Donkeys to Monoids: A Computational Framework for Dynamic Semantics

Thomas Dybdahl Ahle

May 30, 2013

Abstract

In this paper I build a fully automatic system for translation of natural language into formal logic. I combine a variety of tools and theories to target standard issues such as anaphora resolution, sentence polarity etc. In particular I base my framework on dynamic logics in form of Dynamic Predicate Logic and a set of enhancements by Albert Visser[27].

'Donkey Monoids', as I will call these monoidal DPL variants, are part of a movement in linguistics towards logics that can more naturally approximate human language in aspects such as polarity and scope. Indeed their name derives from the famous 'Donkey Sentence' by Geach[7] which exhibit these issues so well.

Visser didn't test his ideas computationally and neither did most people involved with DPL. In fact most of the classical literature on logification of natural language exhibits this 'problem': Not testing computationally makes it easy to miss obvious details. With the present paper I supply such an experiment.

I build a rewriting system with a well defined set rules, discussing possible semantics. I apply the framework to a large collection of sentences, showing the conversion process step by step. Finally I discuss ways to enlarge the supported fragment of English, how we can combat and interpret the inherent ambiguity we meet, and how the flexibility of Donkey Monoids relate to the classical Montague lambda structures.

Keywords: Natural Language Semantics, Meaning, Dynamic Predicate Logic, Donkey, Monoid, Montague Grammar, Computational, Haskell

Contents

1 Introduction

There are many reasons why it is interesting to represent information in a logical language. For a computer scientist, the biggest advantage is that we can compute with logic. There is a horde of algorithms for doing querying, question answering, reasoning, unification and resolution - to all of which they key is a formal logic definition of your information.

Knowledge bases In the field of Knowledge Representation and Reasoning, they long ago discovered that formal logic provides a good language for building ontologies. Typical users are crime fighters or life scientists that need to draw conclusions based on complex systems[17] where most of the known facts are only described in natural language. Today those people use mainly have to manually build large models in GUI tools, but an automated approach is much wanted.

Semantic search From the early days of database systems logic was identified[5] as a good language for querying. Today technologies like Facebook Graph Search allows users to execute searches like 'Restaurants in London my friends have been to'. The missing piece is a reliable translation system.

Robots Since the beginning of Artificial Intelligence, logic based approaches have done well.[25] Robots need to store relationships they learn about the world and react to it. For modern 'robots' like Apple's Siri to be able to communicate with people, a translation layer between natural language and logic is a plausible path.

It would seem that getting computers to understand and work with natural language is on of the biggest challenges in information technologies this day. Moving the problems into the space of formal logic is a classical and tempting path. It does however remain open whether our formal logics actually have the power to express most of the sentences people casually voice on a daily basis. In this paper we will assume it does for enough practical cases to be useful.

The reason logics like DPL and Visser Monoids exist is a famous sentence noted by Peter Thomas Geach in 1962[7]: 'Every farmer who owns a donkey beats it'. As we will see, this sentence illustrates that even if we restrict ourselves to factual sentences, formal first-order logic still runs into a lot of problems when representing many natural sentences. In fact this sentence broke early automatic approaches to logification, and led to the construction of Discourse Representation Theory (DRT)[12].

Before I go into more details on my automatic logification scheme, let's have a look at some of the issues we run into if we try approach the task manually. This

should help us get a feeling of what a formal logic translation of natural language actually looks like:

1. 'Socrates is a donkey': $donkey(Socrates)$. This is a classical logical example, treating Socrates as a constant. If we want to do without constants we can write: $\forall x(Socrates(x) \rightarrow donkey(x))$. See how this allows for more than one Socrates to exist.

2. 'All his donkeys are treated well': This sentence somewhat willingly lets itself translate into: $\forall y(his(x, y) \wedge donkey(y) \rightarrow treated\_well(y))$, which has a free variable x referring to some (male?) entity in the context of the discourse.

3. 'Every farmer beats his donkey': This sentence clearly has two possible meanings: $\forall x(farmer(x) \rightarrow \exists y(beats(x, y) \wedge his(x, y) \wedge donkey(y))$, if 'his' refers to the farmers, or $\exists y(his(x, y) \wedge donkey(y) \wedge \forall z(farmer(z) \rightarrow beats(z, y))$ if it refers to somebody else. There is no way we can know without more context information.

4. 'The happy donkey doesn't exist': Here we get into more problems. We might write: $\exists x(happy\_donkey(x) \rightarrow doesn't\_exist(x))$, but that asserts some sort of entity representing the non existing donkey. Alternatively we could write $\exists x(happy\_donkey(x) \rightarrow \bot)$, but that is another sentence: 'There are no happy donkeys'. In any case it makes it hard to say what the logical meaning of 'the' really is.

5. 'The donkeys are being stubborn!': This is an often used example of an 'implied action' sentence and thought to have the same meaning as 'Please go out and beat them!'. The best we can do here seems to be $please\_beat\_the\_donkeys()$ or perhaps $\forall x(donkey(x) \rightarrow please\_beat(x))$ or even $please(\forall x(donkey(x) \rightarrow beat(x)))$.

6. 'Jane beats a donkey': This is easy to translate to $\exists x(donkey(x) \wedge beat(Jane, x))$, but are we comfortable that this is also our interpretation of the sentence 'It is not the case that Jane doesn't beat a donkey' and even 'Jane beats a donkey and either it is raining or it isn't'? It is hard to imagine a language with strong logical foundations, where those three sentences wouldn't be equal.

Hopefully those examples make a convincing argument that some uses of English, especially example 4 and 5, are well beyond what we can hope to automatically logify. However you might think the sentences of 'factual nature' like 1 and 2 look pretty trivial? In this paper we will see that they are certainly possible,

but as sentences get longer and more complex, certainly tricky, and this is where Donkey Monoids, Dynamic Predicate Logic (DPL) and Discourse Representation Theory (DRT) come into play. Let's take a close look at that 'Donkey Sentence': 'If a farmer owns a donkey, he beats it'.

From the examples before, and perhaps from common intuition, we might find it natural to think of the sentence as two sub-sentences combined by 'if': 'if (a farmer owns a donkey) (he beats it)'. Translating the parentheses first: 'A farmer owns a donkey' logifies to $\exists x(farmer(x) \wedge \exists y(donkey(y) \wedge owns(x,y))$, and 'he beats it' translates to $beats(x,y)$. Hence the whole formula should probably be:

$$\exists x(farmer(x) \wedge \exists y(donkey(y) \wedge owns(x,y))) \rightarrow beats(x,y) \tag{1}$$

Which is totally wrong. The $beats(x,y)$ clause ends up referring to the variables outside of their scope, and even if we try to push it inside the parentheses:

$$\exists x(farmer(x) \wedge \exists y(donkey(y) \wedge owns(x,y) \rightarrow beats(x,y))) \tag{2}$$

It doesn't work either. Sentence (2), while syntactically correct, evaluates to true as long as a single farmer beats his donkey. In fact **as long as any farmer exists** (2) evaluates to true. This is because $\exists y$ also spans over the farmer entity (and he's not a donkey) which makes the antecedent false. Clearly this is not the formula we are after.

What turns out to be the correct logification is this:

$$\forall x(farmer(x) \rightarrow \forall y(owns(x,y) \wedge donkey(y) \rightarrow beats(x,y))) \tag{3}$$

A far cry from our original 'guess'. Moreover the above version even has the oddity that the interpretation of 'all' is 'the logical one': The formula (3) is, in addition to the intuitive situations[1], also considered true if no farmers exist in the domain, or none of them own donkeys.

Now the above example may seem a bit contrived. Why did I have to try to combine the meaning from the obviously wrong pieces? Wasn't some other strategy more obvious? Before we go into details of such logification strategies, let's first see how easy it could have been, if we instead of first-order logic had used Visser's Donkey Monoids:

$$\bowtie \cdot \exists x \cdot farmer(x) \cdot owns(x,y) \cdot \triangle \cdot \exists y \cdot donkey(y) \cdot \triangle \cdot \bowtie \cdot beats(x,y) \tag{4}$$

If we read $\exists$ as 'a' this is basically identical to the English sentence. The donkey monoid even supports placing the transitive verb between the two referents. Clearly some magic is going on, that we will have to investigate further.

[1]For non logicians at least.

Neat as the Donkey Monoids are, and interested as we are in their automation, we should be very concerned about those magical $\bowtie$s and $\triangle$s which have popped up in our formula. We'll want to ask questions such ask (1) What do they mean? (2) How do they work? And (3) how do we know when to use them? Visser himself, while answering question 1 and 2 in great detail, doesn't give many hints on 3. Only that they 'switches' are probably controlled by adverbs and other support words.

So if we don't know how to use them, how can we know if the Donkey Monoids are good logics for natural language? In the later chapters I will discuss their formal meaning in more detail and also what semantic meaning I intend to give them. In the final chapters we will see just how much flexibility this approach allows us.

We need a framework to encode our rules into, and luckily the field of computational semantics offer several solutions. Basically there are two well described ways to go: statistical vector space methods, which is very much the most popular alternative right now, and logical methods. As you might have guessed I will lean in the direction of the later. The most well described methods are Montague Grammars and Discourse Representation Theory (invented after the donkey sentence).

In the early sixties the American logician, Richard Montague, was teaching a class in which the students had to do logification of simple sentences. He noticed that there were many patterns in the way they did it, and conjectured that it would be possible to formalize the entire process. So he set out to write a series of papers that became very influential in the philosophy of language, linguistics and computer science.

The Montague Grammar, as Montague's approach came to be know, has survived in computational linguistics for decades. Perhaps its main charm is in its rich promises for a simple, consistent way to describe all of language. The basic idea is to define logical formulas recursively over the so called constituent tree of a sentence. The tree is based on the idea of context free grammars for natural language developed mainly by Noam Chomsky[2]. Figure 1 is an example of what one might look like.

We will see later how exactly the Montague Grammar obtains a full sentence formula from its recursive process, but in any case for a computer scientist the introduction of trees is always a promising step.

The main thing to notice now is that for Montague's approach to work, it is crucial that we can always get the meaning of a composed phrase from a composition of the meaning of its sub-phrases. This compositional property is exactly what we saw earlier that first-order logic doesn't posses, and Montague goes through many hoops to simulate it.

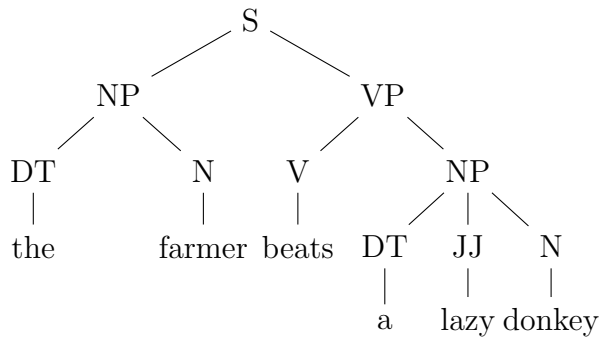


Figure 1: A Constituent Tree

I do however avoid many of those problems, since my choices of logic behave much more compositionally than the FOL in Montague's proposed rules. To draw a full circle I am however going to translate the obtained Donkey Monoid representations into first-order logic so we can easily check if our formulas have the meanings we expect.

This is the chain I am going to build:

1 Natural Language We start with a file of simple English natural language sentences. Each line is processed on its own, but may contain more than one sentence.

2 POS tagger This is the first of three Stanford NLP programs my Haskell script is going to pass the text to. Then the POS tagger tokenizes the sentences and statistically tries to figures out part of speech for each word. This is also my first source of error.

3 Constituent Grammar It might be that the reason Montague and other early pioneers didn't test their programs mechanically was their lack of access to a good parser. The Stanford Parser is the best on the market, but it still an error rate of around 5% which is one in every two sentences.

4 Coreference Resolution A lot has been said about how anaphora might be resolved using formal Montague Grammar like approaches. DRT is especially focused on this topic. None the less the current champion of anaphora competitions[16][22] is the heuristical Stanford Deterministic Coreference Resolution System, the workings of which I will explain in detail.

5 Donkey Monoid For us the step from 4 to 5 is the most interesting, as it is where we go from syntax to semantics. If we manage to push our original

text through to here, not much can go wrong. Only issue is if the logical meaning we create is not the one intended by the speaker.

6 Dynamic Predicate Logic Since the Donkey Monoids are variants of DPL, they are easy to convert into it. It is much more surprising that DPL lets itself convert into first-order logic.

7 First-Order Logic We end up translating our dynamic logics into this old friend. This is mainly done to test the conversion algorithm, and because the first-order logic is what we are used to read. The results can be surprising.

However before we can go into the many technical details, we need to develop a very good understanding of the logics and ideas we are working with.

2 Dynamic Semantics and Montague

In these chapters I'm going to describe the logical theories by Visser and earlier dynamic semanticists. I will introduce some additions of my own and prove interesting properties about them. Finally, but most importantly, I will go into depth with the tools and ideas we need to write a strong logification framework.

In the previous chapter I showed how first-order logic lacks certain compositional properties. A possible solution to this problem is to use a so called 'update logic' (a logic with update semantics). Update semantics are logics inspired on the semantics of programming languages where a variable, once introduced, will stick around for a period not know at the time of its introduction.

The idea of those logics is a 'growth of information in time', starting from no information, and with each statement adding more. Information is here seen in the semantic way, where no information is when everything is possible ($\top$) and total information is when nothing is possible ($\bot$).

Logicians and linguists have come up with many different update logics since the 80s. I will quickly describe the four most cited ones, to give an idea of the landscape we're working in:

- Discourse Representation Theory, 1981 by Hans Kamp[11], is perhaps the best known approach to using update semantics for understanding language. Kamp takes a very philosophical approach, thinking of his logic as a model for how a person builds up a context during a conversation.

  For example, say you you listen to 'Some farmer owns no donkey. He beats it'. Discourse Representation Theory (DRT) builds this up from two blocks. Each consists of a set of referents and a set of properties using those. Underlined referents are referents that need to be 'merged' with a properly 'introduced' one:

  $$\begin{aligned} &[_1x \mid farmer(x), \neg[_2y \mid donkey(y), owns(x,y)]] \\ \oplus\ &[_1\underline{v}, \underline{w} \mid beats(v,w)] \end{aligned}$$

  The rule for '$\oplus$' is simply to do a disjoint union of the referents and the properties:

  $$[_1x, \underline{v}, \underline{w} \mid farmer(x), \neg[_2y \mid donkey(y), owns(x,y)], beats(v,w)]$$

  At this step it is possible to perform anaphora related heuristics. Since x is the first parameter of $farmer$, we assume it should be so for *beats* as well:

  $$\begin{aligned} &[_1x, \underline{v}, w \mid farmer(x), v = x, \neg[_2y \mid donkey(y), owns(x,y)], beats(v,w)] \\ &[_1x, w \mid farmer(x), \neg[_2y \mid donkey(y), owns(x,y)], beats(x,w)] \end{aligned}$$

On the other hand y is not known in the scope of *beats*, so there is no way we can get rid of w. And we shouldn't, because in the sentence there is of course no way 'it' can refer to 'a donkey'.

To deal with our example of 'If a farmer owns...', DRT introduces more symbols for weak and strong conditionals.

