticmethod
    def supported_score(graph, context_graph):
        edge_count = len(graph.edges)
        supported = graph.edges.intersection(context_graph.edges)
        return len(supported) / float(edge_count), supported

    @staticmethod
    def old_supported_score(graph, context_graph):
        zero_count, zero_supported, bin_count, bin_supported = 0, 0, 0, 0
        evidence = []
        binaries = defaultdict(set)
        # logging.info('context edges: {0}'.format(context_graph.edges))
        for edge in graph.edges:
            # logging.info('testing edge: {0}'.format(edge))
            if edge[2] == 0:
                zero_count += 1
                if edge in context_graph.edges:
                    # logging.info('supported 0-edge: {0}'.format(edge))
                    evidence.append(edge)
                    zero_supported += 1
            else:
                binaries[edge[0]].add(edge)

        for binary, edges in binaries.iteritems():
            bin_count += 1
            if all(edge in context_graph.edges for edge in edges):
                # logging.info('supported binary: {0}'.format(edges))
                evidence.append(edges)
                bin_supported += 1

        if zero_count + bin_count == 0:
            logging.warning("nothing to support: {0}".format(graph))
            return 0.0, []

        return (zero_supported + bin_supported) / float(
            zero_count + bin_count), evidence


class SimComparer():
    def __init__(self, cfg_file, batch=True):
        self.config_file = cfg_file
        self.config = ConfigParser()
        self.config.read(cfg_file)
        self.get_vec_sim()
        self.get_machine_sim(batch)

    def get_vec_sim(self):
        model_fn = self.config.get('vectors', 'model')
        model_type = self.config.get('vectors', 'model_type')
        logging.warning('Loading model: {0}'.format(model_fn))
        if model_type == 'word2vec':
            self.vec_model = Word2Vec.load_word2vec_format(model_fn,
                                                           binary=True)
        elif model_type == 'gensim':
            self.vec_model = Word2Vec.load(model_fn)
        else:
            raise Exception('Unknown LSA model format')
        logging.warning('Model loaded: {0}'.format(model_fn))

    def vec_sim(self, w1, w2):
        if w1 in self.vec_model and w2 in self.vec_model:
            return self.vec_model.similarity(w1, w2)
        return None

    def get_machine_sim(self, batch):
        wrapper = MachineWrapper(
            self.config_file, include_longman=True, batch=batch)
        self.sim_wrapper = WordSimilarity(wrapper)

    def sim(self, w1, w2):
        return self.sim_wrapper.word_similarity(w1, w2, -1, -1)

    def get_words(self):
        self.words = set((
            line.strip().decode("utf-8") for line in open(
                self.config.get('words', 'word_file'))))
        logging.warning('read {0} words'.format(len(self.words)))

    def get_machine_sims(self):
        sim_file = self.config.get('machine', 'sim_file')
        self.machine_sims = {}
        out = open(sim_file, 'w')
        count = 0
        for w1, w2 in self.sorted_word_pairs:
            if count % 100000 == 0:
                logging.warning("{0} pairs done".format(count))
            sim = self.sim(w1, w2)
            if sim is None:
                logging.warning(
                    u"sim is None for non-ooovs: {0} and {1}".format(w1, w2))
                logging.warning("treating as 0 to avoid problems")
                self.machine_sims[(w1, w2)] = 0
            else:
                self.machine_sims[(w1, w2)] = sim
            count += 1
            out.write(
                u"{0}_{1}\t{2}\n".format(w1, w2, sim).encode('utf-8'))
        out.close()

    def get_vec_sims(self):
        sim_file = self.config.get('vectors', 'sim_file')
        out = open(sim_file, 'w')
        self.vec_sims = {}
        for w1, w2 in self.sorted_word_pairs:
            vec_sim = self.vec_sim(w1, w2)
            self.vec_sims[(w1, w2)] = vec_sim
            out.write(
                u"{0}_{1}\t{2}\n".format(w1, w2, vec_sim).encode('utf-8'))
        out.close()

    def get_sims(self):
        self.get_words()
        self.non_oov = set(
            (word for word in self.words if word in self.vec_model))

        logging.warning(
            'kept {0} words after discarding those not in embedding'.format(
                len(self.non_oov)))

        logging.warning('lemmatizing words to determine machine-OOVs...')
        self.non_oov = set(
            (word for word in self.non_oov
                if self.sim_wrapper.lemmatizer.lemmatize(
                    word, defined=self.sim_wrapper.machine_wrapper.definitions,
                    stem_first=True, uppercase=True) is not None))

        logging.warning(
            'kept {0} words after discarding those not in machine sim'.format(
                len(self.non_oov)))

        self.sorted_word_pairs = set()
        for w1 in self.non_oov:
            for w2 in self.non_oov:
                if w1 != w2 and w1 == sorted([w1, w2])[0]:
                    self.sorted_word_pairs.add((w1, w2))

        self.get_machine_sims()
        self.get_vec_sims()

    def compare(self):
        sims = [self.machine_sims[pair] for pair in self.sorted_word_pairs]
        vec_sims = [self.vec_sims[pair] for pair in self.sorted_word_pairs]

        pearson = pearsonr(sims, vec_sims)
        print "compared {0} distance pairs.".format(len(sims))
        print "Pearson-correlation: {0}".format(pearson)


def main_compare(cfg):
    comparer = SimComparer(cfg)
    comparer.get_sims()
    comparer.compare()


def main_sen_sim(cfg):
    graph_dir = cfg.get("sim", "graph_dir")
    dep_dir = cfg.get("sim", "deps_dir")
    ensure_dir(graph_dir)
    ensure_dir(dep_dir)
    text_to_4lang = TextTo4lang(cfg)
    for i, line in enumerate(sys.stdin):
        preprocessed_line = line.decode('utf-8').strip().lower()
        sen1, sen2 = preprocessed_line.split('\t')
        machines1 = text_to_4lang.process(
            sen1, dep_dir=dep_dir, fn="{0}a".format(i))
        machines2 = text_to_4lang.process(
            sen2, dep_dir=dep_dir, fn="{0}b".format(i))

        print_text_graph(machines1, graph_dir, fn="{0}a".format(i))
        print_text_graph(machines2, graph_dir, fn="{0}b".format(i))

        graph1, graph2 = map(
            MachineGraph.create_from_machines,
            (machines1.values(), machines2.values()))
        print GraphSimilarity.graph_similarity(graph1, graph2)

    # text_to_4lang.dep_to_4lang.lemmatizer.write_cache()


def get_test_pairs(fn):
    pairs = {}
    for line in open(fn):
        w1, w2, sim_str = line.decode('utf-8').strip().split('\t')
        pairs[(w1, w2)] = float(sim_str) / 10
    return pairs


def main_word_test(cfg):
    from scipy.stats.stats import pearsonr
    word_sim = WordSimilarity(cfg)

    # TODO: only testing
    # machine = word_sim.lexicon.get_machine('merry-go-round')
    # links, nodes = word_sim.get_links_nodes(machine)

    test_pairs = get_test_pairs(cfg.get('sim', 'word_test_data'))
    sims, gold_sims = [], []
    for (w1, w2), gold_sim in test_pairs.iteritems():
        sim = word_sim.word_similarity(w1, w2, 'foo', 'foo')  # dummy POS-tags
        if sim is None:
            continue
        gold_sims.append(gold_sim)
        sims.append(sim)
        print "{0}\t{1}\t{2}\t{3}\t{4}".format(
            w1, w2, gold_sim, sim, math.fabs(sim-gold_sim))

    print "Pearson: {0}".format(pearsonr(gold_sims, sims))


def main():
    logging.basicConfig(
        level=logging.INFO,
        format="%(asctime)s : " +
        "%(module)s (%(lineno)s) - %(levelname)s - %(message)s")

    cfg_file = sys.argv[1] if len(sys.argv) > 1 else None
    cfg = get_cfg(cfg_file)
    sim_type = cfg.get('sim', 'similarity_type')
    if sim_type == 'sentence':
        main_sen_sim(cfg)
    elif sim_type == 'word':
        raise Exception("main function for word sim not implemented yet")
    elif sim_type == 'word_test':
        main_word_test(cfg)
    else:
        raise Exception('unknown similarity type: {0}'.format(sim_type))

if __name__ == '__main__':
    # import cProfile
    # cProfile.run('main()')
    main()
```

Feature generation based on `4lang` subgraphs takes place in the `SimFeats` module, which also implements some recent experimental features:

