Rule-based Logical Forms Extraction

Cenny Wenner
Department of Computer Science, Faculty of Science
Lund University, Sweden
cwenner@gmail.com
http://cennywenner.com

Abstract

In this paper, we present concise but robust rules for dependency-based logical form identification with high accuracy. We describe our approach from an intuitive and formalized perspective, which we believe overcomes much of the complexity. In comparison to previous work, we believe ours is more compact and involves less rules and exceptions. We also provide the reader with a comparison of the respective impacts of the most essential rules on the logical form identification task of the 2004 Senseval 3 test set.

1 Introduction

The logical form of a text segment is a first-order logic (FOL) well-formed formula (wff) representing the meaning of the segment. Such a representation is used for a number of problems involving semantics and inference, for instance question answering (QA) (Moldovan et al., 2003) and textual entailment (TE) (Bar-Haim et al., 2006; Tatu et al., 2006). Moldovan and Rus (2001) introduced a simplified representation where the wff is restricted to a conjunction of predicates and where functions/functors are not allowed as arguments. The arguments are partitioned into two sets: events and objects. Conventionally, objects are called entities and we extend that term to refer to both events and objects. With this representation, more complicated constructions of natural language are ignored but these are not crucial to the previously mentioned problems.

As an example, the representation of *John and the dog ran to his car* is

$$\text{John:\_n}(x_1), \text{and}(x_2, x_1, x_3), \text{dog:\_n}(x_3), \text{run:\_v}(e_1, x_2; x_4), \text{to}(e_1, x_4), \text{his}(x_4), \text{car:\_n}(x_4), \quad (1)$$

where the arguments of the form x_i and e_i denotes objects and events respectively. In this particular example, x_1 is John, x_3 the dog, x_2 John and the dog as a group, x_4 his car, and e_1 the event that they are running. The *to* predicate expresses a relation between the event *run* and the car, while the predicate *his* expresses a quality of the car alone. Note that John is not present in the *his* predicate, this is a task for a coreference solver. The suffixes *:_n* and *:_v* denote nouns and verbs respectively. For verbs, complements are listed after a semicolon.

A task for identifying these logical forms was introduced at the 2004 Senseval 3 conference. 27 teams were registered, although only five submitted sensible results, whereof one involved manual parsing (Rus, 2004). Our work builds on the results of these systems and we use the conference's annotated and unannotated data sets to evaluate and verify our approach.

The goal of this article is to present the reader with a robust rule-based scheme which only relies on a few exceptions to the simple default search rules. The transformation takes as input a dependency graph, a sequence of tokens, the tokens' parts of speech (POS), and morphological base forms (lemmatised word form). We briefly describe how to produce the input from raw English text in Sect. 3. The transformation is done in two steps. The first step constructs predicates with argument placeholders, called slots, and we cover this in Sect. 4. Sect. 5 deals with the second step where we substitute the placeholders with real arguments. The later step is more complicated and we devote several subsections to it. We evaluate the system on the Senseval 3 test set in Sect. 6 and concludes the paper with a discussion of these results and potential further work.

Joakim Nivre, Heiki-Jaan Kaalep, Kadri Muischnek and Mare Koit (Eds.)
NODALIDA 2007 Conference Proceedings, pp. 402–409

2 Previous Work

All contemporary systems for logical form identification rely on a syntactical parser. Constituent trees appear to be the most frequent approach today, introduced by Rus (2001). Two rivals to this approach are depedency graphs (Anthony and Patrick, 2004) and link grammar (Bayer et al., 2004). However, these systems seems to employ a large number of rules for only a slight improvement in accuracy. Some systems seems to use over a hundred rules (Bayer et al., 2004; Ahn et al., 2004). Unfortunately, descriptions of these systems only mention a few of the rules and do little evaluation of their respective impacts. We aim to give a concise but robust overview that will allow the readers to replicate the system and compare the different rules.

Other work on logical form identification includes Mohammed et al. (2004), Moldovan and Rus (2001), and van Eijck and Alshawi (1992).

