ICARUS – An Extensible Graphical Search Tool for Dependency Treebanks

Markus Gärtner **Gregor Thiele** **Wolfgang Seeker** **Anders Björkelund** **Jonas Kuhn**
Institut für Maschinelle Sprachverarbeitung
University of Stuttgart
`firstname.lastname@ims.uni-stuttgart.de`

Abstract

We present ICARUS, a versatile graphical search tool to query dependency treebanks. Search results can be inspected both quantitatively and qualitatively by means of frequency lists, tables, or dependency graphs. ICARUS also ships with plugins that enable it to interface with tool chains running either locally or remotely.

1 Introduction

In this paper we present **ICARUS**[1] a search and visualization tool that primarily targets dependency syntax. The tool has been designed such that it requires minimal effort to get started with searching a treebank or system output of an automatic dependency parser, while still allowing for flexible queries. It enables the user to search dependency treebanks given a variety of constraints, including searching for particular subtrees. Emphasis has been placed on a functionality that makes it possible for the user to switch back and forth between a high-level, aggregated view of the search results and browsing of particular corpus instances, with an intuitive visualization of the way in which it matches the query. We believe this to be an important prerequisite for accessing annotated corpora, especially for non-expert users.

Search queries in ICARUS can be constructed either in a graphical or a text-based manner. Building queries graphically removes the overhead of learning a specialized query language and thus makes the tool more accessible for a wider audience. ICARUS provides a very intuitive way of breaking down the search results in terms of frequency statistics (such as the distribution of part-of-speech on one child of a particular verb against the lemma of another child). The dimensions for the frequency break-down are simply specified by using grouping operators in the query. The frequency tables are filled and updated in real time as the search proceeds through the corpus – allowing for a quick detection of misassumptions in the query.

ICARUS uses a plugin-based architecture that permits the user to write his own plugins and integrate them into the system. For example, it comes with a plugin that interfaces with an external parser that can be used to parse a sentence from within the user interface. The constraints for the query can then be copy-pasted from the resulting parse visualization. This facilitates example-based querying, which is particularly helpful for inexperienced users – they do not have to recall details of the annotation conventions outside of their focus of interests but can go by what the parser provides.[2]

ICARUS is written entirely in Java and runs out of the box without requiring any installation of the tool itself or additional libraries. This makes it platform independent and the only requirement is that a Java Runtime Environment (JRE) is installed on the host system. It is open-source and freely available for download.[3]

As parsers and other Natural Language Processing (NLP) tools are starting to find their way into other sciences such as (digital) humanities or social sciences, it gets increasingly important to provide intuitive visualization tools that integrate seamlessly with existing NLP tools and are easy to use also for non-linguists. ICARUS interfaces readily with NLP tools provided as web services by CLARIN-D,[4] the German incarnation of the European Infrastructure initiative CLARIN.

[1]Interactive platform for Corpus Analysis and Research tools, University of Stuttgart

[2]This is of course only practical with rather reliable automatic parsers, but in our experience, the state-of-the-art quality is sufficient.

[3]`www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/icarus.en.html`

[4]`http://de.clarin.eu`

The remainder of this paper is structured as follows: In Section 2 we elaborate on the motivation for the tool and discuss related work. Section 3 presents a running example of how to build queries and how results are visualized. In Section 4 we outline the details of the architecture. Section 5 discusses ongoing work, and Section 6 concludes.

2 Background

Linguistically annotated corpora are among the most important sources of knowledge for empirical linguistics as well as computational modeling of natural language. Moreover, for most users the only way to develop a systematic understanding of the phenomena in the annotations is through a process of continuous exploration, which requires suitable and intuitive tools.

As automatic analysis tools such as syntactic parsers have reached a high quality standard, exploration of large collections of auto-parsed corpus material becomes more and more common. Of course, the querying problem is the same no matter whether some target annotation was added manually, as in a treebank, or automatically. Yet, the strategy changes, as the user will try to make sure he catches systematic parsing errors and develops an understanding of how the results he is dealing with come about. While there is no guaranteed method for avoiding erroneous matches, we believe that an easy-to-use transparent querying mechanism that allows the user to look at the same or similar results from various angles is the best possible basis for an informed usage: frequency tables breaking down the corpus distributions in different dimensions are a good high-level hint, and the actual corpus instances should be only one or two mouse clicks away, presented with a concise visualization of the respective instantiation of the query constraints.