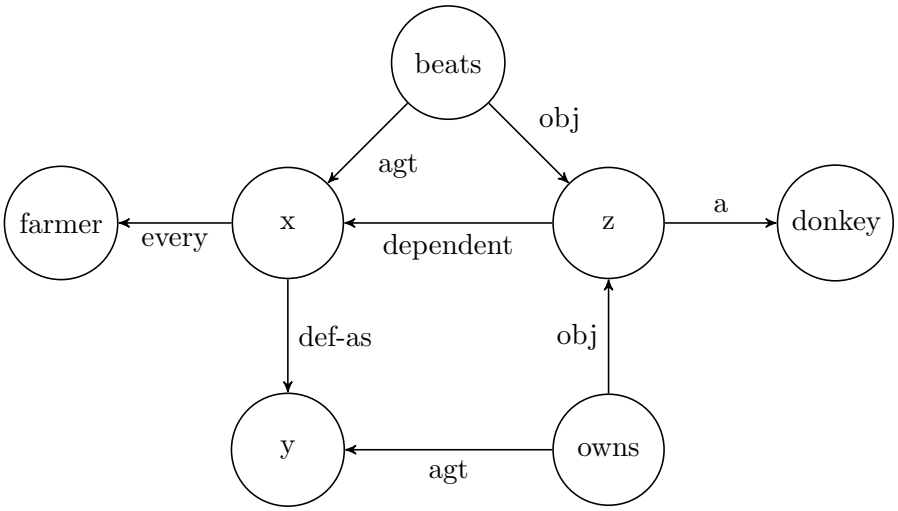
- File Change Semantics, 1982 by Irene Heim[10], are based around the puzzle of discourse 'referents' which may, as we have seen, sometimes act like quantifiers, sometimes referents and sometimes antecedents. Heim suggests using so called 'file cards' which though developed independently, has a lot in common with DRT.

  However, where DRT defines complex rules for merging different 'discourse referents', in File Change Semantics (FCS) the part 'added onto' a file gets to decide how the merge is made. Some how similar to how lambda functions in Montague grammars give you a lot of flexibility. For example in case of 'adding' p, an n-ary atomic predicate, to a file F:

  $$Sat(F + p) = \{a \in Sat(F) \mid R(a_{i_1}, \ldots, a_{i_n})\}$$

  Where Sat is the function that takes a file to the set of 'individuals' 'satisfying' it.

- Mental Spaces, 1984 by Fauconnier[6], is not really a logic in that it's not build on a model theoretic interpretation. It does however build on the same idea of 'information growth' on a context representation. Mental Spaces (MS) represent the current context as a graph of predicates and variables:



Notice here in particular the interesting 'dependent' arrow. This arrow communicates that there is a one-many relationship between farmer and donkey, and hence that we are talking about a specific donkey for each farmer, not a single poor donkey beaten by everyone.

- Dynamic Predicate Logic, 1991 by Groenendijk and Stokhof[8], is what I will focus on in the remains of this paper. In contrast to some of the previous models, the authors of Dynamic Predicate Logic (DPL) don't make any big philosophical claims on connections between their model and the brain. Instead they simply create a logic that is isomorphic to PL, while keeping a reasonable amount of 'update semantics' like properties, such as being closer syntactically to natural language.

  DPL also doesn't provide any guidance into resolving anaphora. Just like giving meaning to the actual concepts, this must be handled outside of the logic. We will see later in this paper that this stance makes sense, since the current state of the art anaphora resolvers are based on simple heuristics over noun phrases.

  In the end of this chapter we will see how a certain monoid construction by Visser can give us as much flexibility in DPL as we would get with a more involved system such as DRT or FCS.

2.1 DPL

Dynamic Predicate Logic has been developed by a wide group of people: Jan van Eijck and Fer-Jan de Vries[4], Hans Kamp, Veltman, Groenendijk and Stokhof. It is the linguist version of Dynamic Logic invented by Pratt[21] which is again built on top of Floyd-Hoare Logic for reasoning about computer programs.

DPL cleverly uses this basic logical research to create a very simple, self-contained logic, while still getting all the benefits of compositionality etc. To model a simple sentence 'A man comes in. He sees a donkey. He smiles.', we write:

$$\begin{aligned} &(\exists x \cdot man(x) \cdot comes\_in(x)) \\ \cdot\ &(\exists y \cdot donkey(y) \cdot sees(x, y)) \\ \cdot\ &(smiles(x)) \end{aligned}$$

(The parenthesis is only for clarity. The logic is 100% associative). Notice how we introduce referents casually as we progress in the discourse. We even manage to look pretty similar to PL at the same time.

Too understand the semantics of the above, let's start by considering a simple example with two variables (a and b) and a domain with three entities (0, 1 and 2). I said earlier, that the update semantics modeled 'growth of information over time', so we'll start with the case that every 3^2 assignments are possible:

$$\begin{aligned}\{&\langle a \mapsto 0, b \mapsto 0\rangle, \langle a \mapsto 0, b \mapsto 1\rangle, \langle a \mapsto 0, b \mapsto 2\rangle,\\ &\langle a \mapsto 1, b \mapsto 0\rangle, \langle a \mapsto 1, b \mapsto 1\rangle, \langle a \mapsto 1, b \mapsto 2\rangle,\\ &\langle a \mapsto 2, b \mapsto 0\rangle, \langle a \mapsto 2, b \mapsto 1\rangle, \langle a \mapsto 2, b \mapsto 2\rangle\}\end{aligned} \tag{5}$$

A DPL formula is defined as a relation between assignments, and $[\![\cdot]\!]$ is the function that takes a formula to its relation. For example $\langle a \mapsto 1, b \mapsto 1\rangle [\![a = 1]\!] \langle a \mapsto 1, b \mapsto 1\rangle$ is true because the $a = 1$ 'program' executes successfully. We also require that simple tests like this one doesn't change the state, so $\langle a \mapsto 1, b \mapsto 2\rangle$ would not have been a valid assignment on the right hand side.

Just like with programs, we can combine multiple formulas into bigger ones: $\langle a \mapsto 0, b \mapsto 2\rangle [\![a = 0 \cdot b = a]\!] \psi$ is false for all ψ, and $\langle a \mapsto 2, b \mapsto 1\rangle [\![a = 0 \cup b < 2]\!] \langle a \mapsto 2, b \mapsto 1\rangle$ is true. If we take the image of (5) under $a = 0 \cdot \neg(b = 2)$ we get

$$\{\langle a \mapsto 0, b \mapsto 0\rangle, \langle a \mapsto 0, b \mapsto 1\rangle\} \tag{6}$$

If we take the image of (6) under $a = 1$, we end up with the empty set of states. That is because there are no possible assignments satisfying our formula. Hence we consider $a = 0 \cdot \neg(b = 2) \cdot a = 1$ a falsum, also defined as $\bot$.

So why do we define all of this using relations, and not just functions from states to truth values? The reason is that we want to introduce a special relation $\exists x$ (for some variable name x). We want it to analogous to $\exists x(...)$ from first-order logic. It's not immediately obvious what the correct definition should be.

The trick is to consider the conjunctive DPL relation $\exists x \cdot x = 1$ - we want this to work such that $\langle a \mapsto 0\rangle [\![\exists x \cdot x = 1]\!] \langle a \mapsto 0, x \mapsto 1\rangle$. Now considering $x = 1$ is a filter, it must be that $\exists x$ is a relation that allows any value for x on its second argument. For example we see that the image of (5) under $\exists x$ has 27 possible states.

Let's have a look at the model theoretic definition of the semantics. This is sometimes done in two steps: First defining a relational algebra and then builid DPL on top of it. However DPL is basically identical to the relational algebra, so we are going to try and do everything in one step:

Let a signature Σ for DPL be a structure $\langle Pred, Ar, Var\rangle$. $Pred$ is the set of predicate symbols containing at least $=$, $\top$ and $\bot$. Ar is the function from predicate symbols to their arity and in particular we have $Ar(=) \mapsto 2$, $Ar(\top) \mapsto 0$, $Ar(\bot) \mapsto 0$. Finally Var is the set of variable symbols we may use. We are not going to work with constants in this treatment.

We are further going to define:

$$Prop_\Sigma = \{P(x_1, \ldots, x_n) \mid P \in Pred,\ Ar(P) = n,\ x_1, \ldots, x_n \in Var\} \tag{7}$$
$$Reset_\Sigma = \{\exists x \mid x \in Var\} \tag{8}$$
$$Atom_\Sigma = Prop_\Sigma \cup Reset_\Sigma \tag{9}$$

The set of formulas we will say to be the smallest set $Form_\Sigma$ satisfying:

$$Atom_\Sigma \subseteq Form_\Sigma \tag{10}$$
$$\text{If } \psi \in Form_\Sigma \text{ then } \neg(\psi) \in Form_\Sigma \tag{11}$$
$$\text{If } \psi_1, \psi_2 \in Form_\Sigma \text{ then } \psi_1 \cdot \psi_2 \in Form_\Sigma \tag{12}$$
$$\text{If } \psi_1, \psi_2 \in Form_\Sigma \text{ then } \psi_1 \cup \psi_2 \in Form_\Sigma \tag{13}$$

A model $\mathcal{M}$ of a signature Σ is a tuple $\langle \mathcal{D}, \mathcal{I} \rangle$. The model is very similar to first-order logic in that $\mathcal{D}$ is the domain, a set of entities, and $\mathcal{I}$ is the interpretation function $Prop_\Sigma \to (Asgmt \to \mathbb{B})$, ($Asgmt$ being the set of possible assignments, $\mathbb{B}$ the set of boolean values).

Just like in first-order logic we want to denote whether a formula is true or false, and just like in first-order logic we do this by saying that an assignment can satisfy a formula, defined as below. If a formula is satisfied by all assignments (equivalent to being satisfied by the empty assignment), it is a tautology.

$$\alpha \models \psi :\equiv \exists \beta(\alpha[\![\psi]\!]\beta) \tag{14}$$

We then define our relations in the set theoretic way, as sets of pairs:

$$[\![\psi_1 \cdot \psi_2]\!] :\equiv \{\langle \alpha, \beta \rangle \in Asgmt^2 \mid \exists \gamma(\langle \alpha, \gamma \rangle \in [\![\psi_1]\!] \wedge \langle \gamma, \beta \rangle \in [\![\psi_2]\!])\} \tag{15}$$
$$[\![P(x_1, \ldots, x_n)]\!] :\equiv \{\langle \alpha, \alpha \rangle \in Asgmt^2 \mid \mathcal{I}(P)(\alpha)\} \tag{16}$$

Or we can define them more succinctly, pointwise for all assignments α, β:

$$\alpha[\![\psi_1 \cdot \psi_2]\!]\beta :\equiv \exists \gamma(\alpha[\![\psi_1]\!]\gamma \wedge \gamma[\![\psi_2]\!]\beta) \tag{17}$$
$$\alpha[\![\psi_1 \cup \psi_2]\!]\beta :\equiv \alpha[\![\psi_1]\!]\beta \vee \alpha[\![\psi_2]\!]\beta \tag{18}$$
$$\alpha[\![P(x_1, \ldots, x_n)]\!]\beta :\equiv \alpha = \beta \wedge \mathcal{I}(P)(\alpha) \tag{19}$$
$$\alpha[\![\neg\psi]\!]\beta :\equiv \alpha = \beta \wedge \alpha \not\models [\![\psi]\!] \tag{20}$$
$$\alpha[\![\exists x]\!]\beta :\equiv \forall \omega(\omega \neq x \to \alpha_\omega = \beta_\omega) \tag{21}$$

Using those basic axioms we can derive $\top$ and $\bot$ given our usual interpretations:

$$\alpha[\![\bot]\!]\beta \equiv \alpha = \beta \wedge \mathcal{I}(\bot)(\alpha) \equiv \bot \tag{22}$$
$$\alpha[\![\top]\!]\beta \equiv \alpha = \beta \wedge \mathcal{I}(\top)(\alpha) \equiv \alpha = \beta \tag{23}$$
$$\tag{24}$$

Taking $\psi_1 \to \psi_2$ to mean $\neg(\psi_1 \wedge \neg\psi_2)$ we also get the following neat equation, originally by Groenendijk and Stokhof[8]

$$\begin{aligned}
\alpha[\![\psi_1 \to \psi_2]\!]\beta &:\equiv \alpha[\![\neg(\psi_1 \wedge \neg\psi_2)]\!]\beta \\
&\leftrightarrow \alpha = \beta \wedge \alpha \not\models [\![\psi_1 \wedge \neg\psi_2]\!] \\
&\leftrightarrow \alpha = \beta \wedge \neg\exists\gamma(\alpha[\![\psi_1 \wedge \neg\psi_2]\!]\gamma) \\
&\leftrightarrow \alpha = \beta \wedge \neg\exists\gamma(\exists\delta(\alpha[\![\psi_1]\!]\delta \wedge \delta[\![\neg\psi_2]\!]\gamma)) \\
&\leftrightarrow \alpha = \beta \wedge \neg\exists\gamma(\exists\delta(\alpha[\![\psi_1]\!]\delta \wedge \delta = \gamma \wedge \delta \not\models [\![\psi_2]\!])) \\
&\leftrightarrow \alpha = \beta \wedge \forall\gamma(\forall\delta(\delta = \gamma \to (\alpha[\![\psi_1]\!]\delta \to \delta \models [\![\psi_2]\!]))) \\
&\leftrightarrow \alpha = \beta \wedge \forall\gamma(\alpha[\![\psi_1]\!]\gamma \to \gamma \models [\![\psi_2]\!]) \qquad (25)
\end{aligned}$$

In my implementation section, I describe how the above can be efficiently implemented in Haskell using an isomorphism between relations $\mathcal{P}(A \times B)$ and functions $A \to \mathcal{P}(B)$. I also show how the Kleisli combinator takes the role of $\cdot$.

However let's first look at some of the implications of the above definitions. For example, how can we intuitively understand the DPL notion of $\to$? Well, first notice that many of the equations start with $\alpha = \beta \wedge \ldots$. These equations we can think of intuitively as having a closed scope. We saw earlier how DRT nicely determined that the 'it' in 'Some farmer owns no donkey. He beats it' couldn't refer to 'a donkey' because the negation didn't let its variable out. In DPL we get the same effect. Because an assignment must be the same before and after a negated clause, all new variables introduced inside the clause have disappeared.

Of course this 'static negation' might not always be the right thing. Consider the sentences 'It is not that case that a farmer doesn't own a donkey. He beats it'. This should be a good sentence, but we can't handle it. In section 6.1 I will look at how this might be handled.

The DPL $\psi_1 \to \psi_2$ in particular can be seen as the filter, that if you can get 'through' ψ_1 then you must also be able to get through ψ_2. But notice that γ doesn't have to equal α. Hence variables introduced in ψ_1 can well bind variables in ψ_2. This will show to be extremely important in the treatment of donkey sentences later.

Talking about binding, also notice that our union or 'or' construct is a bit special. ψ_1 and ψ_2 cannot bind variables in each other, but in $(\psi_1 \cup \psi_2) \cdot \psi_3$ they can both bind variables in ψ_3. We won't actually use 'or' in the main implementation[2], but in the appendix 6.1 I look more closely at the weird situations that can arrise. I also consider the alternative definition of 'or': $\neg(\neg\psi_1 \cdot \neg\psi_2)$.

[2]Turns out the Stanford tools find it weird as well.

2.1.1 Connection with First-Order Logic

An interesting property of DPL is that it is isomorphic with classical logic. Any PL formula corresponds to one in DPL and vice versa. This gives an alternative way to define and understand the semantics of DPL, and it relaxes us that nothing 'weird' is going on. We might also be a bit disappointed that after all this work, we don't possess any extra power, but let's forget about that for now, and see how the conversion works.