```
import logging
from pymachine.utils import MachineGraph, jaccard

import networkx as nx
import networkx.algorithms.isomorphism as iso
import itertools
import os.path

class SimFeatures:
    def __init__(self, cfg, section, lexicon):
        self.lexicon = lexicon
        self.batch = cfg.getboolean(section, 'batch')
        self.feats_to_get = cfg.get(section, 'sim_types').split('|')
        self.feats_dict = {
            'links_jaccard' : ['links_jaccard'],
            'entities_jaccard' : ['entities_jaccard'],
            'nodes_jaccard' : ['nodes_jaccard'],
            'links_contain' : ['links_contain'],
            'nodes_contain' : ['nodes_contain'],
            '0-connected' : ['0-connected'],
            'is_antonym' : ['is_antonym'],
            'subgraphs' : ['subgraph_3N'],
            'fullgraph' : ['shortest_path']
        }

        self.shortest_path_file_name = cfg.get(section, 'shortest_path_res')
        if not os.path.isfile(self.shortest_path_file_name) or cfg.getboolean(section, '
            calc_shortest_path'):
            self.calc_path = True
            shortest_path_dir = os.path.dirname(self.shortest_path_file_name)
            if not os.path.exists(shortest_path_dir):
                os.makedirs(shortest_path_dir)
            self.shortest_path_res = open(self.shortest_path_file_name, 'w')
        else:
            self.calc_path = False


        if 'fullgraph' in self.feats_to_get:
            self.full_graph = self.lexicon.get_full_graph()
            print "NODES count: {0}".format(len(self.full_graph.nodes()))
            print "EDGES count: {0}".format(len(self.full_graph.edges()))
            self.UG = self.full_graph.to_undirected()

    def get_all_features(self, graph1, graph2):
        all_feats = dict()
        for f in self.feats_to_get:
            all_feats.update(self.get_feature_class(f, graph1, graph2))
        return all_feats

    def get_feature_class(self, feature_name, graph1, graph2):
        if feature_name == 'links_jaccard':
            return self.links_jaccard(graph1.links_expand, graph2.links_expand)
        elif feature_name == 'entities_jaccard':
            return self.entitiess_jaccard(graph1.links_expand, graph2.links_expand)
        elif feature_name == 'nodes_jaccard':
            return self.nodes_jaccard(graph1.nodes_expand, graph2.nodes_expand)
        elif feature_name == 'links_contain':
            return self.links_contain(graph1.name, graph1.links_expand, graph2.name,
                graph2.links_expand)
        elif feature_name == 'nodes_contain':
            return self.nodes_contain(graph1.name, graph1.nodes_expand, graph2.name,
                graph2.nodes_expand)
        elif feature_name == '0-connected':
            return self.zero_connected(graph1.name, graph1.links, graph1.links_expand,
                                       graph2.name, graph2.links, graph2.links_expand)
        elif feature_name == 'is_antonym':
            return self.is_antonym(graph1.name, graph1.nodes_expand, graph2.name, graph2.
                nodes_expand)
        elif feature_name == 'subgraphs':
            return self.subgraphs(graph1.machine, graph2.machine)
        elif feature_name == 'fullgraph':
            return self.fullgraph(graph1.name, graph2.name)
        else:
            return { feature_name : 0 }

    def links_jaccard(self, links1, links2):
        return { "links_jaccard" : jaccard(links1, links2)}

    def entitiess_jaccard(self, links1, links2):
        entities1 = filter(lambda l: "@" in l, links1)
        entities2 = filter(lambda l: "@" in l, links2)
        return {'entities_jaccard' : jaccard(entities1, entities2)}

    def nodes_jaccard(self, nodes1, nodes2):
        return { "nodes_jaccard" : jaccard(nodes1, nodes2)}

    def links_contain(self, name1, links1, name2, links2):
        val = -1
        if (self.contains(links1, name2) or
                self.contains(links2, name1)):
            val = 1
        return { "links_contain" : val}

    def nodes_contain(self, name1, nodes1, name2, nodes2):
        val = -1
        if (self.contains(nodes1, name2) or
                self.contains(nodes2, name1)):
            val = 1
        return { "nodes_contain" : val}

    def zero_connected(self, name1, links1, links1_expand, name2, links2, links2_expand):
        val = -1
        if name1 in links2 or name2 in links1:
            val = 1
        ret = { "0-connected" : val }
        val2 = -1
        if val == -1:
            if name1 in links2_expand or name2 in links1_expand:
                val2 = 1
        return ret

    def is_antonym(self, name1, nodes1, name2, nodes2):
        is_antonym = -1
        if ("lack_" + name1 in nodes2 and name1 not in nodes2):
            is_antonym = 1
        elif("lack_" + name2 in nodes1 and name2 not in nodes1):
            is_antonym = 1
        return {"is_antonym" : is_antonym }

    def subgraphs(self, machine1, machine2):
        temp = SubGraphFeatures(machine1, machine2, 5)
        return temp.subgraph_dict

    def fullgraph(self, name1, name2):
        ####################
        # Only for calculating shortest path
        ####################
        if self.calc_path:
            length = 0
            if name1 not in self.UG.nodes() or name2 not in self.UG.nodes():
                return {"shortest_path" : length}
            if nx.has_path(self.UG, name1, name2):
                path = nx.shortest_path(self.UG, name1, name2)
                length = len(path)
                print "PATH: " + name1 + " " + name2
                print path
                print length
                self.shortest_path_res.write("\t".join(path))
                self.shortest_path_res.write("\n")
        else:
            length = self.lexicon.get_shortest_path(name1, name2, self.
                shortest_path_file_name)
        return {"shortest_path" : length}
    def contains(self, links, name):
        for link in links:
            if link == name or (name in link and isinstance(link, tuple)):
                self.log('link "{0}" is/contains name "{1}"'.format(link, name))
                return True
        else:
            return False

    def uniform_similarities(self, s):
        temp_dict = dict()
        for sim_type in self.feats_to_get:
            for feat_type in self.feats_dict[sim_type]:
                temp_dict[feat_type] = s
        return temp_dict

    def zero_similarities(self):
        return self.uniform_similarities(0.0)

    def one_similarities(self):
        return self.uniform_similarities(1.0)

    def log(self, string):
        if not self.batch:
            logging.info(string)

class MachineInfo():
    def __init__(self, machine, nodes, nodes_expand, links, links_expand):
        self.name = machine.printname()
        self.machine = machine
        self.nodes = nodes
        self.links = links
        self.nodes_expand = nodes_expand
        self.links_expand = links_expand

class SubGraphFeatures():
    def __init__(self, machine1, machine2, max_depth):
        G1 = MachineGraph.create_from_machines([machine1], max_depth=max_depth)
        G2 = MachineGraph.create_from_machines([machine2], max_depth=max_depth)
        name1 = machine1.printname()
        name2 = machine2.printname()

        self.subgraph_dict = dict()
        # self.subgraph_dict.update(self._get_subgraph_N(G1.G, G2.G, name1, name2))
        # self.subgraph_dict.update(self._get_subgraph_N_X_N(G1.G, G2.G, name1, name2))
        self.subgraph_dict.update(self._get_subgraph_3_nodes(G1.G, G2.G, name1, name2))

    # TODO: not useful
    def _get_subgraph_N(self, graph1, graph2, name1, name2):
        ret = 0
        subgraphs1 = self._get_subgraphs(graph1, name1, 1)
        subgraphs2 = self._get_subgraphs(graph2, name2, 1)

        for r in itertools.product(subgraphs1, subgraphs2):
            GM =  nx.algorithms.isomorphism.GraphMatcher(r[0], r[1],
                                                     node_match=iso.
                                                         categorical_node_match(['
                                                         str_name'], ['name']),
                                                     edge_match=iso.
                                                         numerical_edge_match(['color
                                                         '], [-1]))
        if GM.is_isomorphic():
            is_upper = False
            for n, d in r[0].nodes_iter(data=True):
                if d['str_name'].isupper():
                    is_upper = True
            if not is_upper:
                ret = 1
    return {'subgraph_N' : ret}

def _get_subgraph_N_X_N(self, graph1, graph2, name1, name2):
    ret = {
        'subgraph_N_0_N' : 0
    }
    # TODO: not worth counting all of them
    # ret = {
    #       'subgraph_N_0_N' : 0,
    #       'subgraph_N_1_N' : 0,
    #       'subgraph_N_2_N' : 0
    # }
    subgraphs1 = self._get_subgraphs(graph1, name1, 2)
    subgraphs2 = self._get_subgraphs(graph2, name2, 2)

    for r in itertools.product(subgraphs1, subgraphs2):
        GM =  nx.algorithms.isomorphism.GraphMatcher(r[0], r[1],
                                                     node_match=iso.
                                                         categorical_node_match(['
                                                         str_name'], ['name']),
                                                     edge_match=iso.
                                                         numerical_edge_match(['color
                                                         '], [-1]))
        if GM.is_isomorphic():
            for u, v, d in r[0].edges(data=True):
                if d['color'] == 0:
                    ret['subgraph_N_0_N'] += 1
                    # print u + " " + v + " 0"
                # TODO: appears to be unuseful
                # elif d['color'] == 1:
                #       ret['subgraph_N_1_N'] += 1
                #      # print u + " " + v + " 1"
                # elif d['color'] == 2:
                #       ret['subgraph_N_2_N'] += 1
                #      # print u + " " + v + " 2"
    return ret