Syntactic annotations are quite difficult to query if one is interested in specific constructions that are not directly encoded in the annotation labels (which is the case for most interesting phenomena). Several tools have been developed to enable researchers to do this. However, many of these tools are designed for constituent trees only.

Dependency syntax has become popular as a framework for treebanking because it lends itself naturally to the representation of free word order phenomena and was thus adopted in the creation of treebanks for many languages that have less strict word order, such as the Prague Dependency Treebank for Czech (Hajič et al., 2000) or SynTagRus for Russian (Boguslavsky et al., 2000).

A simple tool for visualization of dependency trees is *What's wrong with my NLP?* (Riedel, 2008). Its querying functionality is however limited to simple string-searching on surface forms. A somewhat more advanced tool is *MaltEval* (Nilsson and Nivre, 2008), which offers a number of predefined search patterns ranging from part-of-speech tag to branching degree.

On the other hand, powerful tools such as *PML-TQ* (Pajas and Štěpánek, 2009) or *INESS* (Meurer, 2012) offer expressive query languages and can facilitate cross-layer queries (e.g., involving both syntactic and semantic structures). They also accommodate both constituent and dependency structures.

In terms of complexity in usage and expressivity, we believe ICARUS constitutes a middle way between highly expressive and very simple visualization tools. It is easy to use, requires no installation, while still having rich query and visualization capabilities. ICARUS is similar to PML-TQ in that it also allows the user to create queries graphically. It is also similar to the search tool *GrETEL* (Augustinus et al., 2012) as it interfaces with a parser, allowing the user to create queries starting from an automatic parse. Thus, queries can be created without any prior knowledge of the treebank annotation scheme.

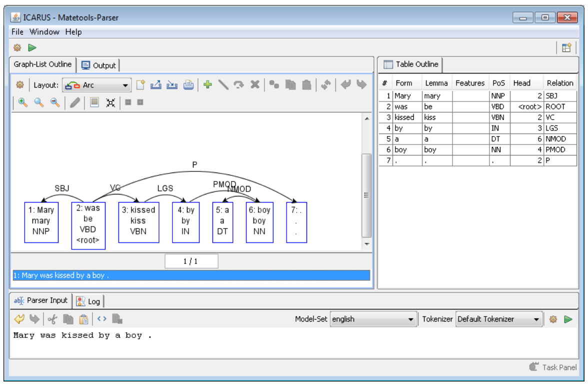
As for searching constituent treebanks, there is a plethora of existing search tools, such as *TGrep2* (Rohde, 2001), *TigerSearch* (Lezius, 2002), *MonaSearch* (Maryns, 2009), and *Fangorn* (Ghodke and Bird, 2012), among others. They implement different query languages with varying efficiency and expressiveness.

3 Introductory Example

Before going into the technical details, we show an example of what you can do with ICARUS. Assume that a user is interested in passive constructions in English, but does not know exactly how this is annotated in a treebank. As a first step, he can use a provided plugin that interfaces with a tool chain[5] to parse a sentence that contains a passive construction (thus adopting the example-based querying approach laid out in the introduc-

[5]using mate-tools by Bohnet (2010); available at `http://code.google.com/p/mate-tools`

tion). Figure 1 shows the parser interface. In the lower field, the user entered the sentence. The other two fields show the output of the parser, once as a graph and once as a feature value description.



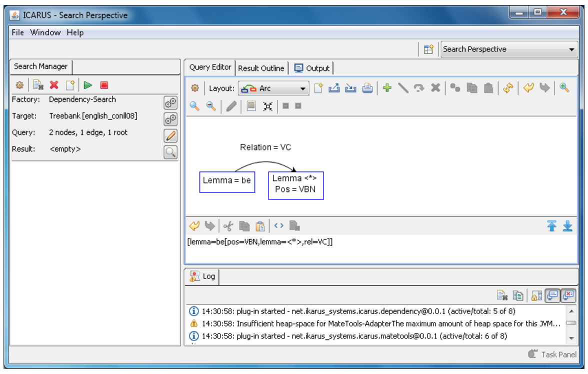
#	Form	Lemma	Features	PoS	Head	Relation
1	Mary	mary		NNP	2	SBJ
2	was	be		VBD	<root>	ROOT
3	kissed	kiss		VBN	2	VC
4	by	by		IN	3	LGS
5	a	a		DT	6	NMOD
6	boy	boy		NN	4	PMOD
7	.	.		.	2	P

1: Mary was kissed by a boy .

```
Mary was kissed by a boy .
```

Figure 1: Parsing the sentence *"Mary was kissed by a boy."* with a predefined tool chain.