We can define a conversion function $[\![\cdot]\!]$ from FOL to DPL very easily. The only trick is that we need a way in DPL to create the closed scopes used in FOL. $\neg(\neg(\cdot))$ does nicely:

$$[\![P(x_1, \ldots, x_n)]\!] \equiv P(x_1, \ldots, x_n) \tag{26}$$
$$[\![\neg(\phi)]\!] \equiv \neg[\![\phi]\!] \tag{27}$$
$$[\![\phi_1 \wedge \phi_2]\!] \equiv [\![\phi_1]\!] \cdot [\![\phi_2]\!] \tag{28}$$
$$[\![\phi_1 \vee \phi_2]\!] \equiv [\![\phi_1]\!] \cup [\![\phi_2]\!] \tag{29}$$
$$[\![\exists x(\phi)]\!] \equiv \neg(\neg(\exists x \cdot [\![\phi]\!])) \tag{30}$$
$$[\![\forall x(\phi)]\!] \equiv \exists x \rightarrow [\![\phi]\!] \tag{31}$$

Similarly we can create a function from DPL to FOL, but perhaps we should be a bit concerned on how to interpret the meaning of an FOL formula and a DPL formula being equivalent. After all DPL formulas are relations where PL formulas are propositions. In the above it was all easy, but let's think again about what we are doing.

We are going to use our definition of $\alpha \models \phi$ for DPL formulas. An FOL formula ϕ and a DPL formula ψ are defined to be equivalent exactly when $\forall\alpha(\alpha \models \phi \leftrightarrow \alpha \models \psi)$. Perhaps the best way to see why this is a good definition (other than its simplicity) is to think programs and weakest preconditions. Every initial state for the 'computation' $farmer(y) \cdot \exists x \cdot x = y$ must model $farmer(y) \wedge \exists x(x = y)$ in order to succeed.

To smoothen the following derivations, we will define the function $\langle\psi\rangle\phi : DPL \times FOL \rightarrow FOL$ by

$$\alpha \models \langle\psi\rangle\phi \;\leftrightarrow\; \exists\beta(\alpha[\![\psi]\!]\beta \wedge \beta \models \phi) \tag{32}$$

Notice from this definition that $\langle\psi\rangle\top$ is the conversion function from DPL to FOL. Using definition (32) and formulas (17) to (21) we can derive the following

equivalences. See appendix 6.2 for my full proofs of meaning preservation.

$$\langle \exists x \rangle \phi \equiv \exists x(\phi) \tag{33}$$
$$\langle \psi_1 \cdot \psi_2 \rangle \phi \equiv \langle \psi_1 \rangle \langle \psi_2 \rangle \phi \tag{34}$$
$$\langle \psi_1 \cup \psi_2 \rangle \phi \equiv \langle \psi_1 \rangle \phi \vee \langle \psi_2 \rangle \phi \tag{35}$$
$$\langle P(x_1, \ldots, x_n) \rangle \phi \equiv P(x_1, \ldots, x_n) \wedge \phi \tag{36}$$
$$\langle \neg \psi \rangle \phi \equiv \neg \langle \psi \rangle \top \wedge \phi \tag{37}$$
$$\langle \psi_1 \vee \psi_2 \rangle \phi \equiv (\langle \psi_1 \rangle \top \vee \langle \psi_2 \rangle \top) \wedge \phi \tag{38}$$

The simple predicates $\langle \bot \rangle \phi \equiv \bot$ and $\langle \top \rangle \phi \equiv \phi$ follow trivially from (36). Using the first 5 equations, equivalences for derives operators such as $\rightarrow$ and $\vee$ can be derived syntactically as also shown in appendix 6.2.

An alternative way to the $\langle \psi \rangle \phi$ function for looking at the above equations is to think of the $\langle \cdot \rangle$s as elements on a stack. This is indeed how I have implemented the DPL to FOL routine in appendix 6.4.

2.2 Donkey Monoids

With the introduction of DPL we came a long way in terms of a compositional logic. However looking once more at the donkey sentence variation:

> A farmer owns a donkey, if he beats it.

And its DPL formula:

$$\exists x \cdot farmer(x) \cdot \exists y \cdot donkey(y) \cdot beats(x, y) \rightarrow owns(x, y) \tag{39}$$

We see that we still haven't reached the compositional Nirvana. Equation (39) is clearly not based on the composition of 'A farmer owns a donkey' and 'if he beats it' in a direct way.

2.2.1 Polarity Switcher

To help our course, Albert Visser suggests the construction of a special monoid over DPL. The idea is that we are going to simultaneously work on two 'streams' of DPL, and switch easily between them with a special constant $\bowtie$.

The inverse donkey sentence can now be expressed as:

$$\bowtie \cdot \exists x \cdot farmer(x) \cdot \exists y \cdot donkey(y) \cdot \bowtie \cdot owns(x, y) \cdot \bowtie \cdot beats(x, y) \tag{40}$$

which 'internally' is represented by the following streams of information:

$(-)$	$\exists x \cdot farmer(x) \cdot \exists y \cdot donkey(y)$	$beats(x, y)$
$(+)$	$owns(x, y)$	

(41)

Visser interprets the above as the DPL formula $(-) \to (+)$.

Another example from Visser is: 'A man keeps his word. He is honest' which roughly translates into: $\bowtie \cdot \exists x \cdot man(x) \cdot \bowtie \cdot keepword(x) \cdot honest(x)$ for the general reading. Visser has a whole lot more examples in his paper[27].

So how do we define this nifty $\bowtie$? The trick is to move the entire dynamic predicate logic from before into an monoid that will help us keep track of some extra details.

A monoid is an simple algebraic structure with a one associative binary operation and an identity element. Our monoid $\langle \mathcal{D}, \cdot \rangle$ is going to have the domain $\mathcal{D} = DPL \times DPL \times \{+, -\}$, which is populated by:

$$\langle \top, \phi, + \rangle \in \mathcal{D} \quad \text{for all } \phi \in \text{DPL} \tag{42}$$

With no chance of confusion, we will simply denote these elements by their DPL representation. For example $\mathcal{D}$ contains the constants $\top$ and $\bot$ and the $\bowtie$ constant here defined:

$$\top := \langle \top, \top, + \rangle \tag{43}$$
$$\bot := \langle \top, \bot, + \rangle \tag{44}$$
$$\bowtie := \langle \top, \top, - \rangle \tag{45}$$

The associative operation $\cdot$ is as below. To make the definition slicker, we give the set $\{+, -\}$ a multiplication operation $++ \mapsto +$, $+- \mapsto -$, $-+ \mapsto -$, $-- \mapsto +$.

$$\langle q_-, q_+, \alpha \rangle \cdot \langle r_-, r_+, \beta \rangle := \langle q_- \cdot r_{-\alpha},\ \ q_+ \cdot r_{+\alpha},\ \ \alpha\beta \rangle \tag{46}$$

Notice how we keep the negative stream in the first position of the tuple, and the positive stream at the second position. The third position indicates to what stream we are currently writing. For example, if we are writing to the negative stream, and we apply an element that itself asks us to continue at its negative stream, we go back to the positive stream.

2.2.2 Scope Modifier

We won't spend much time with the above monoid though, since we are immediately going to introduce a generalization! We are going to introduce the scope modifier $\triangle$, which allows us to write:

> A farmer owns a donkey. He beats it.

(that is just some single farmer and donkey) as

$$\exists x \cdot farmer(x) \cdot owns(x, y) \cdot \triangle \cdot \exists y \cdot donkey(y) \cdot \triangle \cdot beats(x, y) \tag{47}$$

That is very close to the order of the natural language!

And combining the scope modifier with the polarity switched we can finally, as promised, write our original sentence 'If a farmer owns a donkey, he beats it' as:

$$\bowtie \cdot \exists x \cdot farmer(x) \cdot owns(x,y) \cdot \triangle \cdot \exists y \cdot donkey(y) \cdot \triangle \cdot \bowtie \cdot beats(x,y) \tag{48}$$

In stream form, that is:

$(-,1)$			$\exists y \cdot donkey(y)$	
$(-,0)$	$\exists x \cdot farmer(x)$	$owns(x,y)$		
$(+,1)$				
$(+,0)$				$beats(x,y)$

(49)

With the DPL interpretation being $(-,1) \cdot (-,0) \rightarrow (+,1) \cdot (+,0)$. Notice that the negative relations are again put on the left side of the implication, and the relations with larger scope value are put before those of lower value. We see $\bowtie$ and $\triangle$ are commutative, since they work entirely on their own indicator.

As expected from the above diagram, we are going to define our new monoid $\langle \mathcal{D}, \cdot \rangle$ as having the domain[3] $DPL^4 \times \{+,-\} \times \mathbb{Z}_2$ with the elements:

$$\langle \top, \top, \top, \phi, +, 0 \rangle \in \mathcal{D} \quad \text{for all } \phi \in \text{DPL} \tag{50}$$

In addition to $\bowtie$ and $\triangle$ defined as:

$$\top := \langle \top, \top, \top, \top, +, 0 \rangle \tag{51}$$
$$\bot := \langle \top, \top, \top, \bot, +, 0 \rangle \tag{52}$$
$$\bowtie := \langle \top, \top, \top, \top -, 0 \rangle \tag{53}$$
$$\triangle := \langle \top, \top, \top, \top +, 1 \rangle \tag{54}$$

For $\cdot$ the definition works just like in the previous case, though slightly more complicated:

$$\begin{aligned} &\langle q_{-,1}, q_{-,0}, q_{+,1}, q_{+,0}, \alpha, i \rangle \cdot \langle r_{-,1}, r_{-,0}, r_{+,1}, r_{+,0}, \beta, j \rangle := \\ &\quad \langle q_{-,1} \cdot r_{-\alpha,1+i},\ \ q_{-,0} \cdot r_{-\alpha,i},\ \ q_{+,1} \cdot r_{\alpha,1+i},\ \ q_{+,0} \cdot r_{\alpha,i},\ \ \alpha\beta,\ \ i+j \rangle \end{aligned} \tag{55}$$

In his paper Visser now goes on to define $\triangleleft$ and $\triangleright$ which work similarly to $\bowtie$ except they work 'backwards and forward in time'. This allows us to model 'if he beats it' entirely on its own as $\triangleleft \cdot beats(x,y)$ in

$$\exists x \cdot farmer(x) \cdot owns(x,y) \cdot \triangle \cdot \exists y \cdot donkey(y) \cdot \triangle \cdot \triangleleft \cdot beats(x,y) \tag{56}$$

[3]We have chosen to model our scopes in $\mathbb{Z}_2$. It's an interesting question to consider what sentences might require three or more levels of scoping. If we chose to work on the entire $\mathbb{Z}$, we could define $\triangle$ to go up in scope and $\triangledown$ to go down.

If we immagined a system that tried to build meaning from its input as soon as it came in, $\triangleleft$ would allow us to account for 'surprises' like in the sentence:

> He sees... no donkey

In this case we would compositionally like to start out with $sees(x, y)$ expecting a $\triangle \cdot \exists y \cdot \triangle$ somehow, but when we instead see the negation, we end up with the formula:

$$sees(x, y) \cdot \triangleleft \cdot \bot \cdot \triangleright \cdot \triangle \cdot \exists y \cdot donkey(y) \cdot \triangle \tag{57}$$

Visser suggests $\triangleleft \cdot \bot \cdot \triangleright$ often is a good candidate for words like 'no' and 'nobody'. Unfortunately these retrospective triangles tend to mess sentences enough up, that I had to include brackets everywhere to keep things under control. More brackets are seldom the way to beautiful formulas, so in the automatic framework in the next section, we will settle with the $\bowtie$ and $\triangle$ monoid.

2.3 Montague Semantics

Montague semantics is a theory of natural language that believes its semantics can be described thoroughly in terms of standard mathematical models. Logician Richard Montague (1930-1971) famously wrote

> There is in my opinion no important theoretical difference between natural languages and the artificial languages of logicians; indeed I consider it possible to comprehend the syntax and semantics of both kinds of languages with a single natural and mathematically precise theory. [18]

He believed that just as Chomsky had formulated the syntax of language in a mathematical framework, so too should it be possible to define the meaning of every sentence, based on the meaning of its parts and a set of compositional rules.

While the above remains a hot topic in the philosophy of language, we are going to take a more practical stand. In our quest to automate the translation of language into Visser's DPL monoid, it is interesting that Montague was able to define such a large fragment of English using familiar tools of compositionality and lambda calculus.

The basic idea is very simple. For each POS-tag we define a type. E.g. S has the type $Bool$ and VP has the type $Entity \to Bool$, the intuition being that the VP returns true if the entity 'does what the VP described' and false otherwise. An NP is given type type $VP \to Bool$ which allows it to create a scope for a variable and pass it to the VP, or whatever it sees fitting for that NP.

The point is that we always give a specific phrase type the same logical type. This allows our composition rules to know exactly what type each component has. Here are some more examples:

- Proper noun (John, Oxford): Entity
- One-place predicate constant (farmer, sleeps, walks): $Entity \to Bool$
- Transitive verb (owns, beats): $Entity \to Entity \to Bool$
- Attributive adjective (good, intelligent, former): $(Entity \to Bool) \to (Entity \to Bool)$
- Quantifiers/Determiners (a, the, every, no): $(Entity \to Bool) \to (Entity \to Bool) \to Bool$

To translate a sentence like 'Every farmer beats a donkey' we first create a constituent tree [S [NP [DT Every] [NN farmer]] [VP [VBZ beats] [NP [DT a] [NN donkey]]]].

Next we create objects for the basic elements (a computer would do this recursively, but it is easier for us to do it level by level):

Every, $r_1 =$	$\lambda xy(\bowtie \cdot \triangle \cdot \exists r_1 \cdot (xr_1) \cdot \triangle \cdot \bowtie \cdot (yr_1))$	$:: (E \to B) \to (E \to B) \to B$
farmer $=$	$\lambda x(farmer(x))$	$:: E \to B$
beats $=$	$\lambda xy(beats(xy))$	$:: E \to E \to B$
a, $r_2 =$	$\lambda xy(\triangle \cdot \exists r_2 \cdot (xr_2) \cdot \triangle \cdot (yr_2))$	$:: (E \to B) \to (E \to B) \to B$
donkey $=$	$\lambda x(donkey(x))$	$:: E \to B$
Every farmer $=$	$\lambda y(\bowtie \cdot \triangle \cdot \exists r_1 \cdot farmer(r_1) \cdot \triangle \cdot \bowtie \cdot (yr_1))$	$:: (E \to B) \to B$
a donkey $=$	$\lambda y(\triangle \cdot \exists r_2 \cdot donkey(r_2) \cdot \triangle \cdot (yr_2))$	$:: (E \to B) \to B$
beats a donkey $=$	$\lambda z(\triangle \cdot \exists r_2 \cdot donkey(r) \cdot \triangle \cdot beats(z, r_2))$	$:: E \to B$

Until we can then finally compose the top NP and VP to get the sentence:

$$\begin{aligned} & \bowtie \cdot \triangle \cdot \exists r_1 \cdot farmer(r_1) \cdot \triangle \cdot \bowtie \cdot \triangle \cdot \exists r_2 \cdot donkey(r_2) \cdot \triangle \cdot beats(r_1, r_2) \\ = \; & \bowtie \cdot \triangle \cdot \exists r_1 \cdot farmer(r_1) \cdot \triangle \cdot \bowtie \cdot beats(r_1, r_2) \cdot \triangle \cdot \exists r_2 \cdot donkey(r_2) \cdot \triangle \end{aligned}$$

You might have noticed we didn't discuss how the variables r_1 and r_2 were assigned. This is of course of utter importance in relation to anaphora resolution. However in the section 2.5 we will discuss a way out of this problem.

Later in the implementation section we will look at more issues regarding to using Montague Grammars in practice. First we need to look at a very important source of ambiguity in sentences.

2.3.1 Weak and Strong Readings

'Weak' and 'strong' are quite accepted terms in linguistics for describing different ways to interpret quantifiers. In line with information theory, we can say that weak readings are about weak information and strong readings about strong information.