3 Preprocessing

As mentioned in the introduction, the system's transformation from a sentence to a simplified logical form relies on a dependency graph, an associated sequence of tokens, the tokens' parts of speech, and morphological base forms. Five modules are used to extract this information: a sentence splitter, a tokenizer, a POS tagger, a dependency-graph parser, and a morphological parser. For the first two modules we use simple regular expressions. We also check each subsequence of tokens and substitute collocations with a single token. For this, we use a list containing collocations extracted from WORDNET, that was supplied for the Senseval 3 task.

We represent the sequence, or list, of lexical tokens with $W = (w_1, \ldots, w_n), w_i = (i, t_i)$ where i is the position of the token in the sequence and t_i is the string representation. We also define a total order $w_i \prec w_j \equiv i < j$.

3.1 Part-of-speech tagging

We denote the particular part of speech of a token with $\mathrm{POS}(w_i)$. As POS tagger we have tried the STANFORD LOG-LINEAR POS TAGGER (Toutanova and Manning, 2000; Toutanova et al., 2003), MXPOST (Ratnaparkhi, 1996), and TREETAGGER (Schmid, 1994). The results we list in Sect. 5 are for MXPOST.

3.2 Dependency-graph parsing

We consider dependency graphs to be directed acyclic graphs (DAGs) $D = (W, A)$ where A is the ordered set of labeled arcs $(w_i, r, w_j), r \in R$, r the dependency function,[1] and where R in particular contains the functions subject (SUB), object (OBJ), and verb chain (VC). We also use the notation $w_i \xrightarrow{r} w_j$ to denote $(w_i, r, w_j) \in A$ and $rel(w_i, w_j) = r \Leftrightarrow (w_i, r, w_j) \in A$. We have tried MSTPARSER (McDonald et al., 2006) and Nivre's MALTPARSER 0.4 ENGSVM (Nivre et al., 2006). The results in Sect. 5 are for MALTPARSER.

3.3 Morphological parsing

For the simplified logical forms, the head of the predicates should consist of the token's base form and a part-of-speech suffix. We use WORDNET to identify the base forms, performing a call for each token independent of its neighbours. See for instance Fellbaum (1998).

4 Predicate Introduction

Tokens are divided into two simple groups based on the class of supported and ignored parts of speech. The forms used at Senseval 3 ignores for instance determiners because they complicate matters and do not have a great impact for many inference and semantics tasks. For each token of a supported part of speech, we create a predicate with its base form as head and a number of slots equal to its arity[2]. Table 1 lists the arities of different part-of-speech classes. For the tokens of parts of speech in the ignored set, we do nothing. The exceptions to this are noun groups and noun compounds for which an additional predicate is introduced to refer to the group/compound rather than to the components.

After these steps, our first example (Eqn. 1) would be equivalent to

$$\begin{aligned} &\text{John:\_n}(s_{1,1}), \text{and}(s_{2,1}, s_{2,2}, s_{2,3}), \\ &\text{dog:\_n}(s_{3,1}), \text{meet:\_v}(s_{4,1}, s_{4,2}, s_{4,3}), \\ &\text{at}(s_{5,1}, s_{5,2}), \text{his}(s_{6,1}), \text{car:\_n}(s_{7,1}), \end{aligned} \quad (2)$$

where $s_{i,j}$ are argument placeholders. We repeat that we call the placeholders slots.

[1] Also called the dependency relation and dependency label.

[2] The arity of a predicate is the number of parameters it has.

Table 1: Suffixes and arities of our part-of-speech classes, and the classes equivalences in the PTB.

POS	Suf.	Arity	PTB
Nouns, regular	:_n	1	NN, NNS, NP, NPS
Nouns groups /compounds		3	(We call this __NN)
Verbs	:_v	2-3	V[B]?[DGNPZ]?
Adjectives	:_a	1	JJ, JJR, JJS
Adverbs	:_r	1	RB, RBR, RBS, WRB
Conjunctions		3	CC, UH
Unary mods.		1	POS, PP$, WP$
Binary mods.		2	IN, TO

The dependency graph models relations between nodes. This is interpreted here as relations between entities. The entity that is involved in this kind of relation is the first argument of each predicate. For example, the arguments of the *to* predicate in our first example (Eqn. 1) are e_1 and x_4. These two entities are also the first arguments of other predicates, x_4 is the first argument of the *dog* predicate and e_1 is the first argument of the event *run*. Furthermore, every entity in Eqn. 1 is the first argument of a verb, noun, or conjunction.