# TODO: not useful
def _get_subgraph_3_nodes(self, graph1, graph2, name1, name2):
    ret = {
        'subgraph_3N' : 0
    }
```

```
        subgraphs1 = self._get_subgraphs(graph1, name1, 3)
        subgraphs2 = self._get_subgraphs(graph2, name2, 3)

        for r in itertools.product(subgraphs1, subgraphs2):
            GM =  nx.algorithms.isomorphism.GraphMatcher(r[0], r[1],
                                                         node_match=iso.
                                                             categorical_node_match(['
                                                             str_name'], ['name']),
                                                         edge_match=iso.
                                                             numerical_edge_match(['color
                                                             '], [-1]))
            if GM.is_isomorphic():
                ret['subgraph_3N'] += 1
        return ret

    def _get_subgraphs(self, graph, name, size=3):
        subgraphs = set()
        # print "\nSubgraphs START: " + name
        target = nx.complete_graph(size)
        for sub_nodes in itertools.combinations(graph.nodes(),len(target.nodes())):
            subg = graph.subgraph(sub_nodes)
            if nx.is_weakly_connected(subg):
                # print subg.edges()
                subgraphs.add(subg)
        # print "Subgraphs END \n"
        return subgraphs

def test():
    sf = SimFeatures()
    print sf.get_all_features()