In the second step, the user can then mark parts of the output graph by selecting some nodes and edges, and have ICARUS construct a query structure from it, following the drag-and-drop scheme users are familiar with from typical office software. The automatically built query can be manually adjusted by the user (relaxing constraints) and then be used to search for similar structures in a treebank. The parsing step can of course be skipped altogether, and a query can be constructed by hand right away. Figure 2 shows the query builder, where the user can define or edit search graphs graphically in the main window, or enter them as a query string in the lower window.



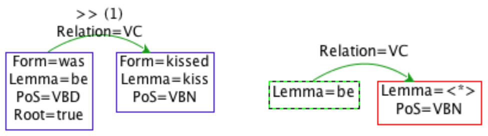
Factory: Dependency-Search
Target: Treebank [english_conll08]
Query: 2 nodes, 1 edge, 1 root
Result: <empty>

```
[lemma=be[pos=VBN,lemma=<*>,rel=VC]]
```

```
14:30:58: plug-in started - net.ikarus_systems.icarus.dependency@0.0.1 (active/total: 5 of 8)
14:30:58: Insufficient heap-space for MateTools-AdapterThe maximum amount of heap space for this JVM...
14:30:58: plug-in started - net.ikarus_systems.icarus.matetools@0.0.1 (active/total: 6 of 8)
```

Figure 2: Query builder for constructing queries.

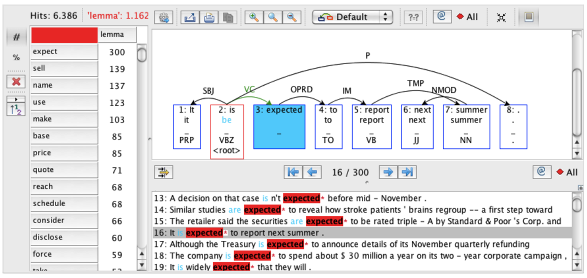
For the example, Figure 3 shows the query as it is automatically constructed by ICARUS from the partial parse tree (3a), and what it might look like after the user has changed it (3b). The modified query matches passive constructions in English, as annotated in the CoNLL 2008 Shared Task data set (Surdeanu et al., 2008), which we use here.



(a) automatically extracted (b) manually edited

Figure 3: Search graphs for finding passive constructions. (a) was constructed automatically from the parsed sentence, (b) is a more general version.

The search returns 6,386 matches. Note that the query (Figure 3b) contains a <*>-expression. This grouping operator groups the results according to the specified dimension, in this case by the lemma of the passivized verb. Figure 4 shows the result view. On the left, a list of lemmas is presented, sorted by frequency. Clicking on the lemma displays the list of matches containing that particular lemma on the right side. The matching sentences can then be browsed, with the active sentence also being shown as a tree. Note that the instantiation of the query constraints is highlighted in the tree display.

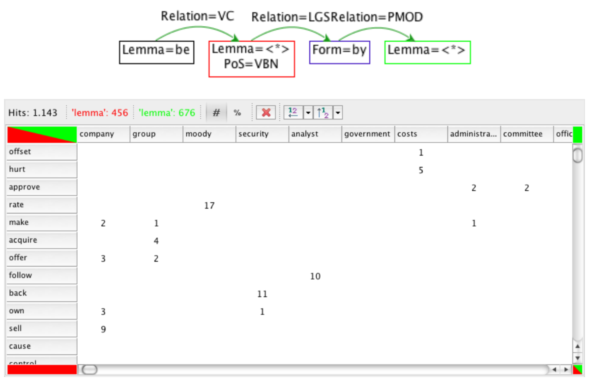


	lemma
expect	300
sell	139
name	137
use	123
make	103
base	85
price	85
quote	71
reach	68
schedule	68
consider	66
disclose	60
force	59

13: A decision on that case is n't expected* before mid – November .
14: Similar studies are expected* to reveal how stroke patients ' brains regroup -- a first step toward
15: The retailer said the securities are expected* to be rated triple – A by Standard & Poor 's Corp. and
16: It is expected* to report next summer .
17: Although the Treasury is expected* to announce details of its November quarterly refunding
18: The company is expected* to spend about $ 30 million a year on its two – year corporate campaign ,
19: It is widely expected* that they will .