Makoto Kanazawa has written an entire paper on issues with weak and strong readings of donkey sentences[13]. For our main donkey sentence, we may have at least the following readings of the second 'a':

1. Weak reading: A farmer who owns a donkey beats a donkey he owns.
2. Strong reading: A farmer who owns a donkey beats every donkey he owns
3. Unique reading: A farmer who owns a donkey beats the donkey he owns.

We have equally many readings for the first 'A', creating nine possible interpretations. This is because 'a' is a very ambiguous quantifier. Other quantifiers such as 'every' and 'all' are more clearly defined. 'Some' is another example of an ambiguous quantifier.

Often we can use the context of a sentence to determine which meaning is the correct one. For example:

> Every farmer owns a donkey. It is pink.

Here it is clear that all the farmers share one donkey. Hence perhaps the best solution to the problem is to generate all possible sentences and let the context decide which ones survive.

2.4 Dependency Graphs

Dependency graphs are an alternative way of parsing sentences. Where constituent trees keep the original word order, dependency graphs are allowed to move thing around in any way they like[4]. Also dependency graphs don't have to be trees.

The flexibility of dependency graphs allow them to do a lot of interesting things. In our favourite example of the donkey and the farmer we had the problem that there wasn't a very strong syntactical relationship between 'farmer' and 'he'. In figure 2 we see how the same sentence would look if parsed with a dependency parser.

The first thing we notice is that dependency graphs focus on verbs. The verbs become the heads of the sentences, and everything else becomes the ' subject', 'the

[4]Until now we've been lucky that things next to each other often were related, but in some languages other than English, words can jump anywhere they like, and our approach might fail.

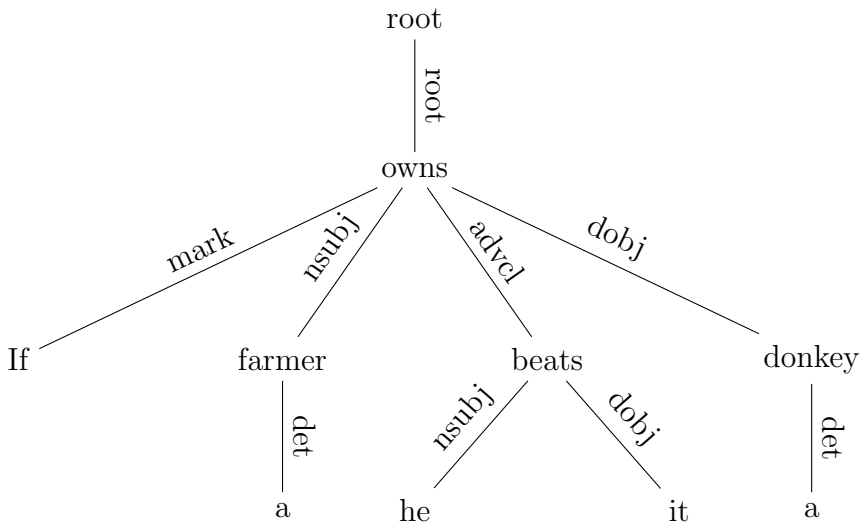


Figure 2: Dependency graph for 'If a farmer owns a donkey, he beats it'

object' or 'the something else' of the verb. In our example 'he beats it' becomes an 'adverbial clause modifier' of the main verb. This means it modifies the verb, here in the sense of making the main verb a condition.

Dependency graphs naturally seems to offer more semantic information than constituent trees, and hence that they would be a good intermediate step towards our goal of logification. However the first disappointment arise as we start to look at how they are created.

The Stanford Parser makes dependency graphs using a special kind of regular expressions for trees called 'tregex'[3]. It defines tregex rules such as:

$$NP < NN \ \$ \ VP \tag{58}$$

Which means an NP that is the parent of an NN and sister of a VP.

Another more complicated example is:

$$NP < (NN < dog) \ \$ \ (VP \ll \#(barks > VBZ)) \tag{59}$$

Which means an NP over an NN over 'dog' and with a sister VP headed by 'barks' which is a verb in present singular.

Obviously this means we can't get any information from the graph that we couldn't also get from the tree. In fact it probably contains less. On the flip side however, the regular expressions are clearly more powerful than simple recursions as we define them in the Montague grammars.

I've worked hard on trying to define a Montague like set of equations for English, but over the dependency graph. This has however failed, mostly due to one simple reason: In a Montague grammar we expect every type of phrase to have a specific logical type. E.g. all NPs are 'VP → Bool' and VPs are 'Entity → Bool'. This makes it fit very nice in a programming context, where we can define a function 'parseNP :: ConstituentTree → (VP → Bool)'. With the dependency approach we don't have this option. A 'subj' arrow can mean many different things. Sometimes it points at a noun, sometimes at a proper noun, sometimes something else.

In conclusion, Montague like grammars for dependency graphs could be very interesting to make. It would however require some well defined rules for what we can expect to find under each type of arrow. This was a blind alley.

2.5 Coreferences

Coreferences (or anaphora) is essential in order to create advanced discourse, since it is what allows us maintain a topic beyond the first mention. Without coreferences we couldn't even express simple logical statements such as transitivity: 'a man's brother's brother is his brother' etc.

Identifying coreferences (or resolving anaphora) is hence an important part of converting natural language into logic. The issue is not touched upon in Visser's paper, which casually sidesteps the problem by assigning variables to all mentioned entities.

For the subtask of resolving simple pronouns a solution could be to work in terms of subjects, objects, indirect objects and so on. However if not earlier, this certainly fails for pronouns separated from their antecedent by sentence boundaries. Also we want to resolve other kinds of coreferences, such as 'John, the farmer, owns a donkey', where 'the farmer' and 'John' refers to the same entity.

It quickly becomes clear that the problem is not only syntactical. In the sentence 'John and his wife had a farm. He took care of the donkeys' we could substitute 'He' with 'She' and to change the referent of the pronoun. In this case we need semantic knowledge of genders to proceed.

In my project I have taken advantage of the coreference tagger built into Stanford CoreNLP[15][16][22]. In the following section I will summarize the workings of this machine, made to work across large texts as well as simple sentences.

Coreference tagging is one part of computational linguistics where machine learning algorithms still haven't been able to outperform large iterative applications of linguistic heuristics. Of course the procedure assumes (most likely) machine learned POS-tagging and named entity recognition, but the main algorithm is transparent and deterministic.

The idea is to iteratively go over the text with different 'sieves'. The below example explains each step with an example I have adopted from [15]:

> John is a farmer. He owned a stubborn donkey. A girl was looking at the donkey. "It was my favorite," John said to her.

Mention detection The first step is to identify all noun phrases (NP) and personal (PRP) and possessive (PRP$) pronouns. This information is easily pulled from the constituent tree. Notice we also identify nested mentions:

$[\text{John}]_1^1$ is $[\text{a farmer}]_2^2$. $[\text{He}]_3^3$ owned $[\text{a stubborn donkey}]_4^4$.
$[\text{A girl}]_5^5$ was looking at $[\text{the donkey}]_6^6$.
"$[\text{It}]_7^7$ was $[[\text{my}]_9^9$ favorite$]_8^8$," $[\text{John}]_{10}^{10}$ said to $[\text{her}]_{11}^{11}$.

All mentions are initially assigned to separate entities (entity ids). The entities are also attached information we can easily pull from the constituent tree, such as 'a girl: number: singular, gender: female' which are used for the following rules.

Speaker Sieve The purpose of this first sieve is to link self referring pronouns to speakers. Speakers are identified simply by their connection to verbs such as 'say' and proximity to the quotations. In our case the pronoun 'my' gets linked to 'John':

$[\text{John}]_1^1$ is $[\text{a farmer}]_2^2$. $[\text{He}]_3^3$ owned $[\text{a stubborn donkey}]_4^4$.
$[\text{A girl}]_5^5$ was looking at $[\text{the donkey}]_6^6$.
"$[\text{It}]_7^7$ was $[[\mathbf{my}]_9^1$ favorite$]_8^8$," $\mathbf{[John]}_{10}^9$ said to $[\text{her}]_{11}^{11}$.

In conversational text we would store information about the speakers in our entities. Thus we could e.g. match mentions of 'you' to the speaker of the previous quote we saw.

We also note restrictions for future use, so we don't risk coreferencing an 'I' with a 'he' etc.

String Match The second step is to merge entities referred to by exactly the same string. In our case we have two mentions of 'John':

$\mathbf{[John]}_1^1$ is $[\text{a farmer}]_2^2$. $[\text{He}]_3^3$ owned $[\text{a stubborn donkey}]_4^4$.
$[\text{A girl}]_5^5$ was looking at $[\text{the donkey}]_6^6$.
"$[\text{It}]_7^7$ was $[[\text{my}]_9^1$ favorite$]_8^8$," $\mathbf{[John]}_{10}^1$ said to $[\text{her}]_{11}^{11}$.

Relaxed String Match This third sieve doesn't get activated by our example. It will try to remove relative clauses and other text following the head word of a mention, and do an exact match of the result. For example [John] and [John, whose donkey loves him] will correctly be merged.

Like the 'String Match' sieve, we are not doing anything fancy here, and we will incorrectly coreference the two mentions of 'John' in: '[John, who loves donkeys] and [John, who hates donkeys] were brothers.'

Precise Constructs The 'Precise Constructs' sieve uses a lot of common patterns that link mentions. In our case two 'X is Y' patterns are found:

[John]$^1_1$ is **[a farmer]**$^1_2$. [He]$^3_3$ owned [a stubborn donkey]$^4_4$.
[A girl]$^5_5$ was looking at [the donkey]$^6_6$.
"**[It]**$^7_7$ was **[**[my]$^1_9$ **favorite]**$^7_8$," [John]$^1_{10}$ said to [her]$^{11}_{11}$.

Other patterns include acronyms and appositives. For example [Canadian Donkey & Mule Association] and [CDMA] are linked, because the second is tagged as an acronym and it matches the upper case letters of the first one. An appositive is usually a pattern 'X, Y, ...' where Y is another description of X.

Strict Head Match A, B, C are simply rules strip mentions down to their head word and try to find exact matches. This is similar to 'Relaxed String Match', but more radical. In our case we correctly get a link between 'a stubborn donkey' and 'the donkey':

[John]$^1_1$ is [a farmer]$^1_2$. [He]$^3_3$ owned **[a stubborn donkey]**$^4_4$.
[A girl]$^5_5$ was looking at **[the donkey]**$^4_6$.
"[It]$^7_7$ was [[my]$^1_9$ favorite]$^7_8$," [John]$^1_{10}$ said to [her]$^{11}_{11}$.

Linking mentions based on their head word is dangerous. e.g. 'University of Oxford' and 'University of Cambridge' both have 'University' as their head word, but are (obviously?) different entities. To combat this a lot of strict rules must be observed. These are however gradually weakened in sieve B and C.

Proper Head Noun Match could be called 'Strict Head Match D'. It is the weakest form of the sieve, as it has only the three constraints: two mentions of the same entity cannot have one included in the other, they cannot contain words referring to conflicting geographical locations, and they cannot have different numbers. e.g. [donkeys] and [at least 5 donkeys].

Pronoun Match This last sieve is perhaps the most important one. It is done in the end so that we may have as much data in our entities as possible for matching. Using guesses on gender, number and animacy we can make our final merges:

[John]$^1_1$ is [a farmer]$^1_2$. **[He]**$^1_3$ owned [a stubborn donkey]$^4_4$.
[A girl]$^5_5$ was looking at **[the donkey]**$^4_6$.
"**[It]**$^4_7$ was [[my]$^1_9$ favorite]$^4_8$," [John]$^1_{10}$ said to **[her]**$^5_{11}$.

As always there are a few constraints put on what can be merged. One rule is that the distance between pronoun and antecedent must be maximum 3 sentences.

After the final step different post processing options are usually performed. For example singleton mentions are often discarded. That would mean we got rid of 'a musician' and 'my favorite'.

The reason the rules are applied in the order above is to give the most certain rules the highest priority.

The Stanford procedure above is currently the strongest competitor in various coreference competitions.[16][22]

3 Implementation

The requirements for the testing framework are very simple: Support as big a chunk of English as possible, while keeping to the intended meaning.

Since this program is going to be used for research by computational linguists, easy access to modify the code is more useful than big user interfaces and parametrization. Our interface is going to be simple text based, with an input file of text you want translated and a command you run on said file to start producing logifications.

3.1 Interfacing with Stanford

The Stanford Parser tools provide a rich Java library for interfacing directly with the system. They have classes for things like constituent trees, dependencies, coreferences and more. Obviously I couldn't use any of that. Instead I ran a simple shell script they include, which outputs the relevant information in XML.

Haskell has a nice XML library in Text.XML.Light. However the constituent found by the parser were not actually in XML, but in a simple plain text format. To parse this I wrote my own parser using the great monadic Parsec library.

The next step for my code was to tag the relevant tree leafs with correct variable names, so it could go into the Montague code. As stated before the Stanford tool chain provided me with coreference tagging, but unfortunately not in a great format for my needs. If a sentence said 'A man and a dog, they had a shower', it would assign the same 'mention' to 'a man' and 'a dog', but nothing to 'a shower'. Luckily Haskell is quite good at such tree manipulation. One thing I made sure was that I always use distinct variables so I never have to perform α-conversion.

One major issue with the Stanford tools is the fact that they are very slow. It uses less than a second per input text, but actually starting the program takes more than two minutes. This is due to the suite loading large compressed files of language statistics into RAM. This issue is the reason my program works with batches of inputs.

3.2 Syntax and Semantics

An interesting issue is the fact that we deal with 4 types of logic. Each having a syntactic and semantic part. That means we have 8 kinds of truth ($\top$), 8 kinds of implication, 8 kinds of everything. And this is with only two Donkey Monoids implemented.

The mentioned types are the following:

```
data Fol = T | F | ...
type IFol = Interpretation → Bool

data Dpl = DT | DF | ...
type IDpl = Interpretation → Assignment → Assignment → Bool
type IDpl2 = Interpretation → Assignment → [Assignment]

data Visser1 = VT | VF | ...
type IVisser1 = (Dpl, Dpl, Integer)
data Visser2 = VT | VF | ...
type IVisser2 = (Dpl, Dpl, Dpl, Dpl, Integer, Integer)
```

Notice that DPL has two possible interpretations[5]. None of these are actually used in the final program, but they serve well to test this middle layer of the program.

For each syntactic type I've defined a number of functions:

'simplify' which takes a formula of the given syntactic type and repeatedly applies a number of identities thought to simplify the logic[6].

'tex' and 'pretty' which prints the formula in either TEXor Unicode format. It's in a way main part of the program's user interface. Hence I have taken extra care to make sure it has visually pleasing bracketing.

'int' which converts the formula to its semantic counterpart. There are also interpretation functions that interpret a logic in the logic 'below' it, hence creating one directional conversion functions.

In terms of the semantic types we have to be careful. Types such as 'Interpretation → Bool' forgets to allow for basic 'errors' such as references to unbound variables. For FOL we can check this statically and not allow the syntax, but for a DPL sentence $(\exists x \cdot P(x) \cup \exists y) \cdot Q(x)$ the existence of x in $Q(x)$ depends on the value of $P(x)$.

One option in case of an error is to simply return $\bot$ and say the formula evaluated to false, but perhaps we want to distinguish the cases. I have experimented with a lot of types, such as 'Interpretation → Maybe Bool', which can return the value 'Nothing' in case of an error. It doesn't make the formulas any nicer though. If we decide to even implement Visser's 'many world' semantics as discussed earlier, we get absolutely terrible formulas that don't honor the simplicity of the logic in any way.