The primary POS tags that we use are listed in Table 1 along with the equivalent tags in the Penn-Treebank (Santorini, 1991).

We believe our system can relatively easily be extended to different arities and with support for any part of speech. At least as long as inference is not required in the transformation step. For instance, the meaning of the sentence *Do not not breathe* is that one should breathe, with a slightly different emphasis. This sentence is represented as

$$\begin{aligned}&\text{do:\_v}(e_1, x_1, e_2) \wedge \text{not:\_r}(e_2, e_3)\\ \wedge&\text{not:\_r}(e_3, e_4) \wedge \text{breathe:\_v}(e_4, x_1),\end{aligned} \quad (3)$$

which could be reduced to

$$\text{do:\_v}(e_1, x_1, e_4) \wedge \text{breathe:\_v}(e_4, x_1) \quad (4)$$

(*Do breathe*). If reductions such as the above are required, it might not be straightforward to extend the system.

Let $\{p_i = pred_i(s_{i,1}, \ldots, s_{i,\text{arity}(p_i)})\}$ denote the set of created predicates where $w(p_i)$ is the associated token, $\text{POS}(p_i)$, its part of speech[3], $\text{arity}(p_i)$, the predicate's arity, and $s_{i,j}$, the j^{th} slot of the i^{th} predicate.

5 Argument identification

So far, the steps have only involved simple rules directly given by the representation or relying on existing systems. The last step is less trivial and consists of identifying the real arguments for the slots.

5.1 Formalization

We introduce some notation that we use in the latter sections. We let $\arg(s_{i,j})$ be the function that maps the j^{th} slot of the i^{th} predicate to an entity. With $\arg(s_{i,j}) \in E$ and $\arg(s_{i,j}) \in O$ we denote that the argument is an event and object, respectively.

The meaning of Eqn. 1 would be the same if we replaced all the occurrences of x_1 with x_5, since x_5 does not appear elsewhere in the formula, i.e. we merely create an unique entity for it. At this point, we therefore concentrate on finding out what slots are mapped to the same arguments and not so much which specific entity it is. In fact, once we know which arguments should be same and which ones should not, it is an easy task to find a satisfying substitution since there are no restrictions on the number of entities we may create.

Aside from identifying which arguments are the same, we must determine if they are events or objects. We will handle these two tasks in different sections even though the problems overlap.

5.2 Simplifications

Each predicate has an associated token and this token has a node in the dependency graph. We also said that the first slot of each predicate should be the entity that participates in the relation that the dependencies model. We therefore do not always distinguish between tokens, predicates, nodes, and entities when talking about argument identification.

We define the argument search space of a slot to be a list of nodes in the DG that will we search through for a suitable argument. The slot is contained in a predicate and the predicate has an associated node in the dependency graph. The space we use is defined as a region in relation this node.

[3]Also defined for the noun groups/compounds even though there are no associated tokens.

Note that the space is over nodes in the graph and not entities.

For a dependency graph, the arguments of a predicate are usually found in the near vicinity of its node in the dependency graph. Particularly often among its children or siblings, and occasionally its parent or grandchildren. It is also not certain that a suitable argument exists in the sentence. We therefore restrict our argument search space to consist of the node's parent, children, and siblings. If no suitable argument is found among these, we assume that the slot should contain an entity that does not appear elsewhere in the formula. To model preferences between nodes, we order the search space in different ways depending on the parts of speech. This is covered in Sect. 5.3.

Different parts of speech also put different requirements on their arguments. For instance, adjectives generally only modify objects, not events. We collect these constraints under a function that depends on the parts of speech, among other things. We handle these functions, which we call argument constraint functions, in Sect. 5.4.