Figure 4: Passive constructions in the treebank grouped by lemma and sorted by frequency.

The query could be further refined to restrict it to passives with an overt logical subject, using a more complex search graph for the *by*-phrase and a second instance of the grouping operator. The results will then also be grouped by the lemma of the logical subject, and are therefore presented as a two-dimensional table. Figure 5 shows the new query and the resulting view. The user is presented with a frequency table, where each cell contains the number of hits for this particular combination of verb lemma and logical subject. Clicking on the cell opens up a view similar to the right part of Figure 4 where the user can then again browse the actual trees.

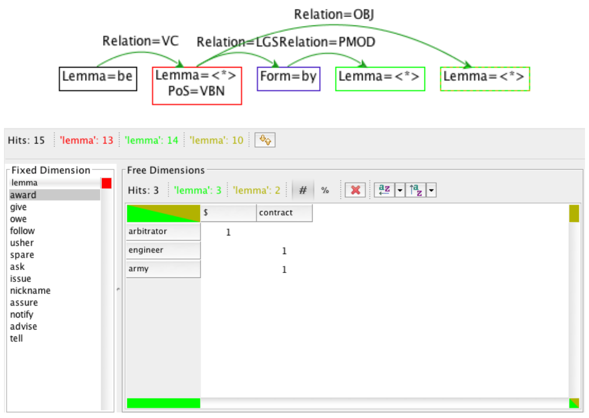


Hits: 1.143 'lemma': 456 'lemma': 676

	company	group	moody	security	analyst	government	costs	administra...	committee	offic
offset							1			
hurt							5			
approve								2	2	
rate			17							
make	2	1						1		
acquire		4								
offer	3	2								
follow					10					
back				11						
own	3			1						
sell	9									
cause										

Figure 5: Search graph and result view for passive constructions with overt logical subjects, grouped by lemma of the verb and the lemma of the logical subject.

Finally, we can add a third grouping operator. Figure 6 shows a further refined query for passives with an overt logical subject and an object. In the results, the user is presented with a list of values for the first grouping operator to the left. Clicking on one item in that list opens up a table on the right presenting the other two dimensions of the query.



Hits: 15 'lemma': 13 'lemma': 14 'lemma': 10

Fixed Dimension: lemma

award
give
owe
follow
usher
spare
ask
issue
nickname
assure
notify
advise
tell

Free Dimensions — Hits: 3 'lemma': 3 'lemma': 2

	$	contract
arbitrator	1	
engineer		1
army		1

Figure 6: Search graph and result view for passive constructions with an overt logical subject and an object, grouped by lemma of the verb, the logical subject, and the object.

This example demonstrates a typical use case for a user that is interested in certain linguistic constructions in his corpus. Creating the search graph and interpreting the results does not require any specialized knowledge other than familiarity with the annotation of the corpus being searched. It especially does not require any programming skills, and the possibility to graphically build a query obviates the need to learn a specialized query language.

4 Architecture

This section goes into more details about the inner workings of ICARUS. A main component is the search engine, which enables the user to quickly search treebanks for whatever he is interested in. A second important feature of ICARUS is the plugin-based architecture, which allows for the definition of custom extensions. Currently, ICARUS can read the commonly used CoNLL dependency formats, and it is easy to write extensions in order to add additional formats.

4.1 Search Engine and Query Builder

ICARUS has a tree-based search engine for treebanks, and includes a graphical query builder. Structure and appearance of search graphs are similar to the design used for displaying dependency trees (cf. Figure 1), which is realized with the open-source library JGraph.[6] Queries and/or their results can be saved to disk and later reloaded for further processing.

Defining a query graphically basically amounts to drawing a partial graph structure that defines the type of structure that the user is interested in. In practice, this is done by creating nodes in the query builder and connecting them by edges. The nodes correspond to words in the dependency trees of the treebank. Several features like word identity, lemma, part of speech, etc. can be specified for each node in the search graph in order to restrict the query. Dominance and precedence constraints over a set of nodes can be defined by simply linking nodes with the appropriate edge type. Edges can be further specified for relation type, distance, direction, projectivity, and transitivity. A simple example is shown in Figures 2 and 3. The search engine supports regular expressions for all string-properties (form, lemma, part of speech, relation). It also supports negation of (existence of) nodes and edges, and their properties.