if __name__ == "__main__":
    test()
```

7.9 Configuration

All 4lang modules can be configured using standard Python configuration files, command line parameters have been avoided nearly everywhere. All parameters left unspecified in the cfg file passed to a module will be set to the values specified in default.cfg. If no configuration file is passed, defaults are used everywhere, running simple tests for most modules on data in the test/input directory. Options are documented in default.cfg, see Appendix A.

Chapter 8

Outlook

This chapter outlines our future plans for using 4lang to solve some of the most challenging tasks in computational semantics. In Section 8.1 we mention some outstanding issues in the 4lang library which we plan to address in the near future. We shall then proceed to briefly discuss the tasks of measuring *sentence similarity* and *entailment* (Section 8.2), *question answering* (Section 8.3), and semantics-based *parsing* (Section 8.4), arguing that each of these should be approached via the single generic task of determining the *likelihood* of some 4lang representation based on models of context trained on other 4lang graphs relevant to the task at hand (the context). Our plans for such a generic component are outlined in Section 8.5. Finally, Section 8.6 will discuss ways to exploit existing sources of both linguistic and extra-linguistic knowledge in the 4lang system by converting them to 4lang constructions and graphs, respectively.

8.1 Outstanding issues

8.1.1 True homonyms

At present we do not treat multiple entries for the same word, e.g.

- club$_1$: an organization for people who share a particular interest or enjoy similar activities, or a group of people who meet together to do something they are interested in
- club$_2$: a long thin metal stick used in golf to hit the ball
- club$_3$: one of the four suits in a set of playing cards, which has the design of three round black leaves in a group together

In the future these will have to be accommodated by three separate 4lang concepts. We will still not require a separate word sense disambiguation process, we shall rely on the spreading activation process to select exactly one entry upon encountering the surface form *club*.

8.1.2 Alternate word forms, synonyms

When processing dictionaries with dict_to_4lang, we do not currently handle definitions that consist of a single synonym of the headword. Resulting graphs such as purchase $\xrightarrow{0}$ buy are adequate representations of meaning, since the 0-edge warrants inheritence of all links, but explicitly replacing such words with their synonyms may have its practical advantages. The Collins Dictionary also lists alternate forms of many headwords, these could also be added to the concept dictionary, e.g. realise could point to the graph built from the definition of *realize*. Sometimes dictionaries give identical definitions for (perfect) synonyms, e.g. Longman defines both *vomit* and *upchuck* as *to bring food or drink up from your stomach and out through your mouth because you are ill or drunk*. Such duplicates can be detected to add the edges vomit $\overset{0}{\underset{0}{\rightleftharpoons}}$ upchuck.

8.2 Sentence similarity and entailment

In Sections 6.1 and 6.2 we have introduced measures of semantic similarity between words based on their 4lang definitions which helped achieve state of the art performance on the tasks of measuring word similarity. Most top STS systems reduce the task of measuring textual similarity to that of word similarity, and lexical resources such as WordNet and surface features such as character-based similarity play an important role in most approaches. Our current systems are no exception. We believe that the task of directly quantifying the similarity of two meaning representations amounts to detecting entailment between parts of such representations. The nature of the similarity scale (e.g. what it means for two sentences to be 70% similar) is unclear, but it can be assumed that (i) if two sentences S_1 and S_2 are perfectly similar (i.e. mean exactly the same thing), then each of them must entail the other, and (ii) if S_1 and S_2 are similar *to some extent* then there must be exist some substructures of the meanings of S_1 and S_2 such that these substructures are perfectly similar, i.e. entail each other. The connection between STS and RTE tasks has recently been made by (Vo & Popescu, 2016), who present a corpus annotated for both semantic relatedness and entailment, measure correlation between the two sets of scores, and propose a joint architecture for simultaneously performing the two tasks.

The nature of these substructures is less obvious. A straightforward approach is to consider subgraphs, and assume that similarity of two representations is connected to the intersection of graphs (i.e. the intersection of the sets of edges over the intersection of the sets of nodes). For example, the sentences *John walks* and *John runs*, when interpreted in `4lang` and properly expanded, will map to graphs that share the subgraph $\texttt{John} \overset{0}{\underset{1}{\rightleftharpoons}} \texttt{move} \overset{1}{\leftarrow} \texttt{INSTRUMENT} \overset{2}{\rightarrow} \texttt{foot}$. Other common configurations between graphs can also warrant similarity, e.g. *John walks with a stick* and *John fights with a stick* both map to $\texttt{John} \overset{0}{\underset{1}{\rightleftharpoons}} \texttt{X} \overset{1}{\leftarrow} \texttt{INSTRUMENT} \overset{2}{\rightarrow} \texttt{stick}$ for some X. If our notion of similarity could refer to shared subgraphs only, no connection could be made between `John` and `stick` and these sentences could not be judged more similar to each other than to virtually any sentence about John or about a stick being an instrument. We are therefore inclined to include such common templates in determining the similarity of two `4lang` graphs – templates are essentially graphs with some unspecified nodes. The number of such templates matching a given graph grows exponentially with the number of nodes, but we can expect the relevant templates to be of limited size and a search for common templates in two graphs seems feasible[1].

If similarity can be defined in terms of common substructures of `4lang` graphs, a definition of entailment can follow that takes into account the substructures in one graph that are also present in the other. Simply put, *John walks* entails *John moves* because the representation of the latter, $\texttt{John} \overset{0}{\underset{1}{\rightleftharpoons}} \texttt{move}$, is contained in that of the former, but entailment does not hold the other way round, because many edges for *John walks* are left uncovered by *John moves*, e.g. those in $\texttt{move} \overset{1}{\leftarrow} \texttt{INSTRUMENT} \overset{2}{\rightarrow} \texttt{foot}$. Since this asymmetric relationship between graphs – the ratio of templates in one that are present in the other – is also of a gradual nature, it is more intuitive to think of it as the extent to which some utterance *supports* the other – the term *entailment* is typically used as a strictly binary concept. *John moves* may not entail *John walks*, it nevertheless *supports* it to a greater extent than e.g. *John sings*.

How similarity and support between `4lang` graphs should be measured exactly cannot be worked out without considerable experimenting (we are trying to approximate human judgment, as in the case of the STS task in Section 6.1), what we argued for here is that `4lang` representations are powerful and expressive enough that the semantic relatedness of utterances can be measured through them effectively.

[1] The `4lang` theory of representing meaning using networks of Eilenberg machines – of which our graphs are simplifications – will have the machines `walk` and `fight` inherit all properties of all machines to which they have pointers on their 0th partition; in other words they will end up with all properties of concepts that are accessible through a path of IS_A relationships, and will probably share at least some very generic properties such as `voluntary action`. The machine-equivalent of templates could then be networks of machines whose sets of properties do not necessarily contain all properties of any concept.

8.3 Question Answering

In the previous section we discussed the task of measuring the extent to which one utterance *supports* another – a relationship that differs from entailment in being gradual. A workable measure of support can take part in question answering: it can be used to rank candidates in order to determine answers that are more supported by a given context. There remains the task of finding candidates that are relevant answers to the question asked. The `text_to_4lang` pipeline offers no special treatment for questions. A wh-question such as *Who won the 2014 World Cup* are handled by all components in the same way as indicatives, creating e.g. the edges $\texttt{who} \xleftarrow{1} \texttt{win} \xrightarrow{2} \texttt{cup}$. Yes-no questions are simply not detected as such, *Did Germany win the 2014 World Cup* and *Germany won the 2014 World Cup* will map to the same `4lang` graph. In the future we plan to experiment with simple methods for finding candidates: e.g. searching for wh-questions allows us to identify the template $\texttt{X} \xleftarrow{1} \texttt{win} \xrightarrow{2} \texttt{cup(...)}$ and match it against graphs already in the context; we shall discuss how such a context might be modeled in Section 8.5.

8.4 Parsing in `4lang`

For the purposes of the `4lang` modules and applications presented in this thesis, we relegate syntactic analysis to dependency parsers. In Section 4.4.1 we have seen examples of errors introduced by the parsing component, and in sections on evaluation we observed that they are in fact the single greatest source of errors in most of our applications. Our long-term plans for the `4lang` library include an integrated module for semantics-assisted parsing. Since most of our plans are unimplemented (with the exception of some early experiments documented in (Nemeskey et al., 2013)), here we shall only provide a summary of our basic ideas.

Since generic parsing remains a challenging task in natural language processing, many NLP applications rely on the output of chunkers for high-accuracy syntactic information about a sentence. Chunkers typically identify the boundaries of phrases at the lowest level of the constituent structure, e.g. in the sentence *A 61-year old furniture salesman was pushed down the shaft of a freight elevator* they would identify the noun phrases *[A 61-year old furniture salesman], [the shaft],* and *[freight elevator].* Since chunking can be performed with high accuracy across languages ((Kudo & Matsumoto, 2001; Recski & Varga, 2010)), and some of our past experiments suggest that the internal syntactic structure of chunks can also be detected with high accuracy (Recski, 2014), our first goal for `4lang` is to detect phrase-internal semantic relations directly.