5.3 Argument search spaces

If p is a node in the dependency graph, then let $P(p), C(p), S(p), S_L(p), S_R(p)$ be the lists of, respectively; the parents of the node[4], the children of the node, the siblings of the node, the left siblings of the node, and the right siblings of the node. The lists should be sorted according to the order of appearance of the nodes' respective tokens in the token sequence.

We use the set notation for operations on ordered lists and with this, we implicitly mean that the elements of the resulting list follow the same order as the source list. One could see it as picking elements from left to right from the input list, process it, and place the result in the output list from left to right.

Take these equations as an example:

$$\begin{aligned} Y &= (x \in X | x \in Nouns), \\ X &= (saw, cat, yard, in). \end{aligned} \tag{5}$$

We first pick *saw* and discard it. Next we pick *cat* and place it in Y, and *yard* which we place to the right of *cat*. We finally discard *in* to get the resulting list: $(cat, yard)$.

[4]N.B. there is only one parent per node in dependency graphs.

With the described notation, we may formally define the left- and right-siblings:

$$S_L(p) = (p' \in S(p) | t(p') \prec t(p)) \tag{6}$$
$$S_R(p) = (p' \in S(p) | t(p') \succ t(p)) \tag{7}$$

For our simplified logical forms, we search for suitable arguments for up to two slots for each predicate. The slots for which we do not perform search we call stable, and the remaining slots unstable. The stable slots are the first slots of all noun, verb, and conjunction predicates. Since the only classes with an arity of three are noun groups/compounds, verbs, and conjunctions, no predicate contains more than two unstable slots.

We have noticed that the arguments for the first and second unstable slots are found in a bit different locations. We therefore separate the argument search space for the first and second unstable slot. Note that the spaces contains the same nodes, we only permute them differently. We call these $\mathcal{A}_1(p)$ and $\mathcal{A}_2(p, q)$, where the latter term is the argument search space for the second unstable slot of a node p, given that we have identified the first argument as q (which is another node in the graph, recall that the argument spaces consists of nodes). We first introduce some of our common spaces that we then use to define the above mentioned terms with respect to the nodes' part of speech. The selection of the proper order is part linguistics and part understanding or making observations about the structure of dependency graphs. The permutations we describe are those that we through iterative experiments and analysis have found work well.

Let X^R denote the list X in reverse order, and let multiplication implicitly denote concatenation of tuples, we describe the most common permutations of the search space with

$$\mathcal{A}_{Lcrp}(p) = S_L(p)^R \cdot C(p) \cdot S_R(p) \cdot P(p) \tag{8}$$
$$\mathcal{A}_{pLcr}(p) = P(p) \cdot S_L(p)^R \cdot C(p) \cdot S_R(p) \tag{9}$$
$$\mathcal{A}_{rpLc}(p) = S_R(p) \cdot P(p) \cdot S_L(p)^R \cdot C(p) \tag{10}$$
$$\mathcal{A}_{crpL}(p) = C(p) \cdot S_R(p) \cdot P(p) \cdot S_L(p)^R \tag{11}$$

Notice how $\mathcal{A}_{Lcrp}(p)$ is equivalent to a preorder traversal of the nodes in the argument search space, except with the left siblings reversed. The other spaces are also given by shifting terms.

We now define the default search methods for the first and second arguments. With $\mathcal{A}_{\text{Def},1}(p)$ we denote the default argument search space for the

first unstable slot of node p, and with $\mathcal{A}_{\text{Def},2}(p,q)$, the argument space function for the second unstable argument of node p, given that its first identified argument is q. We define a *shifted* order, $\mathcal{A}_{\text{Shi},1}(p)$, because it is very similar to the default and we refer to it later.