As an alternative to the search graph, the user can also specify the query in a text-based format by constructing a comma separated collection of constraints in the form of *key=value* pairs for a single node contained within square brackets. Hierarchical structures are expressed by nesting their textual representation. Figure 7 shows the text-based form of the three queries used in the examples in Section 3.

[6]http://www.jgraph.com/

```
Query 1: [lemma=be[pos=VBN,lemma=<*>,rel=VC]]
Query 2: [lemma=be[pos=VBN,lemma=<*>,rel=VC[form=by,rel=LGS[lemma=<*>,rel=PMOD]]]]
Query 3: [lemma=be[pos=VBN,lemma=<*>,rel=VC[form=by,rel=LGS[lemma=<*>,rel=PMOD]]
                                          [lemma=<*>,rel=OBJ]]]
```

Figure 7: Text representation of the three queries used in the example in Section 3.

A central feature of the query language is the **grouping operator** (<*>), which will match any value and cause the search engine to group result entries by the actual instance of the property declared to be grouped. The results of the search will then be visualized as a list of instances together with their respective frequencies. Results can be sorted alphabetically or by frequency (absolute or relative counts) . Depending on the number of grouping operators used (up to a maximum of three) the result is structured as a list of frequencies (cf. Figure 4), a table of frequencies for pairs of instances (cf. Figure 5), or a list where each item then opens up a table of frequency results (cf. Figure 6). In the search graph and the result view, different colors are used to distinguish between different grouping operators.

The ICARUS search engine offers three different search modes:

Sentence-based. Sentence based search stops at the first successful hit in a sentence and returns every sentence on a list of results at most once.

Exhaustive sentence-based. The exhaustive sentence-based search mode extends the sentence based search by the possibility of processing multiple hits within a single sentence. Every sentence with at least one hit is returned exactly once. In the result view, the user can then browse the different hits found in one sentence.

Hit-based. Every successful hit is returned separately on the corresponding list of results.

When a query is issued, the search results are displayed on the fly as the search engine is processing the treebank. The sentences can be rendered in one of two ways: either as a tree, where nodes are arranged vertically by depth in the tree, or horizontally with all the nodes arranged side-by-side. If a tree does not fit on the screen, part of it is automatically collapsed but can be expanded again by the user.

4.2 Extensibility

ICARUS relies on the Java Plugin Framework,[7] which provides a powerful XML-based framework for defining plugins similarly to the engine used by the popular Eclipse IDE project. The **plugin-based architecture** makes it possible for anybody to write extensions to ICARUS that are specialized for a particular task. The parser integration of mate-tools demonstrated in Section 3 is an example for such an extension.

The plugin system facilitates custom extensions that make it possible to intercept certain stages of an ongoing search process and interact with it. This makes it possible for external tools to preprocess search data and apply additional annotations and/or filtering, or even make use of existing indices by using search constraints to limit the amount of data passed to the search engine. With this general setup, it is for example possible to easily extend ICARUS to work with constituent trees.

ICARUS comes with a dedicated plugin that enables access to web services provided by CLARIN-D. The project aims to provide tools and services for language-centered research in the humanities and social sciences. In contrast to the integration of, e.g., mate-tools, where the tool chain is executed locally, the user can define a tool chain by chaining several web services (e.g., lemmatizers, part-of-speech taggers etc.) together and apply them to his own data. To do this, ICARUS is able to read and write the TCF exchange format (Heid et al., 2010) that is used by CLARIN-D web services. The output can then be inspected and searched using ICARUS. As new NLP tools are added as CLARIN-D web services they can be immediately employed by ICARUS.

5 Upcoming Extensions

An upcoming release includes the following extensions:

- Currently, treebanks are assumed to fit into the executing computer's main memory. The new implementation will support asynchronous loading of data, with notifications passed to the query engine or a plugin when required data is available. Treebanks with millions of entries can then be loaded in less

[7] http://jpf.sourceforge.net/

memory consuming chunks, thus keeping the system responsive when access is requested.

- The search engine is being extended with an operator that allows disjunctions of queries. This will enable the user to aggregate frequency output over multiple queries.