The aim of parsing with `4lang` is to make the process sensitive to (lexical) semantics. Currently the phrase *blue giraffe* would be mapped to the graph $\texttt{giraffe} \xrightarrow{0} \texttt{blue}$ on the basis of the dependency relation `amod(giraffe, blue)`, warranted by a particular fragment of the parse-tree, something along the lines of *[$_{NP}$ [$_A$ blue] [$_N$ giraffe]]*, which is again constructed with little or no regard to the semantics of `blue` or `giraffe`. The architecture we propose would still make use of the constituent structure of phrases, but it would create a connection between *blue giraffe* and $\texttt{giraffe} \xrightarrow{0} \texttt{blue}$ by means of a *construction* that pairs the rewrite rule $\texttt{NP} \rightarrow \texttt{A N}$ with the operation that adds the 0-edge between the concepts corresponding to the words *blue* and *giraffe*[2]. Since many dependency parsers, among them the Stanford Parser used by `dict_to_4lang`, derive their analyses from parse trees using template matching, it seems reasonable to assume that a direct mapping between syntactic patterns and `4lang` configurations can also be implemented straightforwardly. The task of ranking competing parse trees can then be supplemented by some module that ranks `4lang` representations by likelihood; what likelihood means and how such a module could be designed is discussed in Section 8.5. Thus, the problem of resolving ambiguities such as the issue of PP-attachment discussed in Section 4.4.1, e.g. to parse the sentence *He ate spaghetti with meatballs*, becomes no more difficult then predicting that $\texttt{eat} \xrightarrow{2} \texttt{meatball}$ is significantly more likely than $\texttt{eat} \xleftarrow{1} \texttt{INSTRUMENT} \xrightarrow{2} \texttt{meatballs}$. If we plan to make such predictions based on statistics over `4lang` representations seen previously, our approach can be seen as the semantic counterpart of *data-oriented parsing* (Bod, 2008), a theory that estimates the likelihood of syntactic parses based on the likelihood of its substructures, learned from structures in some training data.

8.5 Likelihood of `4lang` representations

We have proposed the notion of support, the extent to which parts of one utterance entail parts of another, in Section 8.2, and we have also indicated in Section 8.3 that we require a model of context that allows us to measure the extent to which the context supports some utterance. Finally, in Section 8.4, we argued that a method for ranking `4lang` (sub)graphs by the extent to which the context supports them could be used to improve the quality of syntactic parsing and thereby reduce errors in the entire `text_to_4lang` pipeline. We shall refer to this measure as the *likelihood* of some `4lang` graph (given some context); we conclude this chapter by presenting our ideas for the design of a future `4lang` module

[2] As mentioned in Section 3.1, the directed graphs used throughout this thesis are simplifications of our formalism; the constructions in `4lang` actually map surface patterns to operations over Eilenberg-machines, in this case one that places a pointer to a `blue` machine on the 0th partition of a `giraffe` machine

that models context and measures likelihood. Given a system capable of comparing the likelihoods of competing semantic representations, we will have a chance of successfully addressing more complex tasks in artificial intelligence, such as the Winograd-schema Challenge (Levesque et al., 2011).

8.5.1 A probabilistic approach

In Section 8.2 we introduced `4lang` *templates* – sets of concepts and paths of edges between them – as the structures shared by `4lang` graphs that are semantically related. Templates are more general structures than subgraphs, two graphs may share many templates over a set of nodes in spite of having only few shared edges; a previous example was the pair of sentences *John walks with a stick* and *John fights with a stick*, sharing the template $\texttt{John} \underset{1}{\overset{0}{\rightleftharpoons}} \texttt{X} \xleftarrow{1} \texttt{INSTRUMENT} \xrightarrow{2} \texttt{stick}$. Our initial approach is to think of the likelihood of some graph as some product of the likelihood of matching templates, given a model of the context. We believe that both the likelihood of templates in some context and the way they can be combined to obtain the likelihood of an utterance should be learned from the set of `4lang` graphs associated with the context. E.g. if we are to establish the likelihood of the utterance *Germany won the 2014 World Cup* and the context is a set of `4lang` graphs obtained by processing a set of newspaper articles on sports using `text_to_4lang`, our answer should be based on (i) the frequency of templates in the target `4lang` graph, as observed in the set of context graphs and (ii) our knowledge of how important each template is, e.g. based on their overall frequency in the context or among all occurrences over their sets of nodes[3].

In theory there is an enormous number of templates to consider over some graph (doubly exponential in the number of nodes), but the search space can be effectively reduced in a fashion similar to the way standard language modeling reduces the space of all possible word sequences to that of trigrams. If e.g. we consider templates of no more than 4 nodes, and we use expansion to reduce all graphs to some form of 'plain English' with a vocabulary no greater than 10^5 (in (Kornai et al., 2015) we have shown that an even greater reduction is possible, by iterative expansion `4lang` representations can be reduced to 129 primitives, possibly fewer), then the number of node sets will remain in the 10^{15} range, and while the total number of theoretically possible `4lang` graphs over 4 nodes is as high as $2^{6\binom{4}{2}} \approx 10^{12}$, we cannot expect to observe more than a fraction of them: the

[3] At this point we must note that likelihood is not (directly related to) truth; in fact none of our previous discussions leading up to this notion makes reference to truth. Neither do we suggest that calculating likelihood can take the place of *inference* – a context may entail or contradict an utterance regardless of how *likely* the latter is; our notion is rather motivated by the various applications discussed in this chapter.

present 4lang architecture in itself determines a much smaller variety.

Note that templates likely to occur in data are also mostly meaningful: e.g. templates over the graph for *Germany won the 2014 World Cup* are representations for states-of-affairs such as 'Germany won a 2014 something' (Germany $\xleftarrow{1}$ win $\xrightarrow{2}$ X $\xrightarrow{0}$ 2014), 'somebody won a world cup' (X $\xleftarrow{1}$ win $\xrightarrow{2}$ cup $\xrightarrow{0}$ world), or 'Germany did something to a world something' (Germany $\xleftarrow{1}$ X $\xrightarrow{2}$ Y $\xrightarrow{0}$ world) – our proposed parameters are the likelihoods of each of these states-of-affairs based on what we've learned from previous experience.

What we outlined here are merely directions for further investigation – the exact architecture, the method of learning (including reduction of the parameter space) need to be determined by experiments, as does the question of how far such an approach can scale across many domains, genres, and large amounts of data. Our purpose was once again to argue for the expressiveness of 4lang representations, and to indicate our plans for future research in computational semantics.

8.5.2 An inference-based approach

In Section 3.3 we have discussed the expected capabilities of an inferencing component in 4lang. The rate of success with which such a component can perform simple reasoning over 4lang graphs may also be an indication of the likelihood of some 4lang representation. Quillian's example presented in Section 2.2.1, the phrase *lawyer's client*, allows for a simplification of its initial 4lang representation (compare Figures 3.7 and 3.8). When dismbiguating between multiple representations of e.g. the same piece of raw text, the potential of a given 4lang subgraph for such simplifications may be a good indicator of its likelihood.

8.6 External sources

8.6.1 World knowledge

Even the most simple forms of reasoning will require some model of world knowledge, and 4lang representations are capable of representing facts taken from publicly available knowledge bases such as WikiData (successor to the widely used but discontinued Freebase (Bollacker et al., 2008)). Such datasets contain *triplets* of the form predicate(argument1, argument2) such as author(George_Orwell, 1984). author is defined in Longman as *someone who has written a book*, which dict_to_4lang uses to build the definition graph in Figure 8.1. If we are ready to make the assumption that the first and second arguments

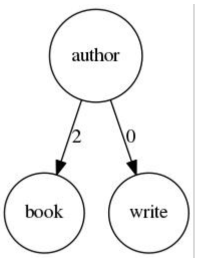


Figure 8.1: `4lang` definition of *author*

of the `WiktData` predicate `author` correspond to the 1- and 2-neighbours of the only binary relation in this definition (`write`), we can combine the fact `author(George_Orwell, 1984)` with the definition of `author` to obtain the graph in Figure 8.2.

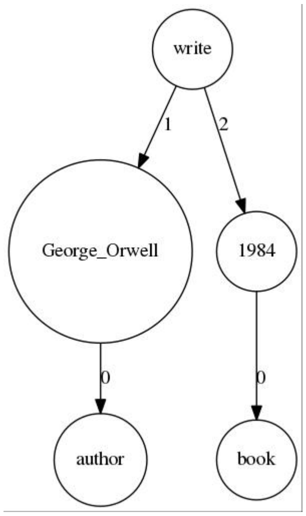


Figure 8.2: `4lang` graph inferred from `author(George_Orwell, 1984)`

A system for building `4lang` graphs from `WiktData` automatically will require a high-precision method for matching `WiktData` relations with arguments of `4lang` definitions, as we did in the case of `author` above. Simple heuristics like the one used in this example will have to be evaluated and only those with reaasonable precision selected. Such a curated set of patterns can then be applied to any subset of `WiktData` to convert large amounts of factual information to the `4lang` format and efficiently combine them with `4lang`'s knowledge of linguistic semantics.

8.6.2 Constructions

As discussed in Section 8.4, in the future we plan to map text to `4lang` representations using *constructions*, which are essentially pairs of patterns mapping classes of surface forms to classes of `4lang` graphs. Such constructions need not be hand-coded, they may be created on a large scale from existing linguistic ontologies. One example is the PropBank database (Palmer et al., 2005), mentioned in Section 2.2.5 and a key component of the AMR representation. PropBank contains argument lists of English verbs along with the semantic roles each argument takes. The example entry in Figure 8.3 establishes that the mandatory roles associated with arguments of the verb *agree* are those of *agreer* and *proposition* and that their *functions* are those of *prototypical agent* (`PAG`) and *prototypical patient* (`PPT`), respectively. This information could be represented as a `4lang` construction stating that concepts accessible from `agree` via 1- and 2-edges should have 0-edges leading to the concepts `agreer` and `proposition`. This construction could be used to extend the `4lang` definition of `agree` (see Figure 8.4). Once again, the large-scale extension of `4lang` data based on this external source will require a carefully selected set of high-precision patterns. A method must be devised to decide for each pair of `PropBank` frameset and `4lang` definition whether such an extension of the latter is warranted.