$$\mathcal{A}_{\text{Def},1}(p) = \begin{cases} \mathcal{A}_{Lcrp}(p) & p, P(p) \in Nouns \\ \mathcal{A}_{pLcr}(p) & \text{else} \end{cases} \tag{12}$$

$$\mathcal{A}_{\text{Shi}}(p) = \begin{cases} \mathcal{A}_{rpLc}(p) & p, P(p) \in Nouns \\ \mathcal{A}_{crpL}(p) & \text{else} \end{cases} \tag{13}$$

$$\mathcal{A}_{\text{Def},2}(p,q) = \begin{cases} \mathcal{A}_{\text{Def},1}(p) & q \notin S_L(p) \\ \mathcal{A}_{\text{Shi}}(p) & q \in S_L(p) \end{cases} \tag{14}$$

The default argument search spaces are used for nouns (N), conjunctions (C), adjectives (A) and adverbs (R). If $k = 1, 2$, then

$$\mathcal{A}_{N,k} = \mathcal{A}_{C,k} = \mathcal{A}_{A,k} = \mathcal{A}_{R,k} = \mathcal{A}_{\text{Def},k} \tag{15}$$

The three exceptions to the default rule are: verbs (V), for which we prefer subject, object or verb chain functions, and unary (U) and binary (B) modifiers, for which we always use the shifted order.

$$\mathcal{A}_{\text{V},1}(p) = (p_i | w(p) \xrightarrow{SUB} w(p_i)) \cdot \mathcal{A}_{\text{Def},1}(p) \tag{16}$$

$$\mathcal{A}_{\text{V},2}(p,q) = (p_i | w(p) \xrightarrow{r} w(p_i)) \cdot \mathcal{A}_{\text{Def},2}(p,q), \quad r \in \{OBJ, VC\} \tag{17}$$

$$\mathcal{A}_{U,1}(p) = \mathcal{A}_{B,1}(p) = \mathcal{A}_{B,2}(p,q) = \mathcal{A}_{\text{Shi}}(p) \tag{18}$$

We may now define the search space for the first and secon arguments with respect to their part of speech as

$$\mathcal{A}_k(p) \equiv \mathcal{A}_{\text{POS}(p),k}(p) \tag{19}$$

5.4 Argument constraint functions

The default constraint function, which we call c_{Def}, merely ensures that the first and second arguments are different, the arity of both is at least zero and if both belong to the preceding siblings, then the first argument must not precede the second. q and r are here the first and second identified arguments of the node p.

$$c_{\text{Def}}(p,q,r) \Leftrightarrow \text{arity}(q), \text{arity}(r) > 0 \\ \wedge q \neq r \wedge (q, r \in S_L(p) \Rightarrow q \succeq r) \tag{20}$$

As with the argument search space, we define the constraint function c with respect to its part of speech,

$$c(p,q,r) \equiv c_{\text{POS}(p)}(p,q,r) \tag{21}$$

There are five classes with more specific requirements than the default but at least unary and binary modifiers are defined as the default constraint function and the rest rely on it. Furthermore, during evaluation, we found that the special constraint functions for conjunctions and adverbs virtually contribute with nothing over the default.

- **Noun groups/compounds (N):** All arguments are restricted to nouns.
- **Verbs (V):** Except for SUB/OBJ/VC functions, arguments must be nouns, conjunctions or verbs. These three classes are called stable and are described in the next section.
- **Conjunctions (C):** Generally both arguments, and subsequently its first argument, are either all events or all objects.
- **Adjectives (A):** Restrict to objects.
- **Adverbs (R):** We restrict adverbs to only modify events.

Recall that E is the set of events, O the set of objects, and with $rel(w_i, w_j)$, we denote the dependency relation between nodes w_i and w_j. The above explanations are formally defined as

$$c_N(p,q,r) \Leftrightarrow \text{POS}(q), \text{POS}(r) \in Nouns \\ \wedge c_{\text{Def}}(p,q,r) \tag{22}$$

$$c_V(p,q,r) \Leftrightarrow (\text{POS}(q) \in Stable \vee rel(p,q) \in (*)) \\ \wedge (\text{POS}(r) \in Stable \vee rel(p,r) \in (*)) \\ \wedge c_{\text{Def}}(p,q,r) \tag{23}$$

$$c_C(p,q,r) \Leftrightarrow (q \in E \Leftrightarrow r \in E) \\ \wedge c_{\text{Def}}(p,q,r) \tag{24}$$