[5]This is more of curiosity due to the fact that a DPL formula is a relation between two assignments, and that an assignment can either be thought of as a set of pairs: $P(D^V \times D^V)$ or a function from type X to a set of type X: $D^V \to P(D^V)$. The first one is nice for proofs, but the second one has a nice and short programmatic definition.

[6]This is extra important because the conversion of Donkey Monoids into DPL and DPL into FOL creates a lot of 'dummy' $\top$s.

In the current version of my program, I have decided to go with the simplest possible type, and simply report errors through Haskell's error system.

3.3 The Grammar

Once I have sculpted the input text into a nice tree shape[7], I parse it to my 'Montague' function. As I described in the early chapters, this applies a set of rules recursively to create a formal interpretation.

Now, since Montague grammars have been around for a long time, and used in commercial machine translating systems from the 70ies and 80ies, you would think the web was would be flooded with code and long machine readable lists of rules I could use to bootstrap my system. This doesn't seem to be the case. Perhaps this is due to people abandoning this approach for vector models before the internet had really picked up the speed of today.

Some people are however wondering if there might still be things to learn from Montague's approach, like Groendijk & Stokhof in Dynamic Montague Grammar[9]:

> '... we are convinced that the capacities of [Montague Grammar] have not been exploited to the limit, that sometimes an analysis is carried out in a rival framework simply because it is more fashionable'

Nonetheless, while that particular paper does have some interesting lists, it is worrying that not even Montague's original papers supply more than a few examples.

This means that nearly all of the rules I shall present are of my own creation. Because these rules are tested in practice on a large set of English sentences, they will hopefully avoid some simple mistakes and oversights that are easy to make in purely theoretical settings. It seems like a good approach to start from a small working fragment, and slowly add more phrase rules and sentences. The opposite of this approach is what is often observed when logicians and philosophers go astray in very abstract corners of English, without having covered the basic ground.

The following describes the set of rules I use in my program:

Let ϕ, ψ, ω be lists of phrase nodes, let α, β, γ leaf nodes. Then we can define a set of interpretation functions $[\![\cdot]\!]_{\text{type}}$ from phrase nodes to logic:

[7] I use the type 'PosTree IWord' where:

```
data PosTree a = P PosTag [PosTree a] | L PosTag a
type IWord = (Word, Index)
```

$$
\begin{aligned}
[\![(_{\mathrm{S}}\phi)\psi]\!]_{\mathrm{ROOT}} &:= [\![\phi]\!]_{\mathrm{S}} \cdot [\![\psi]\!]_{\mathrm{ROOT}} \\
[\![(_{\mathrm{NP}}\phi)\psi]\!]_{\mathrm{ROOT}} &:= [\![\phi]\!]_{\mathrm{NP}}(\lambda x.\top) \cdot [\![\psi]\!]_{\mathrm{ROOT}} \\
\hline
[\![(_{\mathrm{NP}}\phi)(_{\mathrm{VP}}\psi)]\!]_{\mathrm{S}} &:= [\![\phi]\!]_{\mathrm{NP}}[\![\psi]\!]_{\mathrm{VP}} \\
[\![(_{\mathrm{S}}\phi)(_{\mathrm{CC}}\mathrm{and})(_{\mathrm{S}}\psi)]\!]_{\mathrm{S}} &:= [\![\phi]\!]_{\mathrm{S}} \cdot [\![\psi]\!]_{\mathrm{S}} \\
[\![(_{\mathrm{SBAR}}(_{\mathrm{IN}}\mathrm{if})(_{\mathrm{S}}\phi))(_{,}\,,)\psi]\!]_{\mathrm{S}} &:= ?_0(\bowtie \cdot [\![\phi]\!]_{\mathrm{S}} \cdot \bowtie \cdot ?_1([\![\psi]\!]_{\mathrm{S}})) \\
\hline
[\![(_{\mathrm{PRP}}e)]\!]_{\mathrm{NP}} &:= \lambda v.ve \\
&\quad \text{where } e \text{ is a predetermined variable name} \\
[\![(_{\mathrm{DT}}\alpha)(_{\mathrm{NN}}\beta)]\!]_{\mathrm{NP}} &:= \lambda v.[\![\alpha]\!]_{\mathrm{DT}}[\![\beta]\!]_{\mathrm{NN}}v \\
[\![(_{\mathrm{DT}}\alpha)(_{\mathrm{NN}}\beta)(_{\mathrm{CC}}\mathrm{and})(_{\mathrm{NN}}\gamma)]\!]_{\mathrm{NP}} &:= \lambda v.[\![\alpha]\!]_{\mathrm{DT}}(\lambda y.[\![\beta]\!]_{\mathrm{NN}}y \cdot [\![\gamma]\!]_{\mathrm{NN}}y)v \\
[\![(_{\mathrm{NP}}\phi)(_{\mathrm{CC}}\mathrm{and})(_{\mathrm{NP}}\psi)]\!]_{\mathrm{NP}} &:= \lambda v.[\![\phi]\!]_{\mathrm{NP}}(\lambda x.[\![\psi]\!]_{\mathrm{NP}}(\lambda y.\forall z.z = x \vee z = y \to vz)) \\
&\quad \text{where } z \text{ is a globally free variable} \\
\hline
[\![\mathrm{a}, e]\!]_{\mathrm{DT}} := [\![\mathrm{some}, e]\!]_{\mathrm{DT}} &:= \lambda vw.\triangle \cdot \exists e \cdot ve \cdot \triangle \cdot we \\
[\![\mathrm{all}, e]\!]_{\mathrm{DT}} := [\![\mathrm{every}, e]\!]_{\mathrm{DT}} &:= \lambda vw.?_0(\bowtie \cdot \triangle \cdot \exists e \cdot ve \cdot \triangle \cdot \bowtie \cdot we) \\
[\![\mathrm{no}, e]\!]_{\mathrm{DT}} &:= \lambda vw.?_0(\bowtie \cdot \exists e \cdot ve \cdot we \cdot \bowtie \cdot \bot) \\
[\![\mathrm{the}, e]\!]_{\mathrm{DT}} &:= \lambda vw.we \\
\hline
[\![(_{\mathrm{VB}}\alpha)]\!]_{\mathrm{VP}} &:= [\![\alpha]\!]_{\mathrm{VB}} \\
[\![(_{\mathrm{VB}}\alpha)(_{\mathrm{NP}}\phi)]\!]_{\mathrm{VP}} &:= \lambda x.[\![\phi]\!]_{\mathrm{NP}}([\![\alpha]\!]_{\mathrm{TVB}}x) \\
[\![(_{\mathrm{VP}}\phi)(_{\mathrm{CC}}\mathrm{and})(_{\mathrm{VP}}\psi)]\!]_{\mathrm{VP}} &:= \lambda x.[\![\phi]\!]_{\mathrm{VP}}x \cdot [\![\psi]\!]_{\mathrm{VP}}x \\
[\![(_{\mathrm{VB}}\alpha)(_{\mathrm{CC}}\mathrm{and})(_{\mathrm{VB}}\beta)(_{\mathrm{NP}}\phi)]\!]_{\mathrm{VP}} &:= \lambda x.[\![\phi]\!]_{\mathrm{NP}}(\lambda y.[\![\alpha]\!]_{\mathrm{TVB}}xy \cdot [\![\beta]\!]_{\mathrm{TVB}}xy) \\
[\![(_{\mathrm{VB}}\mathrm{do})(_{\mathrm{RB}}\mathrm{not})(_{\mathrm{VP}}\phi)]\!]_{\mathrm{VP}} &:= \lambda x.?_0(\bowtie \cdot [\![\phi]\!]_{\mathrm{VP}}x \cdot \bowtie \cdot \bot)
\end{aligned}
$$

The mentioned 'Montague' function basically passes the tree to $[\![\cdot]\!]_{\mathrm{ROOT}}$ and does a bit of cleaning of the result.

3.4 Machine Learning

Machine Learning is one gaping hole we haven't mentioned in this paper. Obviously something as dynamic as natural language we can never just define a set of rules like we did above. Programming a computer to learn a set of rules like that is unfortunately not something much research has gone into, and so no large banks of input/output pairs exist to learn from.

Another idea for logification using machine learning is to entirely skip the Montague step. The $\bowtie$ and $\triangle$ monoid makes logic look close enough to natural language, that it starts to become tempting utilizing machine learning methods

from translation. Unfortunately this idea runs into the same barrier as the above: that for supervised learning, we first need a large base of already translated sentences.

3.5 Testing

In the previous sections we have tried to be as formal and careful as possible in our derivations and proofs, but as Knuth famously said:

> 'Beware of bugs in the above code; I have only proved it correct, not tried it.'

In the actual implementation of my program I have used the *QuickCheck* and *HUnit* test environments to write unit tests for algebraic properties and actual inputs to my program. To enable QuickTest to test the different preparation algorithms on constituent trees, I implemented a random generator[8].

What remains to test is the validity of the Montague grammar introduced in the previous section. As reasonable as it seems we want to reduce the likelihood of it containing conflicting definitions. In a transformation grammar over natural language we are always running the risk of have been too narrow sighted in its definition. The classic example being '$(_{\mathrm{DT}}\mathrm{a})$' interpreted as $\exists$, but then actually being an $\forall$ if used in an 'if' phrase.

In DPL we don't have that particular problem, but instead we can run into problems with the way the scopes of composed elements affect each other. In Visser logic it is even worse, as $\bowtie$s and $\triangle$s can teleport elements introduced much later in the discourse far around the final statement.

Hence, as reasonable as the introduced grammar seems, we want to reduce the likelihood of conflicts as much as possible. Until now we have been very focused on the same one or two simple sentences, or whatever sentences were relevant to the features we were discussing, but now we are going to introduce a much larger set of complicated test cases. Ideally we want test cases for testing every combination of grammar rules imaginable.

In the test of this program, we used a branch of 53 sentences. Some copied from the papers referenced, others invented based on features considered or the daily life of farmers and donkeys as imagined by the author. The sentences were converted to Visser, DPL and FOL, and the meanings were checked manually to make sense. See appendix 6.3 for the raw program output.

[8] In conclusion QuickTest wasn't as useful as I had hoped, since defining my functions as properties to be tested, required far more code than the functions themselves.

In the following section we will run through the most interesting sentences used, and hopefully make it a bit clearer how the different steps of translation work.

3.5.1 A Bunch of Examples

In this section we translate 12 sentences from English to the Visser logic to DPL and finally to FOL as described in the earlier sections.

The sentences are all simplified through a number of syntactical identities defined for each language. The purpose of this is only to improve readability.

1. Let's start up with a very sequential example to showcase the strength of DPL. Notice how the sentences are easily concatenated with no need for scope handling.

English	A man comes in. He sees a donkey. He smiles.
Visser	$\triangle \cdot \exists a \cdot man(a) \cdot \triangle \cdot come\_in(a) \cdot \triangle \cdot \exists b \cdot donkey(b) \cdot \triangle \cdot see(a,b) \cdot smile(a)$
DPL	$\exists a \cdot man(a) \cdot \exists b \cdot donkey(b) \cdot come\_in(a) \cdot see(a,b) \cdot smile(a)$
FOL	$\exists a(man(a) \wedge \exists b(donkey(b) \wedge come\_in(a) \wedge see(a,b) \wedge smile(a)))$

2. Now let's try a real donkey sentence. Notice how in this case the program chooses the weak reading of 'some donkey'.

English	Every farmer beats some donkey.
Visser	$\bowtie \cdot \triangle \cdot \exists a \cdot farmer(a) \cdot \triangle \cdot \bowtie \cdot \triangle \cdot \exists b \cdot donkey(b) \cdot \triangle \cdot beat(a,b)$
DPL	$\exists a \cdot farmer(a) \rightarrow \exists b \cdot donkey(b) \cdot beat(a,b)$
FOL	$\forall a(farmer(a) \rightarrow \exists b(donkey(b) \wedge beat(a,b)))$

3. We can also handle cases where 'a farmer' is introduced inside the antecedent of an 'if'.

English	If a farmer owns a donkey, he beats it
Visser	$\bowtie \cdot \triangle \cdot \exists a \cdot farmer(a) \cdot \exists b \cdot donkey(b) \cdot \triangle \cdot own(a,b) \cdot \bowtie \cdot beat(a,b)$
DPL	$\exists a \cdot farmer(a) \cdot \exists b \cdot donkey(b) \cdot own(a,b) \rightarrow beat(a,b)$
FOL	$\forall a(farmer(a) \rightarrow \forall b(donkey(b) \wedge own(a,b) \rightarrow beat(a,b)))$

4. This is an example from Visser [27] containing sentences with no verbs. Notice how the Visser logic has to introduce $?_0$ to prevent the antecedent from becoming part of the consequent.

English	A farmer. A Donkey. If he owns it, it is beaten.
Visser	$\triangle \cdot \exists a \cdot farmer(a) \cdot \exists b \cdot donkey(b) \cdot \triangle \cdot ?_0(\bowtie \cdot own(a,b) \cdot \bowtie \cdot beat(b))$
DPL	$\exists a \cdot farmer(a) \cdot \exists b \cdot donkey(b) \cdot (own(a,b) \rightarrow beat(b))$
FOL	$\exists a(farmer(a) \wedge \exists b(donkey(b) \wedge (own(a,b) \rightarrow beat(b))))$

$?_0$ however only scopes off the consequent 'channel' of the monoid. Hence this more complicated example also works as expected.

English	If a farmer owns a donkey, he beats it. If he rewards it, he doesn't own it.
Visser	$?_0(\bowtie \cdot \triangle \cdot \exists a \cdot farmer(a) \cdot \exists b \cdot donkey(b) \cdot \triangle \cdot own(a,b) \cdot \bowtie \cdot beat(a,b))$ $\cdot ?_0(\bowtie \cdot reward(a,b) \cdot \bowtie \cdot ?_0(\bowtie \cdot own(a,b) \cdot \bowtie \cdot \bot))$
DPL	$\exists a \cdot farmer(a) \cdot \exists b \cdot donkey(b) \rightarrow$ $(own(a,b) \rightarrow beat(a,b)) \cdot (reward(a,b) \rightarrow \neg own(a,b))$
FOL	$\forall a(farmer(a) \rightarrow \forall b(donkey(b) \rightarrow$ $(own(a,b) \rightarrow beat(a,b)) \wedge (reward(a,b) \rightarrow \neg own(a,b))))$

5. A big consequence of/decision in the grammar created, is that no new entities can be introduced in the 'then part' of an 'if'. The reason for this is that such objects are not guaranteed to exist on the ground of the sentence. We enforce this rule using the $?_1$ function.

English	If a farmer owns a donkey, he owns a horse. If he doesn't own it, he owns a cow.
Visser	$?_0(\bowtie \cdot \triangle \cdot \exists a \cdot farmer(a) \cdot \exists b \cdot donkey(b) \cdot \triangle \cdot own(a,b) \cdot \bowtie$ $\cdot ?_1(\triangle \cdot \exists c \cdot horse(c) \cdot \triangle \cdot own(a,c))) \cdot$ $?_0(\bowtie \cdot ?_0(\bowtie \cdot own(a,b) \cdot \bowtie \cdot \bot) \cdot \bowtie \cdot ?_1(\triangle \cdot \exists d \cdot cow(d) \cdot \triangle \cdot own(a,d)))$
DPL	$\exists a \cdot farmer(a) \cdot \exists b \cdot donkey(b) \rightarrow (own(a,b) \rightarrow \exists c \cdot horse(c) \cdot own(a,c))$ $\cdot (\neg own(a,b) \rightarrow \exists d \cdot cow(d) \cdot own(a,d))$
FOL	$\forall a(farmer(a) \rightarrow \forall b(donkey(b) \rightarrow (own(a,b) \rightarrow \exists c(horse(c) \wedge own(a,c)))$ $\wedge (\neg own(a,b) \rightarrow \exists d(cow(d) \wedge own(a,d)))))$

6. Because of the doubtful semantic meaning of disjunctive sentences discussed, I have instead chosen to focus on conjunctive problems. And there are a lot of those too. The first one is a case of a conjunctive VP. Notice that we treat 'starves' as a transitive verb, even though the sentence is clearly ambiguous. In this case the phrase parser makes the decision for us.