```
<frameset>
  <predicate lemma="agree">
    <roleset id="agree.01" name="agree">
      (...)
      <roles>
        <role descr="agreer" f="PAG" n="0">
          <vnrole vncls="36.1-1" vntheta="Agent"/>
        </role>
        <role descr="proposition" f="PPT" n="1">
          <vnrole vncls="36.1-1" vntheta="Theme"/>
        </role>
        <role descr="other entity agreeing" f="COM" n="2">
          <vnrole vncls="36.1-1" vntheta="co-agent"/>
        </role>
      </roles>
      (...)
    </roleset>
  </predicate>
</frameset>
```

Figure 8.3: Part of the PropBank frameset for *agree*[4]

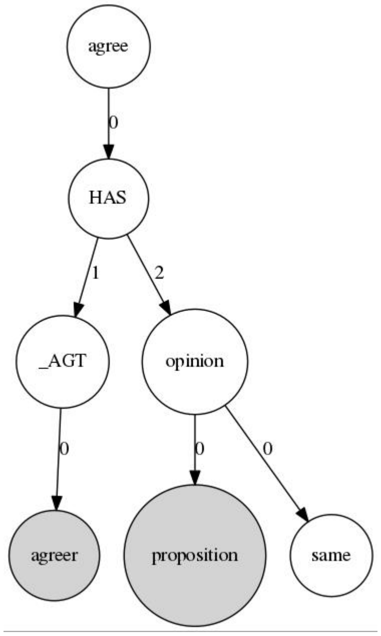


Figure 8.4: Extending the 4lang definition of *agree* (new nodes are shown in grey)

Appendices

Appendix A

Configuration file of the 4lang module

```
#When loading some cfg file in a 4lang module, unspecified parameters are
#assigned default values from this file
#Wherever possible, these values correspond to the most typical settings and
#test datasets distributed with 4lang

#Stanford Parser
[stanford]
#may in the future support using remote servers for parsing, leave it False for now
remote = False

#full path of Stanford Parser directory
dir = /home/recski/projects/stanford_dp/stanford-parser-full-2015-01-30/

#name of parser JAR file
parser = stanford-parser.jar

#name of model to load
model = englishRNN.ser.gz

#full path of jython executable
jython = /home/recski/projects/jython/jython

#Stanford CoreNLP
[corenlp]
#name of Java class to load
class_name = edu.stanford.nlp.pipeline.StanfordCoreNLP

#full path of Stanford CoreNLP directory
#CAUTION: when you change this path to point to your download, make sure it
#still ends with /*
```

```
classpath = /home/recski/projects/stanford_coreNLP/stanford-corenlp-full-2015-04-20/*

[magyarlanc]
path = magyarlanc/magyarlanc-2.0.jar

#miscellaneous data
[data]
#directory to save output of dependency parsing
deps_dir = test/deps
#directory for temporary files
tmp_dir = test/tmp

#dictionary data
[dict]
#input format
#possible values are: longman, collins, wiktionary, eksz, nszt
input_type = longman

#path to input file
input_file = test/input/longman_test.xml

#path to JSON file containing parsed dictionary entries
output_file = test/dict/longman_test.json

#text_to_4lang options
[text]
#path to input data
input_sens = test/input/mrhug_story.sens

#set to True to perform expansion on graphs built from text
expand = False

#set True to print dot files for each sentence
print_graphs = True

#path to save dot files
graph_dir = test/graphs/text

#if True, only dependency parsing will run and its output saved, but 4lang
#graphs won’t be built. Useful when working with large datasets.
parse_only = False

#path to save output of parsers
deps_dir = test/deps/text
```

```
#options to control which definitions are included by dict_to_4lang
[filter]

#include multiword expressions
keep_multiword = False

#include words with apostrophes
keep_apostrophes = False

#discard all but the first definition of each headword
first_only = True

[lemmatizer]
#full path of hunmorph binaries and models
hunmorph_path = /home/recski/sandbox/huntools_binaries

#path of cache (loaded but not updated by default, see docs)
cache_file = data/hunmorph_cache.txt

#options related to 4lang graphs
[machine]
#file containing 4lang dictionary
definitions = 4lang

#extra data for 4lang, currently not in use
plurals = 4lang.plural
primitives = 4lang.primitive