$$c_A(p,q,r) \Leftrightarrow q, r \in O \\ \wedge c_{\text{Def}}(p,q,r) \tag{25}$$

$$c_R(p,q,r) \Leftrightarrow q, r \in E \\ \wedge c_{\text{Def}}(p,q,r) \tag{26}$$

$$(*) = \{SUB, OBJ, VC\}$$

5.5 Argument introduction

The next step is merely to test each predicate or pair of predicates in the argument search spaces for each predicate. The first predicate or pair of predicates that fulfills the constraint function is the best candidate as argument(s).

As mentioned earlier, the first slot of each node represents it in the relations. With potential arguments identified, we may therefore simply introduce equalities to represent these in terms of the predicates' slots. If we find that p_k is the first predicate in the search space for slot $s_{i,j}$ that satisfies the constraints, then $\arg(s_{i,j}) = \arg(s_{k,1})$.

Whenever we during search encounter nouns in the argument search space, we first check any associated noun groups/compounds and any children that are conjunctions. This could be formalized as a function of a node which by default only lists the node itself and where nouns are the only exceptions.

The constraint functions are defined with respect to both arguments, but in our system we use a simplified version; we begin by selecting a first argument that could fulfill the constraints for some second argument, we then search for the second argument. If no satisfying second argument can be found, we still keep the first:

$$\hat{c}(p, q) \equiv \exists_r c(p, q, r) \tag{27}$$

It would be preferable to balance the position of the first and second argument in the ordered argument space.

We conclude this step by merely going through slots and assigning entities. If an equality to another slot with an assigned entity exists, we assign the same entity to this slot; if none exists, we create a new entity.

5.6 Entity partitioning

By default, all entities are considered objects. If it appears in the first slot of a verb, or in a conjunction where all the children are events, it is however considered an event. Because some constraint functions require knowledge of whether an entity is an event or not, the argument identification and entity partitioning decisions cannot be made independently. To handle this, our system identifies equalities like the ones above in two steps: the first step containing predicates which do not rely on whether slots refer to events or not, and the second containing those that do.

Table 2: Argument and predicate F-score for the systems of Senseval 3. The values are ± 0.001.

Team	Argument	Predicate
University of Amsterdam	0.709	0.801
LCC	0.776	0.892
MITRE	0.694	0.809
University of Sydney	0.705	0.844
Our system	0.649	0.845

5.7 Verb complements

As mentioned in Table 1 and as seen in Eqn. 1, verbs may have additional arguments besides their standard one or two. We have taken a simple approach for identifying these modifier arguments: We check every path from the parent that does not contain another verb node. If a modifier *at*, *by*, *for*, *from*, *in*, *of*, *on* or *to* is encountered, introduce an equality to its second argument. Preferably one would want to require the first argument to refer to be the verb in question but this lowers the accuracy of our system. This list needs to be extended but we have not explored this further.

6 Results

For the logical forms identification task at Senseval 3, evaluation was done on a test set of 300 sentences. Participating teams submitted their outputs, which were compared to a gold standard. To evaluate our system, we have use the annotated test set of the Senseval 3 conference and a released evaluation script. The evaluation is done in two parts: F-score on predicates and F-score on arguments. The systems at Senseval 3 achieved an argument F-score between 0.694 and 0.776 and a predicate F-score between 0.801 and 0.892. We report here an argument F-score of 0.649 and a predicate F-score of 0.845 on this test set[5]. Table 2 lists the respective systems and their F-scores. The values have been calculated from Rus (2004).

We therefore do worse than the systems at Senseval 3 on the argument level but good on the predicate level, placing just below second place. The evaluation script does not seem to take into consideration all the equalities or inequalities between arguments or whether they are events or entities though.

[5] We have only run the system on the test set for three configurations with minor changes besides the purpose of testing modifications such as those listed in Table 3. The reported results are for a scheme selected prior to these tests. Development was done on a separate supplied set.

Table 3: Impact on the argument precision, recall and F-score for different modifications. The changes are absolute, not relative.