English	A farmer starves a horse and beats a donkey.
Visser	$\triangle \cdot \exists a \cdot farmer(a) \cdot \exists b \cdot horse(b) \cdot \triangle \cdot starve(a,b)$
	$\cdot \triangle \cdot \exists c \cdot donkey(c) \cdot \triangle \cdot beat(a,c)$
DPL	$\exists a \cdot farmer(a) \cdot \exists b \cdot horse(b) \cdot \exists c \cdot donkey(c) \cdot starve(a,b) \cdot beat(a,c)$
FOL	$\exists a(farmer(a) \wedge \exists b(horse(b) \wedge \exists c(donkey(c) \wedge starve(a,b) \wedge beat(a,c))))$

7. In Montague's PTQ[19] he doesn't give rules for conjunction of NPs, but only sentences and VP's. In [24] and [20] Mats Rooth and Barbara Partee discuss different issues arising, but never reach any usable construction. Personally I've found the below to be very useful for conjunctive NPs that act as a union.

English	All donkeys and a horse sing a song.
Visser	$\bowtie \cdot \triangle \cdot \exists a \cdot donkey(a) \cdot \triangle \cdot \bowtie \cdot \triangle \cdot \exists b \cdot horse(b) \cdot \triangle$
	$\cdot ?_0(\bowtie \cdot \exists c \cdot eq\_on\_of(c,a,b) \cdot \bowtie \cdot \triangle \cdot \exists d \cdot song(d) \cdot \triangle \cdot sing(c,d))$
DPL	$\exists a \cdot donkey(a) \rightarrow \exists b \cdot horse(b) \cdot \exists d \cdot song(d)$
	$\cdot (\exists c \cdot eq\_on\_of(c,a,b) \rightarrow sing(c,d))$
FOL	$\forall a(donkey(a) \rightarrow \exists b(horse(b) \wedge \exists d(song(d)$
	$\wedge \forall c(eq\_on\_of(c,a,b) \rightarrow sing(c,d)))))$

8. This is a more simple example of a conjunctive NN.

English	Some farmer and entrepreneur owns a donkey.
Visser	$\triangle \cdot \exists a \cdot farmer(a) \cdot entrepreneur(a) \cdot \exists b \cdot donkey(b) \cdot \triangle \cdot own(a,b)$
DPL	$\exists a \cdot farmer(a) \cdot entrepreneur(a) \cdot \exists b \cdot donkey(b) \cdot own(a,b)$
FOL	$\exists a(farmer(a) \wedge entrepreneur(a) \wedge \exists b(donkey(b) \wedge own(a,b)))$

9. The great thing about the construct introduced for conjunctive NPs is that it also works well when we need a cross product of subjects and objects, as

this example shows.

English	A man and a woman owned a horse and a donkey.
	If the donkey did not walk, and the man was a farmer, he beat it.
Visser	$\triangle \cdot \exists a \cdot man(a) \cdot \exists b \cdot woman(b) \cdot \triangle \cdot ?_0(\bowtie \cdot \exists c \cdot eq\_on\_of(c,a,b) \cdot \bowtie$
	$\cdot \triangle \cdot \exists d \cdot horse(d) \cdot \exists e \cdot donkey(e) \cdot \triangle \cdot ?_0(\bowtie \cdot \exists f \cdot eq\_on\_of(f,d,e) \cdot \bowtie$
	$\cdot own(c,f))) \cdot ?_0(\bowtie \cdot ?_0(\bowtie \cdot walk(e) \cdot \bowtie \cdot \bot) \cdot a\_farmer(a) \cdot \bowtie \cdot beat(a,e))$
DPL	$\exists a \cdot man(a) \cdot \exists b \cdot woman(b) \cdot \exists d \cdot horse(d) \cdot \exists e \cdot donkey(e)$
	$\cdot (\exists c \cdot eq\_on\_of(c,a,b) \cdot \exists f \cdot eq\_on\_of(f,d,e) \rightarrow own(c,f))$
	$\cdot (\neg walk(e) \cdot a\_farmer(a) \rightarrow beat(a,e))$
FOL	$\exists a(man(a) \wedge \exists b(woman(b) \wedge \exists d(horse(d) \wedge \exists e(donkey(e)$
	$\wedge \forall c(eq\_on\_of(c,a,b) \rightarrow \forall f(eq\_on\_of(f,d,e) \rightarrow own(c,f)))$
	$\wedge (\neg walk(e) \wedge a\_farmer(a) \rightarrow beat(a,e))))))$

10. Finally it is interesting to see how the program handles pronouns. I've chosen to make 'free pronouns' not be an error, but a feature. Free pronouns are treated as free variables, which allow you to create sentences that might be use e.g. to filter a database.

English	She is a mother of five
Visser	$a\_mother\_of\_five(a)$
DPL	$a\_mother\_of\_five(a)$
FOL	$a\_mother\_of\_five(a)$

11. The following sentence is very similar to the original donkey sentences, but it features a free variable.

English	If a farmer owns it and he beats it, it is a donkey
Visser	$\bowtie \cdot \triangle \cdot \exists a \cdot farmer(a) \cdot \triangle \cdot own(a,b) \cdot beat(a,b) \cdot \bowtie \cdot a\_donkey(b)$
DPL	$\exists a \cdot farmer(a) \rightarrow (own(a,b) \cdot beat(a,b) \rightarrow a\_donkey(b))$
FOL	$\forall a(farmer(a) \wedge own(a,b) \wedge beat(a,b) \rightarrow a\_donkey(b))$

12. A final example that really shows how DPL scope and free variables all play along together is this the case of a 'non existent' antecedent. The Stanford

Coreference tagger mistakenly tags the two men in this sentence as the same, but because negation creates a closed scope, the final sentence still comes out right.

English	No man walks in the park. He yodels.
Visser	$?_0(\bowtie \cdot \exists a \cdot man(a) \cdot walk\_in\_the\_park(a) \cdot \bowtie \cdot \bot) \cdot yodel(a)$
DPL	$\neg(\exists a \cdot man(a) \cdot walk\_in\_the\_park(a)) \cdot yodel(a)$
FOL	$\forall a(man(a) \rightarrow \neg walk\_in\_the\_park(a)) \wedge yodel(a)$

This concludes the implementation part of the paper.

4 Conclusions

I have successfully developed a testing framework that lets researchers define logics and interpretation functions and test them on large inputs of interesting text. I have discussed many possible directions I could have taken, I have proved every result involved, and I have done extensive testing.

My program can correctly logify entire paragraphs of text containing advanced anaphora, conjunctive nouns, verbs and adjectives, implications and scoping. In some cases the my defined rules create quite creative FOL formulas that would likely have taken a person a few tries to get right. In addition to factual sentences my program also handles query type sentences with free variables.

My fragment of English is however quite limited to the extent that most sentences require some adaption to pass through. Word classes such as adverbs and plurals are not supported, and in some cases my framework supports a sentence, but logification still fails due to shortcomings in the Stanford toolkit. For example in 'Every dog sees a cat. It chases it.' where the anaphora doesn't get resolved, and 'Buffalo buffalo Buffalo buffalo buffalo buffalo Buffalo buffalo'[23] that also manages to slightly confuse the parser.

My tests of Visser's Donkey Monoids show that they can be used for automatic logification, but that they lose some of their charm. Mainly it seems that the $\bowtie$ and $\triangle$ switches are a bit too strong, and hence require extensive use of the scoping function $?(\cdot)$. At least the rules I came up with essentially uses Donkey Monoids as if they were DPL.

My tests also show that the Montague grammar can be made very simple when implemented using dynamic logics. I couldn't however entire eliminate the need for lambda calculus, but perhaps a more refined Donkey Monoid will be up for the challenge.

4.1 Future work

I had a lot of issues with the Stanford Parser not being able to parse sentences correctly. In every paper I have read, this part always 'just works', but I didn't have any options, but to discard the sentence. It would be interesting to make a fault tolerant logification system. Perhaps some knowledge from the semantics could be passed back to the syntax parser for better results.

It would also be interesting to return more than one result in the case of ambiguity. We have seen how weak and strong readings of the actors in the sentences can give very different results. Also conjunction of transitive verbs like 'A farmer starves and beats a donkey' are inherently ambiguous. Perhaps if we returned many possible logifications for each sentence, we could take the intersection to get

the real meaning.

In terms of expanding the scope of our translations, the fruits I have identified are hanging at different heights:

At the lowest level there are a lot of trivial additions. Support in more cases for more negatives like 'no' and 'nobody' hasn't been explored well. Possessive pronouns is something I've struggled with, but if we accept that 'He took his jacket' translates to $\exists y \cdot owns(x, y) \cdot jacket(y) \cdot took(x, y)$ it shouldn't be hard. Also there are things like 'existential there' and more kinds of SBARs like started by 'then', 'whom' etc. that we can add, but for the expense of additional complicated rules.

Higher up are things like adverbs. This will require a more complicated treatment of predicates to hopefully allow something like $furiously(beat(x, y))$. How exactly these things can be logified is still an open question in linguistics. Another fruit at this level is model verbs. Sentences like 'It rains. It might rain' and 'It might rain. It rains' should be treated differently. For this we could look into the field of modal logics.

Even higher up are lots of classical linguistic problems. Sentences such as 'The golden mountain does not exist', 'The ancient Greeks thought the morning star was the evening star' have had entire papers written on them, without any definite answers. Probably we'll have to consider the use case to know how a well defined computational logic should work. Another thing at the top level is novel words and sentence structures. These will clearly require some sort of machine learning.

Finally I have mentioned that we might be able to expand the Donkey Monoids with more functionality as we sees fit during Montague rule making. The possibility for pushing control elements into the semantic of the logic is really the beauty of Visser's approach and unfortunately a road I haven't studied much in this paper. It is not hard to imagine $(Dpl, Dpl, Dpl, Dpl, Integer, Integer)$ exchanged for something like $(Dpl, \ldots, PosTag)$ and corresponding composition rules.

5 Acknowledgments

I would like to thank Samson Abramsky for introducing me to the world of Natural Language parsing and semantics and for helpful and stimulating discussions.

Also thanks to Phil Blunsom and Edward Grefenstette for exciting and enlightening lectures into the classical and modern approaches.

Thanks to the Natural Language Processing Group at Stanford University for making their software and results easily available for use in my project[14].

Thanks to Érica Baena Kato for providing interesting linguistic perspectives on many issues.

Finally I wish to thank The Donkey Sanctuary[26] for taking donkey's welfare seriously and rescuing neglected donkeys in the UK and all over the world.

6 Appendix

6.1 The Problem with Disjunction

A major problem with our use of the union of relations as disjunction is the way it allows seemingly sound sentences to combine into syntactically invalid ones. Consider the sentence 'Either a farmer owns a donkey or he doesn't'. We would write that as

$$\exists x \cdot farmer(x) \cdot (\exists y \cdot owns(x, y) \cdot donkey(y) \vee \bot) \tag{60}$$

Now concatenating this sentence with 'he beats it': $beat(x, y)$ would give us a problem. In fact this problem would be more than a syntactic problem, because before we know the model and the existence of the donkey, we don't know if the sentence is valid or not.

Visser[27] suggests that we should build our logic on sets of relations read disjunctively instead of just relations. This 'possible world' scenario would transform $(60)\cdot beats(x, y)$ into $\{\exists x \cdot farmer(x) \cdot \exists y \cdot donkey(y) \cdot owns(x, y) \cdot beats(x, y), \exists x \cdot farmer(x) \cdot \bot \cdot beats(x, y)\} = \{\exists x \cdot farmer(x) \cdot \exists y \cdot donkey(y) \cdot owns(x, y) \cdot beats(x, y), \bot\}$. This clearly is always well defined.

Visser's idea doesn't fully save us though. If we now consider 'a farmer either owns a donkey or owns a horse':

$$\begin{aligned} \exists x \cdot farmer(x) \cdot (\exists y \cdot owns(x, y) \cdot donkey(y) \\ \vee \exists z \cdot owns(x, z) \cdot horse(z)) \end{aligned} \tag{61}$$

And combines that with 'He beats the donkey': $beat(x, y)$, we get, in Visser's model, $\{\exists x \cdot farmer(x) \cdot \exists y \cdot owns(x, y) \cdot donkey(y) \cdot beats(x, y), \exists x \cdot farmer(x) \cdot \exists z \cdot owns(x, z) \cdot horse(z) \cdot beats(x, y)\}$. The second one is clearly not valid (though we wouldn't get a problem if there is no horse or farmer) which we might catch 'at compile time' and replace it with something like '$\bot$' or 'error'. This sort of error handling is discussed and implemented in the implementation section, but is not terribly interesting from a formal point of view.

Much more interesting are the following two questions: How could we have translated so it would actually work? After all it seemed like a reasonable sentence until variable naming got us. And does the way variable binding works with our definition actually make sense?

Clearly just renaming z to y in the horse term would have solved the first problem. However to work in a compositional way, that would either require great luck, or renaming, and we don't have a renaming operator.

A trick we could consider is to declare an 'object' variable outside of the disjunction:

$$\exists s \cdot farmer(s) \cdot \exists o \cdot (\exists y \cdot owns(s,y) \cdot donkey(y) \cdot o = y \\ \vee \exists z \cdot owns(s,z) \cdot horse(z) \cdot o = z) \cdot beats(s,o) \tag{62}$$

This even inspires on idea that we might be able to build everything up around subjects, objects, indirect objects etc. This is explored in the implementation, but it does suffer from many problems. E.g. we might well have a sentence very similar to the above: 'A farmer owns a donkey and owns a horse. He beat them'. Here we suddenly have two objects in the same sentence.

Sentences like the above are easy to make in English because the absence of noun genders make it easy to use the same pronoun to refer to a lot of things at once. By choosing the right actors however, it is always possible to run into problems.

The problem is perhaps the non-linearity in the way variables pass through the disjunction. Somehow we lose track of their history when the 'results' are merged together.

This leads on to the other question we mentioned: Is the binding flow through our disjunction operator really any good? We are is inspired by the sentence 'Either a farmer or his scapegoat must go to court. In any case he will be picked up by sunset'. In terms of DPL formulas, this would be modelled something like $(\psi_1 \cup \psi_2) \cdot \psi_3$. Clearly we have a problem here: Variables bound in ψ_1 are not accessible in ψ_2.

This sentence suggests that we might want to define a disjunction operator with a linear flow of variables: $\psi_1 \to \psi_2 \to \psi_3$ instead of $\psi_1 \to \psi_3$ and $\psi_2 \to \psi_3$. However there is a problem: We currently define 'false' as the absence of possible assignments. Hence if ψ_1 is false, there is no way we can define disjunction to push its variables into ψ_2. If any variables were left undefined, ψ_1 wouldn't be false.