6 Conclusion

We have presented ICARUS, a versatile and user-friendly search and visualization tool for dependency trees. It is aimed not only at (computational) linguists, but also at people from other disciplines, e.g., the humanities or social sciences, who work with language data. It lets the user create queries graphically and returns results (1) quantitatively by means of frequency lists and tables as well as (2) qualitatively by connecting the statistics to the matching sentences and allowing the user to browse them graphically. Its plugin-based architecture enables it to interface for example with external processing pipelines, which lets the user apply processing tools directly from the user interface.

In the future, specialized plugins are planned to work with different linguistic annotations, e.g. cross-sentence annotations as used to annotate coreference chains. Additionally, a plugin is intended that interfaces the search engine with a database.

Acknowledgments

This work was funded by the Deutsche Forschungsgemeinschaft (DFG) via the SFB 732 "Incremental Specification in Context", project D8, and by the Bundesministerium für Bildung und Forschung (BMBF) via project No. 01UG1120F, CLARIN-D center Stuttgart. The authors are also indebted to André Blessing and Heike Zinsmeister for reading an earlier draft of this paper.

References

Liesbeth Augustinus, Vincent Vandeghinste, and Frank Van Eynde. 2012. Example-based Treebank Querying. In *Proceedings of the Eight International Conference on Language Resources and Evaluation (LREC'12)*, Istanbul, Turkey. ELRA.

Igor Boguslavsky, Svetlana Grigorieva, Nikolai Grigoriev, Leonid Kreidlin, and Nadezhda Frid. 2000. Dependency Treebank for Russian: Concept, Tools, Types of Information. In *COLING 2000*, pages 987–991, Saarbrücken, Germany.

Bernd Bohnet. 2010. Top Accuracy and Fast Dependency Parsing is not a Contradiction. In *COLING 2010*, pages 89–97, Beijing, China.

Sumukh Ghodke and Steven Bird. 2012. Fangorn: A System for Querying very large Treebanks. In *COLING 2012: Demonstration Papers*, pages 175–182, Mumbai, India.

Jan Hajič, Alena Böhmová, Eva Hajičová, and Barbora Vidová-Hladká. 2000. The Prague Dependency Treebank: A Three-Level Annotation Scenario. In *Treebanks: Building and Using Parsed Corpora*, pages 103–127. Amsterdam:Kluwer.

Ulrich Heid, Helmut Schmid, Kerstin Eckart, and Erhard Hinrichs. 2010. A Corpus Representation Format for Linguistic Web Services: The D-SPIN Text Corpus Format and its Relationship with ISO Standards. In *Proceedings of the Seventh International Conference on Language Resources and Evaluation (LREC'10)*, Valletta, Malta. ELRA.

Wolfgang Lezius. 2002. *Ein Suchwerkzeug für syntaktisch annotierte Textkorpora*. Ph.D. thesis, IMS, University of Stuttgart.

Hendrik Maryns. 2009. MonaSearch – A Tool for Querying Linguistic Treebanks. In *Proceedings of TLT 2009*, Groningen.

Paul Meurer. 2012. INESS-Search: A Search System for LFG (and Other) Treebanks. In Miriam Butt and Tracy Holloway King, editors, *Proceedings of the LFG2012 Conference*. CSLI Publications.

Jens Nilsson and Joakim Nivre. 2008. MaltEval: an Evaluation and Visualization Tool for Dependency Parsing. In *Proceedings of the Sixth International Conference on Language Resources and Evaluation (LREC'08)*, Marrakech, Morocco. ELRA.

Petr Pajas and Jan Štěpánek. 2009. System for Querying Syntactically Annotated Corpora. In *Proceedings of the ACL-IJCNLP 2009 Software Demonstrations*, pages 33–36, Suntec, Singapore. Association for Computational Linguistics.

Sebastian Riedel. 2008. What's Wrong With My NLP? `http://code.google.com/p/whatswrong/`.

Douglas L.T. Rohde. 2001. TGrep2 the next-generation search engine for parse trees. `http://tedlab.mit.edu/~dr/Tgrep2/`.

Mihai Surdeanu, Richard Johansson, Adam Meyers, Lluís Màrquez, and Joakim Nivre. 2008. The CoNLL 2008 Shared Task on Joint Parsing of Syntactic and Semantic Dependencies. In *CoNLL 2008*, pages 159–177, Manchester, England.