Modification	Prec. %	Rec. %	F-sc. %
No collocations	$+1.5$	-0.1	-0.7
Noun in cases	-4.7	-4.8	-5.4
Not noun in cases	-1.0	-0.5	-1.3
Parent first	-2.0	-2.3	-2.0
Only parent	-7.4	-9.8	-8.5
$\mathcal{A}_k \leftarrow \mathcal{A}_{\mathrm{Def},1}$	-4.0	-3.7	-4.7
$\mathcal{A}_k \leftarrow \mathcal{A}_{\mathrm{Shi}}$	-9.5	-9.1	-10.7
$\mathcal{A}_{\mathrm{Def},2} \leftarrow \mathcal{A}_{\mathrm{Def},1}$	-0.2	-0.4	-0.5
$\mathcal{A}_{V,k} \leftarrow \mathcal{A}_{\mathrm{Def},k}$	-1.2	-1.0	-1.4
$\mathcal{A}_{U/B,k} \leftarrow \mathcal{A}_{\mathrm{Def},k}$	-0.4	-0.3	-0.4
Grandchildren	-0.3	$+0.4$	$+0.1$
Ignore $p \succeq q$	-0.8	-0.7	-0.8
$c_N \leftarrow c_{\mathrm{Def}}$	-0.4	-0.3	-0.4
$c_V \leftarrow c_{\mathrm{Def}}$	-1.3	-1.6	-0.5
$c_C \leftarrow c_{\mathrm{Def}}$	± 0.0	± 0.0	± 0.0
$c_A \leftarrow c_{\mathrm{Def}}$	-0.2	-0.3	-0.4
$c_R \leftarrow c_{\mathrm{Def}}$	-0.1	-0.3	-0.1
No noun checks	$+0.7$	-3.2	-1.1

The impact on the argument F-score for different rules are given in Table 3.

7 Discussion

Observing the mean F-score, our system, the University of Amsterdam, and MITRE are very close to each other. There is then a large difference with University of Sydney and even greater with LCC. We still believe our results are good in comparison to the other systems if they have larger rule sets. Our approach is simple, robust, general and uncluttered and there is a lot of work that can be done to improve it further without being constrained by a complicated or brittle design.

Auxiliary verbs are ignored by the simplified logical forms but it is not trivial as to what should be considered auxiliary verbs in specific contexts. We have difficulties in consistently separating the cases without introducing rules which are too specific. Since this appears even in the development set, we expect it to have a large impact on the test set.

All argument search spaces we described have been permutations of the nodes in the subgraph rooted at a node's parent, excluding the node itself. It seems like the argument search space should frequently contain grandchildren or nodes past the parent, such as the grandparent or uncles. However, carelessly including these classes in all the argument search spaces reduces the accuracy. This is because the search then discovers too many false equalities.

We notice that a fairly large portion of the errors originate from the preprocessing step. A mistake by the POS tagger propagates to the dependency graph, morphological parser, and the application of transformation rules. On the development set, we at one point noticed a difference of 4% in the logical form identification accuracy caused only be a 1% improvement in POS tagging (absolute units). It seems the accuracy would benefit a fair amount from further improvement of the part-of-speech and dependency-graph modules in particular.

Our system assumes that all noun groups/compounds should consist of components of exactly two objects each. In one example however, a noun group/compound of three objects was used with an arity of four. A related difficulty is the nesting of conjunctions in which its not immediately clear how the dependency-graph parser deals with them from case to case.

8 Conclusions

It appears as though we can conclude that transforming dependency graphs to the simplified logical forms with high accuracy is possible through a few simple rules. Formalizing them as search rules has reduced the complexity particularly and we believe this scheme should make further improvement easy. Previous work in the area (Ahn et al., 2004) confirms the conclusion that it is relatively straightforward to implement a fair transformation scheme. We would like to add that implementing such a system is anything but complex or time-consuming, given the modules mentioned above.