There is a cure to this problem. When we extend our logic with more power in the monoid chapter, we get a second chance. However let's first have a look at an obvious alternative definition for disjunction considered by Groenendijk[8]. We'll call it 'static disjunction':

$$\begin{aligned} \alpha[\psi_1 \vee \psi_2]\beta &:\equiv \alpha[\neg(\neg\psi_1 \wedge \neg\psi_2)]\beta && (63) \\ &\leftrightarrow \alpha[\neg\psi_1 \to \psi_2]\beta \\ &\leftrightarrow \alpha = \beta \wedge \forall\gamma(\alpha[\neg\psi_1]\gamma \to \gamma \models [\psi_2]) \\ &\leftrightarrow \alpha = \beta \wedge \forall\gamma((\alpha = \gamma \wedge \gamma \not\models \psi_1) \to \gamma \models [\psi_2]) \\ &\leftrightarrow \alpha = \beta \wedge \forall\gamma(\alpha = \gamma \to (\gamma \models \psi_1 \vee \gamma \models \psi_2)) \\ &\leftrightarrow \alpha = \beta \wedge (\alpha \models \psi_1 \vee \alpha \models \psi_2) && (64) \end{aligned}$$

The idea to this follows naturally from De Morgan's laws of classical logic. Clearly the semantics are different from the ones we defined for $\cup$. However not in a good way. This static disjunction doesn't allow any variables to escape from inside its reach.

So does this fix our problem of binding between clauses? No. In fact this definition doesn't even allow binding from a clause and out. The villain is $\neg\psi$. From the first mention we defined it to be static so it wouldn't allow unsafe, arbitrary assignments as long as they didn't satisfy ψ. We wanted our logic to be reasonably monotonic, and we got what we deserved. $\neg$ not only closed down this version of disjunction, but our implication definition as well.

We can imagine defining $\neg$ dynamically: $\neg\psi_1 \cdot \psi_2 = \neg(\psi_1 \cdot \psi_2)$. However not in our current model, and it won't solve any of our problems with disjunction. It does however allow this sentence: 'It is not the case that a farmer doesn't own a donkey. He beats it'. While it made sense not to allow variables to escape from a single $\neg$, we want them to escape from a double.

6.2 Derivation of DPL Preconditions

Remember that we have defined, for assignments α and β, DPL formula ψ and FOL formula ϕ: $\alpha \models \langle\psi\rangle\phi$ to mean $\exists\beta(\alpha[\![\psi]\!]\beta \wedge \beta \models \phi)$. $\alpha \models \phi$ is just the usual $\models$ from FOL.

Since the proof is over the syntax of DPL, we use $[\![\cdot]\!]$ as the interpretation function so that $\alpha[\![\psi]\!]\beta$ is the FOL statement '$(\alpha, \beta) \in \Psi$', where Ψ is the relation defined by ψ.

Let's start out with the most tricky derivation. How does the resetting, unbounded $\exists$ of DPL translate into the defining, bounded $\exists$ of FOL?

$$
\begin{aligned}
\alpha \models \langle\exists x\rangle\phi &\leftrightarrow \exists\beta(\alpha[\![\exists x]\!]\beta \wedge \beta \models \phi) \\
&\text{This is the definition from the DPL chapter:} \\
&\leftrightarrow \exists\beta(\forall\omega(\omega \neq x \rightarrow \alpha_\omega = \beta_\omega) \wedge \beta \models \phi) \\
&\text{The trick is this switch to assignments:} \\
&\leftrightarrow \exists\beta(\exists\nu(\forall\omega(\alpha[x := \nu]_\omega = \beta_\omega)) \wedge \beta \models \phi) \\
&\leftrightarrow \exists\beta(\exists\nu(\alpha[x := \nu] = \beta) \wedge \beta \models \phi) \\
&\leftrightarrow \exists\nu(\exists\beta(\alpha[x := \nu] = \beta \wedge \beta \models \phi)) \\
&\text{Here we apply the } \models \text{ definition:} \\
&\leftrightarrow \exists\nu(\alpha[x := \nu] \models \phi) \\
&\leftrightarrow \alpha \models \exists x(\phi) \qquad (33)
\end{aligned}
$$

Where we have taken $\alpha[x := \nu]$ to mean the α with α_x assigned to the value of ν. The derivation for $\cdot$ goes more smoothly:

$$\begin{aligned}
\alpha \models \langle \psi_1 \cdot \psi_2 \rangle \phi &\leftrightarrow \exists \beta (\alpha [\![\psi_1 \cdot \psi_2]\!] \beta \wedge \beta \models \phi) \\
&\leftrightarrow \exists \beta (\exists \gamma (\alpha [\![\psi_1]\!] \gamma \wedge \gamma [\![\psi_2]\!] \beta) \wedge \beta \models \phi) \\
&\leftrightarrow \exists \gamma (\alpha [\![\psi_1]\!] \gamma \wedge \exists \beta (\gamma [\![\psi_2]\!] \beta \wedge \beta \models \phi)) \\
&\leftrightarrow \exists \gamma (\alpha [\![\psi_1]\!] \gamma \wedge \gamma \models \langle \psi_2 \rangle \phi) \\
&\leftrightarrow \alpha \models \langle \psi_1 \rangle \langle \psi_2 \rangle \phi
\end{aligned} \tag{34}$$

$\cup$ interestingly doesn't translate to $\langle \psi_1 \vee \psi_2 \rangle \phi$ as we might have expected:

$$\begin{aligned}
\alpha \models \langle \psi_1 \cup \psi_2 \rangle \phi &\leftrightarrow \exists \beta (\alpha [\![\psi_1 \cup \psi_2]\!] \beta \wedge \beta \models \phi) \\
&\leftrightarrow \exists \beta ((\alpha [\![\psi_1]\!] \beta \vee \alpha [\![\psi_2]\!] \beta) \wedge \beta \models \phi) \\
&\leftrightarrow \exists \beta (\alpha [\![\psi_1]\!] \beta \wedge \beta \models \phi) \vee \exists \beta (\alpha [\![\psi_2]\!] \beta \wedge \beta \models \phi) \\
&\leftrightarrow (\alpha \models \langle \psi_1 \rangle \phi) \vee (\alpha \models \langle \psi_2 \rangle \phi) \\
&\leftrightarrow \alpha \models \langle \psi_1 \rangle \phi \vee \langle \psi_2 \rangle \phi
\end{aligned} \tag{35}$$

For predicates it's mostly a matter of juggling variables, values and assignments:

$$\begin{aligned}
\alpha \models \langle P(x_1, \ldots, x_n) \rangle \phi &\leftrightarrow \exists \beta (\alpha [\![P(x_1, \ldots, x_n)]\!] \beta \wedge \beta \models \phi) \\
&\leftrightarrow \exists \beta (\alpha = \beta \wedge P(\alpha_{x_1}, \ldots, \alpha_{x_n}) \wedge \beta \models \phi) \\
&\leftrightarrow P(\alpha_{x_1}, \ldots, \alpha_{x_n}) \wedge \alpha \models \phi \\
&\leftrightarrow \alpha \models P(x_1, \ldots, x_n) \wedge \phi
\end{aligned} \tag{36}$$

$\neg$ is interesting because it has to create a scope:

$$\begin{aligned}
\alpha \models \langle \neg \psi \rangle \phi &\leftrightarrow \exists \beta (\alpha [\![\neg \psi]\!] \beta \wedge \beta \models \phi) \\
&\leftrightarrow \exists \beta (\alpha = \beta \wedge \neg \exists \gamma (\alpha [\![\psi]\!] \gamma) \wedge \beta \models \phi) \\
&\leftrightarrow \neg \exists \gamma (\alpha [\![\psi]\!] \gamma) \wedge \alpha \models \phi \\
&\leftrightarrow \neg \exists \gamma (\alpha [\![\psi]\!] \gamma \wedge \gamma \models \top) \wedge \alpha \models \phi \\
&\leftrightarrow \alpha \not\models \langle \psi \rangle \top \wedge \alpha \models \phi \\
&\leftrightarrow \alpha \models \neg \langle \psi \rangle \top \wedge \phi
\end{aligned} \tag{37}$$

Because ψ is this way 'evaluated on $\top$ only', we know that no variables introduced in ψ can affect ϕ. The derivation of $\vee$ can be done from the previous results alone:

$$\begin{aligned}
\langle \psi_1 \vee \psi_2 \rangle \phi &\equiv \langle \neg(\neg\psi_1 \cdot \neg\psi_2) \rangle \phi && \text{by (63)} \\
&\equiv \neg\langle \neg\psi_1 \cdot \neg\psi_2 \rangle \top \wedge \phi && \text{by (37)} \\
&\equiv \neg\langle \neg\psi_1 \rangle \langle \neg\psi_2 \rangle \top \wedge \phi && \text{by (34)} \\
&\equiv \neg\langle \neg\psi_1 \rangle (\neg\langle \psi_2 \rangle \top) \wedge \phi && \text{by (37)} \\
&\equiv \neg(\neg\langle \psi_1 \rangle \top \wedge \neg\langle \psi_2 \rangle \top) \wedge \phi && \text{by (37)} \\
&\equiv (\langle \psi_1 \rangle \top \vee \langle \psi_2 \rangle \top) \wedge \phi && \text{by (63)}
\end{aligned} \tag{38}$$

6.3 Sample Output

```
$ ./run.sh inputs
Running Stanford Parser...
java -mx3g -cp "../stanford-corenlp-full-2012-11-12/*" edu.stanford.nlp.pipeline.
    StanfordCoreNLP -outputDirectory /tmp -filelist /tmp/donkey_inputlist
Searching for resource: StanfordCoreNLP.properties
Searching for resource: edu/stanford/nlp/pipeline/StanfordCoreNLP.properties
Adding annotator tokenize
Adding annotator ssplit
Adding annotator pos
Loading default properties from tagger edu/stanford/nlp/models/pos-tagger/english-
    left3words/english-left3words-distsim.tagger
Reading POS tagger model from edu/stanford/nlp/models/pos-tagger/english-left3words/
    english-left3words-distsim.tagger ... done [6.1 sec].
Adding annotator lemma
Adding annotator ner
Loading classifier from edu/stanford/nlp/models/ner/english.all.3class.distsim.crf.
    ser.gz ... done [13.3 sec].
Loading classifier from edu/stanford/nlp/models/ner/english.muc.7class.distsim.crf.
    ser.gz ... done [11.6 sec].
Loading classifier from edu/stanford/nlp/models/ner/english.conll.4class.distsim.crf.
    ser.gz ... done [19.1 sec].
Initialization JollyDayHoliday for sutime
Reading TokensRegex rules from edu/stanford/nlp/models/sutime/defs.sutime.txt
Reading TokensRegex rules from edu/stanford/nlp/models/sutime/english.sutime.txt
May 18, 2013 10:48:01 PM edu.stanford.nlp.ling.tokensregex.CoreMapExpressionExtractor
     appendRules
INFO: Ignoring inactive rule: temporal-composite-8:ranges
Reading TokensRegex rules from edu/stanford/nlp/models/sutime/english.holidays.sutime
    .txt
Adding annotator parse
Loading parser from serialized file edu/stanford/nlp/models/lexparser/englishPCFG.ser
    .gz ... done [3.1 sec].
Adding annotator dcoref

Processing file /tmp/donkey_input0 ... (writing to /tmp/donkey_input0.xml) [2.959
    seconds]
Processing file /tmp/donkey_input1 ... (writing to /tmp/donkey_input1.xml) [0.268
    seconds]
```

```
Processing file /tmp/donkey_input2 ... (writing to /tmp/donkey_input2.xml) [1.216
    seconds]
Processing file /tmp/donkey_input3 ... (writing to /tmp/donkey_input3.xml) [0.431
    seconds]
Processing file /tmp/donkey_input4 ... (writing to /tmp/donkey_input4.xml) [0.994
    seconds]
Processing file /tmp/donkey_input5 ... (writing to /tmp/donkey_input5.xml) [0.640
    seconds]
Processing file /tmp/donkey_input6 ... (writing to /tmp/donkey_input6.xml) [1.576
    seconds]
Processing file /tmp/donkey_input7 ... (writing to /tmp/donkey_input7.xml) [0.965
    seconds]
Processing file /tmp/donkey_input8 ... (writing to /tmp/donkey_input8.xml) [0.774
    seconds]
Processing file /tmp/donkey_input9 ... (writing to /tmp/donkey_input9.xml) [1.216
    seconds]
Processing file /tmp/donkey_input10 ... (writing to /tmp/donkey_input10.xml) [6.559
    seconds]
Processing file /tmp/donkey_input11 ... (writing to /tmp/donkey_input11.xml) [0.652
    seconds]
Processing file /tmp/donkey_input12 ... (writing to /tmp/donkey_input12.xml) [0.450
    seconds]
Processing file /tmp/donkey_input13 ... (writing to /tmp/donkey_input13.xml) [0.133
    seconds]
Processing file /tmp/donkey_input14 ... (writing to /tmp/donkey_input14.xml) [0.176
    seconds]
Processing file /tmp/donkey_input15 ... (writing to /tmp/donkey_input15.xml)
Processing file /tmp/donkey_input16 ... (writing to /tmp/donkey_input16.xml) [0.491
    seconds]
Processing file /tmp/donkey_input17 ... (writing to /tmp/donkey_input17.xml) [0.144
    seconds]
Processing file /tmp/donkey_input18 ... (writing to /tmp/donkey_input18.xml) [0.240
    seconds]
Processing file /tmp/donkey_input19 ... (writing to /tmp/donkey_input19.xml) [0.156
    seconds]
Processing file /tmp/donkey_input20 ... (writing to /tmp/donkey_input20.xml) [0.230
    seconds]
Processing file /tmp/donkey_input21 ... (writing to /tmp/donkey_input21.xml)
Processing file /tmp/donkey_input22 ... (writing to /tmp/donkey_input22.xml) [0.286
    seconds]
Processing file /tmp/donkey_input23 ... (writing to /tmp/donkey_input23.xml) [0.331
    seconds]
Processing file /tmp/donkey_input24 ... (writing to /tmp/donkey_input24.xml) [0.110
    seconds]
Processing file /tmp/donkey_input25 ... (writing to /tmp/donkey_input25.xml) [0.236
    seconds]
Processing file /tmp/donkey_input26 ... (writing to /tmp/donkey_input26.xml)
Processing file /tmp/donkey_input27 ... (writing to /tmp/donkey_input27.xml) [0.153
    seconds]
Processing file /tmp/donkey_input28 ... (writing to /tmp/donkey_input28.xml) [0.177
    seconds]
Processing file /tmp/donkey_input29 ... (writing to /tmp/donkey_input29.xml) [0.168
    seconds]
Processing file /tmp/donkey_input30 ... (writing to /tmp/donkey_input30.xml) [0.121
    seconds]
Processing file /tmp/donkey_input31 ... (writing to /tmp/donkey_input31.xml) [0.176
    seconds]
```

```
Processing file /tmp/donkey_input32 ... (writing to /tmp/donkey_input32.xml) [0.111
    seconds]
Processing file /tmp/donkey_input33 ... (writing to /tmp/donkey_input33.xml) [0.205
    seconds]
Processing file /tmp/donkey_input34 ... (writing to /tmp/donkey_input34.xml)
Processing file /tmp/donkey_input35 ... (writing to /tmp/donkey_input35.xml)
Processing file /tmp/donkey_input36 ... (writing to /tmp/donkey_input36.xml)
Processing file /tmp/donkey_input37 ... (writing to /tmp/donkey_input37.xml) [0.160
    seconds]
Processing file /tmp/donkey_input38 ... (writing to /tmp/donkey_input38.xml)
Processing file /tmp/donkey_input39 ... (writing to /tmp/donkey_input39.xml)
Processing file /tmp/donkey_input40 ... (writing to /tmp/donkey_input40.xml) [0.184
    seconds]
Processing file /tmp/donkey_input41 ... (writing to /tmp/donkey_input41.xml) [0.121
    seconds]
Processing file /tmp/donkey_input42 ... (writing to /tmp/donkey_input42.xml) [0.156
    seconds]
Processing file /tmp/donkey_input43 ... (writing to /tmp/donkey_input43.xml) [0.120
    seconds]
Processing file /tmp/donkey_input44 ... (writing to /tmp/donkey_input44.xml) [0.356
    seconds]
Processing file /tmp/donkey_input45 ... (writing to /tmp/donkey_input45.xml) [0.172
    seconds]
Processing file /tmp/donkey_input46 ... (writing to /tmp/donkey_input46.xml)
Processing file /tmp/donkey_input47 ... (writing to /tmp/donkey_input47.xml) [0.104
    seconds]
Processing file /tmp/donkey_input48 ... (writing to /tmp/donkey_input48.xml)
Processing file /tmp/donkey_input49 ... (writing to /tmp/donkey_input49.xml) [0.131
    seconds]
Processing file /tmp/donkey_input50 ... (writing to /tmp/donkey_input50.xml)
Processing file /tmp/donkey_input51 ... (writing to /tmp/donkey_input51.xml)
Processing file /tmp/donkey_input52 ... (writing to /tmp/donkey_input52.xml) [0.201
    seconds]
Processing file /tmp/donkey_input53 ... (writing to /tmp/donkey_input53.xml)
Annotation pipeline timing information:
PTBTokenizerAnnotator: 0.2 sec.
WordsToSentencesAnnotator: 0.0 sec.
POSTaggerAnnotator: 0.4 sec.
MorphaAnnotator: 0.7 sec.
NERCombinerAnnotator: 7.4 sec.
ParserAnnotator: 14.0 sec.
DeterministicCorefAnnotator: 1.5 sec.
TOTAL: 24.1 sec. for 599 tokens at 24.9 tokens/sec.
Pipeline setup: 56.5 sec.
Total time for StanfordCoreNLP pipeline: 81.5 sec.
Exit code: ExitSuccess
If a man eats a sausage, he is happy.
?0(⋈ ·△·∃a·man(a)·△·△·∃b·sausage(b)·△·eat(a,b)·⋈ ·?1(happy(a)))
∃a·man(a)·∃b·sausage(b)⇒(eat(a,b)⇒(⊤⇒happy(a)))
∀a(man(a)⇒∀b(sausage(b)∧eat(a,b)⇒happy(a)))

Every person has a pet.
?0(⋈ ·△·∃a·person(a)·△·⋈ ·△·∃b·pet(b)·△·have(a,b))
∃a·person(a)⇒∃b·pet(b)·(⊤⇒have(a,b))
∀a(person(a)⇒∃b(pet(b)∧have(a,b)))

Every man supports every woman.
```