#pickle file to load 4lang graphs from
definitions_binary = data/machines/4lang.pickle

#pickle file to save 4lang graphs
definitions_binary_out = test/machines/wikt_test.pickle

#pickle file to save expanded 4lang graphs
expanded_definitions = test/machines/wikt_test_expanded.pickle

#path of directory for printing dot graphs
graph_dir = test/graphs/wikt_test

[deps]
#path to the map from dependencies to 4lang edges
dep_map = dep_to_4lang.txt
#language of the mapping (en or hu)
lang = en
```

```
#options for testing the word similarity module
[word_sim]
4langpath = /home/recski/sandbox/4lang
definitions_binary = %(4langpath)s/data/machines/longman_firsts.pickle
dep_map = %(4langpath)s/dep_to_4lang.txt
graph_dir = %(4langpath)s/data/graphs/sts
batch = true

#options for experimental sentence similarity system
[sim]
similarity_type = word_test
word_test_data = ws_data/wordsim_similarity_goldstandard.txt
graph_dir = test/graphs/sts_test
deps_dir = test/deps/sts_test

#options for experimental question answering system
[qa]
input_file = test/input/clef_qa_sample.xml
output_file = test/qa/clef_qa_sample.answers
graph_dir = test/graphs/qa_test
deps_dir = test/deps/qa_test
```

References

Ács, J., Pajkossy, K., & Kornai, A. (2013). Building basic vocabulary across 40 languages. In *Proceedings of the Sixth Workshop on Building and Using Comparable Corpora* (pp. 52–58). Sofia, Bulgaria: Association for Computational Linguistics.

Agirre, E., Banea, C., Cardie, C., Cer, D., Diab, M., Gonzalez-Agirre, A., ... Wiebe, J. (2015). SemEval-2015 Task 2: Semantic Textual Similarity, English, Spanish and Pilot on Interpretability. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015).* Denver, CO, U.S.A..

Agirre, E., Cer, D., Diab, M., & Gonzalez-Agirre, A. (2012). SemEval-2012 Task 6: A Pilot on Semantic Textual Similarity. In *First Joint Conference on Lexical and Computational Semantics (*SEM)* (pp. 385–393). Montréal, Canada: Association for Computational Linguistics.

Anderson, J. R. (1976). *Language, Memory, and Thought.* Lawrence Erlbaum Associates.

Anderson, J. R., & Bower, G. H. (1973). *Human associative memory.* Washington, DC: Winston.

Artzi, Y., Lee, K., & Zettlemoyer, L. (2015). Broad-coverage CCG semantic parsing with AMR. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing (EMNLP 2015)* (pp. 1699–1710). Lisbon, Portugal: Association for Computational Linguistics.

Banarescu, L., Bonial, C., Cai, S., Georgescu, M., Griffitt, K., Hermjakob, U., ... Schneider, N. (2013). Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse* (pp. 178–186). Sofia, Bulgaria: Association for Computational Linguistics.

Banjade, R., Maharjan, N., Niraula, N. B., Rus, V., & Gautam, D. (2015). Lemon and tea are not similar: Measuring word-to-word similarity by combining different methods. In A. Gelbukh (Ed.), *International Conference on Intelligent Text Processing and Computational Linguistics* (pp. 335–346). Springer.

Bittencourt, G. (1988). *A unified formalism for knowledge representation.* Laboratoire d'Informatique Fondamentale et d'Intelligence Artificielle.

Bobrow, R. J. (1979a). The RUS natural language parsing framework. *Research in Natural Language Understanding, annual report.*

Bobrow, R. J. (1979b). Semantic interpretation in PSI-KLONE. *Research in Natural Language Understanding, annual report.*

Bod, R. (2008). *The data-oriented parsing approach: theory and application.* Springer.

Boguraev, B. K., & Briscoe, E. J. (1989). *Computational Lexicography for Natural Language Processing.* Longman.

Bohnet, B. (2010). Top accuracy and fast dependency parsing is not a contradiction. In *Proceedings of the 23rd International Conference on Computational Linguistics (COLING 2010)* (pp. 89–97). Beijing, China: Coling 2010 Organizing Committee.

Bolinger, D. (1965). The atomization of meaning. *Language*, 555–573.

Bollacker, K., Evans, C., Paritosh, P., Sturge, T., & Taylor, J. (2008). Freebase: a collaboratively created graph database for structuring human knowledge. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (pp. 1247–1250).

Borgida, A., Brachman, R. J., McGuinness, D. L., & Resnick, L. A. (1989). Classic: A structural data model for objects. In *ACM SIGMOD record* (Vol. 18, pp. 58–67).

Brachman, R. J., Fikes, R. E., & Levesque, H. J. (1983). KRYPTON: A functional approach to knowledge representation. *IEEE Computer*, *10*, 67–73.

Brachman, R. J., & Schmolze, J. G. (1985). An overview of the KL-ONE knowledge representation system. *Cognitive science*, *9*(2), 171–216.

Bullon, S. (2003). *Longman dictionary of contemporary English 4.* Longman.

Chen, W.-T. (2015). Learning to map dependency parses to abstract meaning representations. In *Proceedings of the ACL-IJCNLP Student Research Workshop* (p. 41-46). Association for Computational Linguistics.

Collins, A., & Loftus, E. (1975). A spreading-activation theory of semantic processing. *Psychological Review*, *82*, 407–428.

Collobert, R., & Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th International Conference on Machine Learning* (pp. 160–167). New York, NY, USA: ACM.

DeMarneffe, M.-C., MacCartney, W., & Manning, C. (2006). Generating typed dependency parses from phrase structure parses. In *Proceedings of the 5th International Conference on Language Resources and Evaluation (LREC)* (Vol. 6, pp. 449–454). Genoa, Italy.

De Marneffe, M.-C., & Manning, C. D. (2008a). Stanford typed dependencies manual [Computer software manual]. Retrieved from http://nlp.stanford.edu/software/dependencies_manual.pdf (Revised for the Stanford Parser v. 3.5.1 in February 2015)

De Marneffe, M.-C., & Manning, C. D. (2008b). The Stanford typed dependencies representation. In *COLING 2008: Proceedings of the Workshop on Cross-Framework and Cross-Domain Parser Evaluation* (pp. 1–8). Association for Computational Linguistics.

Dice, L. R. (1945). Measures of the amount of ecologic association between species. *Ecology*, *26*(3), 297–302.

Dixon, R. M. (1994). *Ergativity*. Cambridge University Press.

Eilenberg, S. (1974). *Automata, languages, and machines* (Vol. A). Academic Press.

Fargues, J., Landau, M.-C., Dugourd, A., & Catach, L. (1986). Conceptual graphs for semantics and knowledge processing. *IBM Journal of Research and Development*, *30*(1), 70–79.

Fillmore, C. J. (1977). Scenes-and-frames semantics. In A. Zampolli (Ed.), *Linguistic structures processing* (pp. 55–88). North Holland.

Finkelstein, L., Gabrilovich, E., Matias, Y., Rivlin, E., Solan, Z., Wolfman, G., … Ruppin, E. (2002). Placing search in context: The concept revisited. *ACM Transactions on Information Systems*, *20(1)*, 116–131.

Flanigan, J., Thomson, S., Carbonell, J., Dyer, C., & Smith, N. A. (2014). A discriminative graph-based parser for the Abstract Meaning Representation. In *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics* (pp. 1426–1436). Baltimore, Maryland: Association for Computational Linguistics.

Foland Jr, W. R., & Martin, J. H. (2015). Dependency-based semantic role labeling using convolutional neural networks. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics (*SEM 2015)* (pp. 279–288). Association for Computational Linguistics.

Ganitkevitch, J., Van Durme, B., & Callison-Burch, C. (2013). PPDB: The Paraphrase Database. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT 2013)* (pp. 758–764). Atlanta, Georgia: Association for Computational Linguistics.

Garner, B. J., & Tsui, E. (1988). General purpose inference engine for canonical graph models. *Knowledge-Based Systems*, *1*(5), 266–278.

Grefenstette, E., & Sadrzadeh, M. (2015). Concrete models and empirical evaluations for the categorical compositional distributional model of meaning. *Computational*

Linguistics, *41*(1), 71–118.

Groenendijk, J., & Stokhof, M. (1991). Dynamic predicate logic. *Linguistics and philosophy*, *14*(1), 39–100.

Halácsy, P., Kornai, A., Németh, L., Rung, A., Szakadát, I., & Trón, V. (2004). Creating open language resources for Hungarian. In *Proceedings of the Fourth International Conference on Language Resources and Evaluation (LREC 2004)* (pp. 203–210). ELRA.

Han, L., L. Kashyap, A., Finin, T., Mayfield, J., & Weese, J. (2013). Umbc_ebiquity-core: Semantic textual similarity systems. In *Second Joint Conference on Lexical and Computational Semantics (*SEM)* (pp. 44–52). Atlanta, Georgia, USA: Association for Computational Linguistics.

Han, L., Martineau, J., Cheng, D., & Thomas, C. (2015). Samsung: Align-and-Differentiate Approach to Semantic Textual Similarity. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)* (pp. 172–177). Denver, Colorado: Association for Computational Linguistics.

Harris, Z. S. (1954). Distributional structure. *Word*, *10*, 146–162.

Hill, F., Reichart, R., & Korhonen, A. (2015). Simlex-999: Evaluating semantic models with (genuine) similarity estimation. *Computational Linguistics*, *41*(4), 665-695.

Hobbs, J. R. (1990). *Literature and cognition* (No. 21). Center for the Study of Language (CSLI).

Huang, E., Socher, R., Manning, C., & Ng, A. (2012). Improving word representations via global context and multiple word prototypes. In *Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics (ACL 2012)* (pp. 873–882). Jeju Island, Korea: Association for Computational Linguistics.

Ittzés, N. (Ed.). (2011). *A magyar nyelv nagyszótára III-IV.* Akadémiai Kiadó.

Jaccard, P. (1912). The distribution of the flora in the alpine zone. *New phytologist*, *11*(2), 37–50.

Kamp, H. (1981). A theory of truth and semantic representation. In J. Groenendijk, T. Jansen, & M. Stokhof (Eds.), *Formal methods in the study of language* (pp. 277–322). Amsterdam: Mathematisch Centrum.

Karpathy, A., Joulin, A., & Li, F. F. F. (2014). Deep fragment embeddings for bidirectional image sentence mapping. In Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, & K. Weinberger (Eds.), *Advances in neural information processing systems 27* (pp. 1889–1897). Curran Associates, Inc.

Kashyap, A., Han, L., Yus, R., Sleeman, J., Satyapanich, T., Gandhi, S., & Finin, T. (2014). Meerkat Mafia: Multilingual and Cross-Level Semantic Textual Similarity

Systems. In *Proceedings of the 8th International Workshop on Semantic Evaluation (SemEval 2014)* (pp. 416–423). Dublin, Ireland: Association for Computational Linguistics and Dublin City University.

Katz, J., & Fodor, J. A. (1963). The structure of a semantic theory. *Language*, *39*, 170–210.

Klein, D., & Manning, C. D. (2003). Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting of the Association for Computational Linguistics* (pp. 423–430). Sapporo, Japan: Association for Computational Linguistics.

Kornai, A. (2010). The algebra of lexical semantics. In C. Ebert, G. Jäger, & J. Michaelis (Eds.), *Proceedings of the 11th Mathematics of Language Workshop* (pp. 174–199). Springer.

Kornai, A. (2012). Eliminating ditransitives. In P. de Groote & M.-J. Nederhof (Eds.), *Revised and Selected Papers from the 15th and 16th Formal Grammar Conferences* (pp. 243–261). Springer.

Kornai, A. (in preparation). *Semantics.* `http://kornai.com/Drafts/sem.pdf`. Retrieved from `http://kornai.com/Drafts/sem.pdf`

Kornai, A., Ács, J., Makrai, M., Nemeskey, D. M., Pajkossy, K., & Recski, G. (2015). Competence in lexical semantics. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics (*SEM 2015)* (pp. 165–175). Denver, Colorado: Association for Computational Linguistics.

Kornai, A., & Makrai, M. (2013). A 4lang fogalmi szótár. In A. Tanács & V. Vincze (Eds.), *IX. Magyar Számitógépes Nyelvészeti Konferencia* (pp. 62–70).

Kudo, T., & Matsumoto, Y. (2001). Chunking with support vector machines. In *Proceedings of the 2nd meeting of the North American Chapter of the Association for Computational Linguistics (NAACL 2001)* (pp. 1–8). Association for Computational Linguistics.

Lee, H., Peirsman, Y., Chang, A., Chambers, N., Surdeanu, M., & Jurafsky, D. (2011). Stanford's multi-pass sieve coreference resolution system at the conll-2011 shared task. In *Proceedings of the fifteenth conference on computational natural language learning: Shared task* (pp. 28–34). Portland, Oregon, USA: Association for Computational Linguistics.