9 Future Work

Systems for inference and semantics tasks involving logical forms frequently use different certainty factors for making the right choices. Taking a more probabilistic approach to logical forms identification, which does not necessarily need to be statistical, should both improve the accuracy of the logical forms identification as be of use for the reasoning tasks. This is also partly our way to approach the last paragraph of the discussion.

We have experimented with statistical logical form identification and believe that the scheme we have presented here provides a useful basis. There are however still a variable number of parameters for the decisions and they might not be as independent as preferable. A downside today is the lack of training data.

We have described three simplifications in our system: 1) we restrict the search space to nodes at distance one and two, 2) we greedily select the first argument before the second even though this is not warranted, and 3) we process slots which depend on whether other slots are events or not without any particular order after those that do not share this dependency. It would exciting to explore how to relieve or better handle these simplifications, in particular without causing an explosion of the argument search space.

10 Acknowledgment

The work presented here has received much support and guidance from Pierre Nugues at Lund university, both on the construction of the system and in particular on the authoring of this paper. He would likely have wanted to make many more suggestions on both.

References

David Ahn, Sisay Fissaha, Valentin Jijkoun, and Maarten De Rijke. 2004. The University of Amsterdam at Senseval-3: Semantic roles and Logic forms. In *Senseval-3*, pages 49–53. Association for Computational Linguistics.

Stephen Anthony and Jon Patrick. 2004. Dependency based logical form transformations. In *Senseval-3*, pages 54–57. Association for Computational Linguistics.

Roy Bar-Haim, Ido Dagan, Bill Dolan, Lisa Ferro, Danilo Giampiccolo, Bernardo Magnini, and Idan Szpektor. 2006. *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*.

Samuel Bayer, John Burger, John Greiff, and Ben Wellner. 2004. The mitre logical form generation system. In *Senseval-3*, pages 69–72. Association for Computational Linguistics.

Fellbaum. 1998. *WordNet: An Electronic Lexical Database*. The MIT Press.

R. McDonald, K. Lerman, and F. Pereira. 2006. Multilingual dependency analysis with a two-stage discriminative parser.

Altaf Mohammed, Dan Moldovan, and Paul Parker. 2004. Senseval-3 logic forms: A system and possible improvements. In *Senseval-3*, pages 163–166. Association for Computational Linguistics.

Dan I. Moldovan and Vasile Rus. 2001. Logic form transformation of wordnet and its applicability to question answering. In *ACL '01*, pages 402–409. Association for Computational Linguistics.

Dan Moldovan, Christine Clark, Sanda Harabagiu, and Steve Maiorano. 2003. Cogex: a logic prover for question answering. In *NAACL '03*, pages 87–93. Association for Computational Linguistics.

Joakim Nivre, Johan Hall, and Jens Nilsson. 2006. Maltparser: A data-driven parser-generator for dependency parsing. pages 2216–2219.

Adwait Ratnaparkhi. 1996. A maximum entropy model for part-of-speech tagging. In *Proceedings of the Conference on Empirical Methods in Natural Language Processing*.

Vasile Rus. 2001. High precision logic form transformation. In *ICTAI*, pages 288–.

Vasile Rus. 2004. A first evaluation of logic form identification systems. In *Senseval-3*, pages 37–40. Association for Computational Linguistics.

Beatrice Santorini. 1991. Part-of-speech tagging guidelines for the penn treebank project.

Helmut Schmid. 1994. Probabilistic part-of-speech tagging using decision trees. In *International Conference on New Methods in Language Processing*. unknown.

Marta Tatu, Brandon Iles, John Slavick, Adrian Novischi, and Dan Moldovan. 2006. Cogex at the second recognizing textual entailment challenge. In *Proceedings of the Second PASCAL Challenges Workshop on Recognising Textual Entailment*.

Kristina Toutanova and Christopher D. Manning. 2000. Enriching the knowledge sources used in a maximum entropy part-of-speech tagger. In *EMNLP/VLC-2000*.

Kristina Toutanova, Dan Klein, and Christopher D. Manning. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of HLT-NAACL 03*.

J. van Eijck and H. Alshawi. 1992. Logical forms. In *The Core Language Engine*, pages 11–40. MIT Press.