Lenat, D. B., & Guha, R. (1990). *Building large knowledge-based systems.* Addison-Wesley.

Levesque, H. J., Davis, E., & Morgenstern, L. (2011). The Winograd schema challenge. In *AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning* (Vol. 46, p. 47).

Liu, F., Flanigan, J., Thomson, S., Sadeh, N., & Smith, N. A. (2015). Toward abstractive summarization using semantic representations. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 1077–1086). Denver, Colorado: Association for Computational Linguistics.

May, J. (2016). SemEval-2016 Task 8: Meaning Representation Parsing. In *Proceedings of the 10th International Workshop on Semantic Evaluation (SemEval-2016)* (pp. 1063–1073). San Diego, California: Association for Computational Linguistics.

Miháltz, M. (2010). *Semantic resources and their applications in Hungarian natural language processing* (Doctoral dissertation, Pázmány Péter Catholic University). Retrieved from https://itk.ppke.hu/uploads/articles/163/file/Mihaltz_diss.pdf

Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. In Y. Bengio & Y. LeCun (Eds.), *Proceedings of the ICLR 2013.*

Mikolov, T., Yih, W.-t., & Zweig, G. (2013). Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT 2013)* (pp. 746–751). Atlanta, Georgia: Association for Computational Linguistics.

Miller, G. A. (1995). Wordnet: a lexical database for English. *Communications of the ACM*, *38*(11), 39–41.

Montague, R. (1970a). English as a formal language. In R. Thomason (Ed.), *Formal philosophy* (Vol. 1974, pp. 188–221). Yale University Press.

Montague, R. (1970b). Universal grammar. *Theoria*, *36*, 373–398.

Montague, R. (1973). The proper treatment of quantification in ordinary English. In R. Thomason (Ed.), *Formal philosophy* (pp. 247–270). Yale University Press.

Moser, M. (1983). An overview of NIKL, the new implementation of KL-ONE. *Research in Knowledge Representation and Natural Language Understanding*, 7–26.

Nemeskey, D., Recski, G., Makrai, M., Zséder, A., & Kornai, A. (2013). Spreading activation in language understanding. In *Proceedings of the 9th International Conference on Computer Science and Information Technologies (CSIT 2013)* (pp. 140–143). Yerevan, Armenia: Springer.

Nemeskey, D., Recski, G., & Zséder, A. (2012). Miből lesz a robot MÁV-pénztáros? [The makings of a robotic ticket-clerk]. In *Ix. magyar számítógépes nyelvészeti konferencia [ninth conference on hungarian computational linguistics].*

Nickel, M., Murphy, K., Tresp, V., & Gabrilovich, E. (2015). A review of relational machine learning for knowledge graphs: From multi-relational link prediction to automated knowledge graph construction. *arXiv preprint arXiv:1503.00759*.

nyest.hu. (2012). Miből lesz a robot-MÁV-pénztáros. *Nyelv és Tudomány*. Retrieved from http://www.nyest.hu/hirek/mobol-lesz-a-robbot-mav-penztaros

Palmer, M., Gildea, D., & Kingsbury, P. (2005). The Proposition Bank: An annotated corpus of semantic roles. *Computational linguistics*, *31*(1), 71–106.

Pan, X., Cassidy, T., Hermjakob, U., Ji, H., & Knight, K. (2015). Unsupervised entity linking with Abstract Meaning Representation. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 1130–1139). Denver, Colorado: Association for Computational Linguistics.

Peng, X., Song, L., & Gildea, D. (2015). A Synchronous Hyperedge Replacement Grammar based approach for AMR parsing. In *Proceedings of the 19th Conference on Computational Natural Language Learning (CoNLL 2015)* (pp. 32–41). Beijing, China: Association for Computational Linguistics.

Pennington, J., Socher, R., & Manning, C. (2014). Glove: Global vectors for word representation. In *Conference on Empirical Methods in Natural Language Processing (EMNLP 2014).*

Pust, M., Hermjakob, U., Knight, K., Marcu, D., & May, J. (2015). Parsing English into Abstract Meaning Representation using syntax-based machine translation. In *Proceedings of the 2015 conference on empirical methods in natural language processing (emnlp 2015)* (pp. 1143–1154). Lisbon, Portugal: Association for Computational Linguistics.

Pusztai, F. (Ed.). (2003). *Magyar értelmező kéziszótár*. Akadémiai Kiadó.

Quillian, M. R. (1968). Word concepts: A theory and simulation of some basic semantic capabilities. *Behavioral Science*, *12*, 410–430.

Quillian, M. R. (1969). The teachable language comprehender. *Communications of the ACM*, *12*, 459-476.

Recski, G. (2014). Hungarian noun phrase extraction using rule-based and hybrid methods. *Acta Cybernetica*, *21*, 461–479.

Recski, G. (2016). Building concept graphs from monolingual dictionary entries. In N. Calzolari et al. (Eds.), *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016).* Portorož, Slovenia: European Language Resources Association (ELRA).

Recski, G., & Ács, J. (2015). MathLingBudapest: Concept networks for semantic similarity. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)* (pp. 543–547). Denver, Colorado: Association for Computational Linguistics.

Recski, G., Borbély, G., & Bolevácz, A. (2016). Building definition graphs using monolingual dictionaries of Hungarian. In A. Tanács, V. Varga, & V. Vincze (Eds.), *XI. Magyar Számitógépes Nyelvészeti Konferencia [11th Hungarian Conference on Computational Linguistics].*

Recski, G., Iklódi, E., Pajkossy, K., & Kornai, A. (2016). Measuring semantic similarity of words using concept networks. In *Proceedings of the 1st Workshop on Representation Learning for NLP* (pp. 193–200). Berlin, Germany: Association for Computational Linguistics.

Recski, G., & Varga, D. (2010). A Hungarian NP Chunker. *The Odd Yearbook. ELTE SEAS Undergraduate Papers in Linguistics*, *8*, 87–93.

Richards, I. (1937). *The philosophy of rhetoric.* Oxford University Press.

Rumelhart, D. E., Lindsay, P. H., & Norman, D. A. (1972). *A process model for long-term memory.* Academic Press.

Schank, R. C. (1972). Conceptual dependency: A theory of natural language understanding. *Cognitive Psychology*, *3*(4), 552-631.

Schank, R. C., & Tesler, L. (1969). A conceptual dependency parser for natural language. In *Proceedings of the 1969 Conference on Computational Linguistics* (pp. 1–3). Association for Computational Linguistics.

Schwartz, R., Reichart, R., & Rappoport, A. (2015). Symmetric pattern based word embeddings for improved word similarity prediction. In *Proceedings of the 19th Conference on Computational Natural Language Learning (CoNLL 2015)* (pp. 258–267). Beijing, China: Association for Computational Linguistics.

Sinclair, J. M. (1987). *Looking up: an account of the COBUILD project in lexical computing.* Collins ELT.

Smith, R. (2015). Aristotle's logic. In E. N. Zalta (Ed.), *The Stanford encyclopedia of philosophy* (Summer 2015 ed.). http://plato.stanford.edu/archives/sum2015/entries/aristotle-logic/.

Smola, A., & Vapnik, V. (1997). Support vector regression machines. *Advances in neural information processing systems*, *9*, 155–161.

Smola, A. J., & Schölkopf, B. (2004). A tutorial on support vector regression. *Statistics and computing*, *14*(3), 199–222.

Socher, R., Bauer, J., Manning, C. D., & Andrew Y., N. (2013). Parsing with compositional vector grammars. In *Proceedings of the 51st Annual Meeting of the Association* *for Computational Linguistics (ACL 2013)* (pp. 455–465). Sofia, Bulgaria: Association for Computational Linguistics.

Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A., & Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing* (pp. 1631–1642). Seattle, Washington, USA: Association for Computational Linguistics.

Sondheimer, N. K., Weischedel, R. M., & Bobrow, R. J. (1984). Semantic interpretation using kl-one. In *Proceedings of the 10th International Conference on Computational Linguistics and 22nd Annual Meeting of the Association for Computational Linguistics* (pp. 101–107). Stanford, California, USA: Association for Computational Linguistics.

Sowa, J. F. (1984). *Conceptual structures: Information processing in mind and machine.* Addison-Wesley.

Sowa, J. F. (1992). Conceptual graphs as a universal knowledge representation. *Computers & Mathematics with Applications*, *23*(2), 75–93.

Sultan, M. A., Bethard, S., & Sumner, T. (2015). DLS@CU: Sentence similarity from word alignment and semantic vector composition. In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)* (pp. 148–153). Denver, Colorado: Association for Computational Linguistics.

Szedlák, A. (2012). Felsögödig kérek egy ilyen nyugdijas. *Origo Techbázis.* Retrieved from http://www.origo.hu/techbazis/20120928-felsogodig-kerek-egy-ilyen-nyugdijas-robot-mavpenztarost-epit-a-sztaki.html

Travis, C. (1997). Pragmatics. In B. Hale & C. Wright (Eds.), *A companion to the philosophy of language.* Oxford: Blackwell.

Tron, V., Gyepesi, G., Halácsky, P., Kornai, A., Németh, L., & Varga, D. (2005). Hunmorph: Open source word analysis. In *Proceedings of the ACL Workshop on Software* (pp. 77–85). Ann Arbor, Michigan: Association for Computational Linguistics.

Trubetzkoy, N. S. (1958). *Grundzüge der Phonologie.* Vandenhoeck & Ruprecht.

Turian, J., Ratinov, L.-A., & Bengio, Y. (2010). Word representations: A simple and general method for semi-supervised learning. In *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics* (pp. 384–394). Uppsala, Sweden: Association for Computational Linguistics.

Vanderwende, L., Menezes, A., & Quirk, C. (2015). An AMR parser for English, French, German, Spanish and Japanese and a new AMR-annotated corpus. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Com-*

putational Linguistics (NAACL 2015) (pp. 26–30). Denver, Colorado: Association for Computational Linguistics.

Vilain, M. B. (1985). The restricted language architecture of a hybrid representation system. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI-85)* (Vol. 85, pp. 547–551).

Vincze, V., Szauter, D., Almási, A., Móra, G., Alexin, Z., & Csirik, J. (2010). Hungarian dependency treebank. In N. Calzolari et al. (Eds.), *Proceedings of the Seventh conference on International Language Resources and Evaluation (LREC'10).* Valletta, Malta: European Language Resources Association (ELRA).

Vo, N. P. A., & Popescu, O. (2016). Corpora for learning the mutual relationship between semantic relatedness and textual entailment. In N. Calzolari et al. (Eds.), *Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016).* Paris, France: European Language Resources Association (ELRA).

Wang, C., Xue, N., & Pradhan, S. (2015). A transition-based algorithm for amr parsing. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT 2015)* (pp. 366–375). Denver, Colorado: Association for Computational Linguistics.

Wieting, J., Bansal, M., Gimpel, K., Livescu, K., & Roth, D. (2015). From paraphrase database to compositional paraphrase model and back. *TACL*, *3*, 345–358.

Wilks, Y. A. (1978). Making preferences more active. *Artificial Intelligence*, *11*, 197–223.

Xu, W., Callison-Burch, C., & Dolan, B. (2015). SemEval-2015 Task 1: Paraphrase and Semantic Similarity in Twitter (PIT). In *Proceedings of the 9th International Workshop on Semantic Evaluation (SemEval 2015)* (pp. 1–11). Denver, Colorado: Association for Computational Linguistics.

Zhu, X., Guo, H., & Sobhani, P. (2015). Neural networks for integrating compositional and non-compositional sentiment in sentiment composition. In *Proceedings of the Fourth Joint Conference on Lexical and Computational Semantics* (pp. 1–9). Denver, Colorado: Association for Computational Linguistics.

Zimmermann, T. E. (1999). Meaning postulates and the model-theoretic approach to natural language semantics. *Linguistics and Philosophy*, *22*, 529–561.

Zsibrita, J., Vincze, V., & Farkas, R. (2013). magyarlanc: A tool for morphological and dependency parsing of hungarian. In *Proceedings of the International Conference Recent Advances in Natural Language Processing (RANLP 2013)* (pp. 763–771). Hissar, Bulgaria: INCOMA Ltd. Shoumen.

Zweig, G., Platt, J. C., Meek, C., Burges, C. J., Yessenalina, A., & Liu, Q. (2012). Computational approaches to sentence completion. In *Proceedings of the 50th Annual*

Meeting of the Association for Computational Linguistics (pp. 601–610). Jeju Island, Korea: Association for Computational Linguistics.