?0(⋈ ·△·∃a·man(a)·△·⋈ ·?0(⋈ ·△·∃b·woman(b)·△·⋈ ·support(a,b)))
∃a·man(a)·∃b·woman(b)⇒(⊤⇒support(a,b))
∀a(man(a)⇒∀b(woman(b)⇒support(a,b)))

Every man supports no woman.
?0(⋈ ·△·∃a·man(a)·△·⋈ ·?0(⋈ ·∃b·woman(b)·support(a,b)·⋈ ·⊥))
∃a·man(a)⇒(⊤⇒¬(∃b·woman(b)·support(a,b)))
∀a(man(a)⇒∀b(woman(b)⇒¬support(a,b)))

Every farmer beats some donkey.
?0(⋈ ·△·∃a·farmer(a)·△·⋈ ·△·∃b·donkey(b)·△·beat(a,b))
∃a·farmer(a)⇒∃b·donkey(b)·(⊤⇒beat(a,b))
∀a(farmer(a)⇒∃b(donkey(b)∧beat(a,b)))

If a farmer owns a donkey, he beats it
?0(⋈ ·△·∃a·farmer(a)·△·△·∃b·donkey(b)·△·own(a,b)·⋈ ·?1(beat(a,b)))
∃a·farmer(a)·∃b·donkey(b)⇒(own(a,b)⇒(⊤⇒beat(a,b)))
∀a(farmer(a)⇒∀b(donkey(b)∧own(a,b)⇒beat(a,b)))

A man comes in. He sees a donkey. He smiles.
△·∃a·man(a)·△·come-in(a)·△·∃b·donkey(b)·△·see(a,b)·smile(a)
⊤⇒∃a·man(a)·∃b·donkey(b)·come-in(a)·see(a,b)·smile(a)
∃a(man(a)∧∃b(donkey(b)∧come-in(a)∧see(a,b)∧smile(a)))

A farmer. A Donkey. If he owns it, he beats it.
△·∃a·farmer(a)·△·△·∃b·donkey(b)·△·?0(⋈ ·own(a,b)·⋈ ·?1(beat(a,b)))
⊤⇒∃a·farmer(a)·∃b·donkey(b)·(own(a,b)⇒(⊤⇒beat(a,b)))
∃a(farmer(a)∧∃b(donkey(b)∧(own(a,b)⇒beat(a,b))))

A dog barks. If it is beaten, it whines.
△·∃a·dog(a)·△·bark(a)·?0(⋈ ·beat(b)·⋈ ·?1(whine(b)))
⊤⇒∃a·dog(a)·bark(a)·(beat(b)⇒(⊤⇒whine(b)))
∃a(dog(a)∧bark(a)∧(beat(b)⇒whine(b)))

If a farmer owns a donkey, he beats it. If he rewards it, he doesn't own it.
?0(⋈ ·△·∃a·farmer(a)·△·△·∃b·donkey(b)·△·own(a,b)·⋈ ·?1(beat(a,b)))·?0(⋈ ·reward(a,b)·
⋈ ·?1(?0(⋈ ·own(a,b)·⋈ ·⊥)))
∃a·farmer(a)·∃b·donkey(b)⇒(own(a,b)⇒(⊤⇒beat(a,b)))·(reward(a,b)⇒(⊤⇒¬own(a,b)))
∀a(farmer(a)⇒∀b(donkey(b)⇒(own(a,b)⇒beat(a,b))∧(reward(a,b)⇒¬own(a,b))))

If a farmer owns a donkey, he owns a horse. If he doesn't own it, he owns a cow.
?0(⋈ ·△·∃a·farmer(a)·△·△·∃b·donkey(b)·△·own(a,b)·⋈ ·?1(△·∃c·horse(c)·△·own(a,c)))·?0(
⋈ ·?0(⋈ ·own(a,b)·⋈ ·⊥)·⋈ ·?1(△·∃d·cow(d)·△·own(a,d)))
∃a·farmer(a)·∃b·donkey(b)⇒(own(a,b)⇒(⊤⇒∃c·horse(c)·own(a,c)))·(¬own(a,b)⇒(⊤⇒∃d·cow(d)
·own(a,d)))
∀a(farmer(a)⇒∀b(donkey(b)⇒(own(a,b)⇒∃c(horse(c)∧own(a,c)))∧(¬own(a,b)⇒∃d(cow(d)∧own(a
,d)))))

If a woman is American, she loves a soldier. If she is Dutch, she loves a bike-rider.
?0(⋈ ·△·∃a·woman(a)·△·american(a)·⋈ ·?1(△·∃b·soldier(b)·△·love(a,b)))·?0(⋈ ·dutch(a)·
⋈ ·?1(△·∃c·bike-rider(c)·△·love(a,c)))
∃a·woman(a)⇒(american(a)⇒(⊤⇒∃b·soldier(b)·love(a,b)))·(dutch(a)⇒(⊤⇒∃c·bike-rider(c)·
love(a,c)))
∀a(woman(a)⇒(american(a)⇒∃b(soldier(b)∧love(a,b)))∧(dutch(a)⇒∃c(bike-rider(c)∧love(a,
c))))

A farmer owns a donkey. It is brown and he beats it every day.

```
△·∃a·farmer(a)·△·△·∃b·donkey(b)·△·own(a,b)·brown(b)·beat-every-day(a,b)
⊤⇒∃a·farmer(a)·∃b·donkey(b)·own(a,b)·brown(b)·beat-every-day(a,b)
∃a(farmer(a)∧∃b(donkey(b)∧own(a,b)∧brown(b)∧beat-every-day(a,b)))

A farmer starves a horse and beats a donkey.
△·∃a·farmer(a)·△·△·∃b·horse(b)·△·starve(a,b)·△·∃c·donkey(c)·△·beat(a,c)
⊤⇒∃a·farmer(a)·∃b·horse(b)·∃c·donkey(c)·starve(a,b)·beat(a,c)
∃a(farmer(a)∧∃b(horse(b)∧∃c(donkey(c)∧starve(a,b)∧beat(a,c))))

All donkeys and a horse sing a song.
?0(⋈ ·△·∃a·donkey(a)·△·⋈ ·△·∃b·horse(b)·△·?0(⋈ ·∃c·eq-on-of(c,a,b)·⋈ ·△·∃d·song(d)·△·
    sing(c,d)))
∃a·donkey(a)⇒∃b·horse(b)·∃d·song(d)·(⊤⇒(∃c·eq-on-of(c,a,b)⇒sing(c,d)))
∀a(donkey(a)⇒∃b(horse(b)∧∃d(song(d)∧∀c(eq-on-of(c,a,b)⇒sing(c,d)))))

Some farmer and entrepreneur beats a donkey.
△·∃a·farmer(a)·entrepreneur(a)·△·△·∃b·donkey(b)·△·beat(a,b)
⊤⇒∃a·farmer(a)·entrepreneur(a)·∃b·donkey(b)·beat(a,b)
∃a(farmer(a)∧entrepreneur(a)∧∃b(donkey(b)∧beat(a,b)))

A man and a woman owned a horse and a donkey. If the donkey did not walk, and the man
     was a farmer, he beat it.
△·∃a·man(a)·△·△·∃b·woman(b)·△·?0(⋈ ·∃c·eq-on-of(c,a,b)·⋈ ·△·∃d·horse(d)·△·△·∃e·donkey
    (e)·△·?0(⋈ ·∃f·eq-on-of(f,d,e)·⋈ ·own(c,f)))·?0(⋈ ·?0(⋈ ·walk(e)·⋈ ·⊥)·a-farmer(a
    )·⋈ ·?1(beat(a,e)))
⊤⇒∃a·man(a)·∃b·woman(b)·∃d·horse(d)·∃e·donkey(e)·(∃c·eq-on-of(c,a,b)⇒(∃f·eq-on-of(f,d
    ,e)⇒own(c,f)))·(¬walk(e)·a-farmer(a)⇒(⊤⇒beat(a,e)))
∃a(man(a)∧∃b(woman(b)∧∃d(horse(d)∧∃e(donkey(e)∧∀c(eq-on-of(c,a,b)⇒∀f(eq-on-of(f,d,e)⇒
    own(c,f)))∧(¬walk(e)∧a-farmer(a)⇒beat(a,e))))))

She is a mother of five
a-mother-of-five(a)
⊤⇒a-mother-of-five(a)
a-mother-of-five(a)

If a farmer owns it and he beats it, it is a donkey
?0(⋈ ·△·∃a·farmer(a)·△·own(a,b)·beat(a,b)·⋈ ·?1(a-donkey(b)))
∃a·farmer(a)⇒(own(a,b)·beat(a,b)⇒(⊤⇒a-donkey(b)))
∀a(farmer(a)∧own(a,b)∧beat(a,b)⇒a-donkey(b))

No man walks in the park. He yodels.
?0(⋈ ·∃a·man(a)·walk-in-the-park(a)·⋈ ·⊥)·yodel(a)
⊤⇒¬(∃a·man(a)·walk-in-the-park(a))·yodel(a)
∀a(man(a)⇒¬walk-in-the-park(a))∧yodel(a)

A farmer beats a donkey, if he owns it.
Main.hs: Unsuported by intVP: [L "VB" ("beat",(0,2)),P "NP" [L "DT" ("a",(0,3)),L "NN
    " ("donkey",(0,4))],L "," (",",(0,5)),P "SBAR" [L "IN" ("if",(0,6)),P "S" [P "NP"
     [L "PRP" ("he",(0,7))],P "VP" [L "VB" ("own",(0,8)),P "NP" [L "PRP" ("it",(0,9))
    ]]]]]
```

6.4 Source Code

An online repository of the source code can be found at[1].

Listing 1: run.sh

```
#!/bin/bash
cd "$(dirname $0)/src"
runghc Main.hs $1 $2
```

Listing 2: src/Main.hs

```
module Main (main) where

import Stanford (run, runDry)
import ToLogic (montague)
import qualified Visser as V
import qualified Dpl as D
import qualified Fol as F

import Control.Monad
import Data.List
import Data.Maybe
import System.Environment
import System.IO

-- Reads and returns every line from a file, modulo comments and blanks
readSentences :: FilePath → IO [String]
readSentences path = do
                handle ← openFile path ReadMode
                contents ← hGetContents handle
                return [trimComments l | l ← lines contents, (not.null) (trimComments l)]

-- This trims a string for tabs and spaces. Also remove comment suffixes
trimComments :: String → String
trimComments = reverse.dropWhile isSpace.reverse . dropWhile isSpace . takeWhile (≠ '#')
        where
                isSpace ' '  = True
                isSpace 'λt' = True
                isSpace _    = False

main :: IO ()
main = do
                args ← getArgs
                if null args then error "Usage: donkey inputfile [--dry]" else return ()
                sentences ← readSentences (head args)
                putStrLn "Running Stanford Parser..."
                model ← (if args == ["--dry"] then runDry else run) sentences
                sequence [do {
                        putStrLn s;
                        putStrLn $ (V.pretty . V.simplify) p;
                        putStrLn $ (D.pretty . D.simplify . V.intVisser3) p;
                        putStrLn $ (F.pretty . F.simplify . D.intDpl3 . V.intVisser3) p;
                        putStrLn ""
                } | (s, p) ← zip sentences (map montague model)]
                return ()
```

Listing 3: src/Stanford.hs

```
module Stanford (Store, IWord, run, runDry, runOnFile, PosTree(..), postrees, variables) where
```

```
import Fol (Ref)

import Text.ParserCombinators.Parsec
import Data.Map hiding (map, (\\), mapMaybe, null)

import Data.List hiding (union, insert, lookup)
import Text.XML.Light
import Control.Monad
import Data.Maybe
import System.Process

type Index = (Int, Int)
type Word = String
type IWord = (Word, Index)
type Sentence = [Word]

type PosTag = String
data PosTree a = P PosTag [PosTree a] | L PosTag a deriving (Show)

data DepTree = Dep Int [(Ref, DepTree)] deriving (Show, Eq)

type Store = Index → Ref

-----------------

-- lemmas are standardized forms of words. E.g. did → do, n't → not
lemmas :: Element → [Sentence]
lemmas doc = map lemma (findElements (unqual "sentence") sentences)
        where
                Just sentences = findElement (unqual "sentences") doc
                lemma doc' = map strContent (findElements (unqual "lemma") doc')

-----------------

-- Extracts the constituent tree from a document.
postrees :: Element → [PosTree Word]
postrees doc = map rootfilter roots
        where
                parses = findElements (unqual "parse") doc
                roots = map (postreeRoot . strContent) parses
                rootfilter (P "ROOT" [n]) = n
                rootfilter root = error ("Cant␣parse␣strange␣sentence␣" ++ show root)

postreeRoot :: String → PosTree Word
postreeRoot dat = tree
        where
                Right tree = parse posEither "(unknown)" dat

posEither =
        do
                char '('
                res ← try posTree <|> posLeaf
                char ')'
                return res
posTree =
        do
                tag ← many (noneOf "␣")
```

```
                char ' '
                subs ← sepBy1 posEither (char ' ')
                return (P tag subs)
posLeaf =
        do
                tag ← many (noneOf "␣")
                char ' '
                word ← many (noneOf "␣()")
                return (L tag word)

-- Takes a postree and a lemmaized list of words for each sentence
-- and produces a cleaned up and index tagged tree
cleanTrees :: [PosTree Word] → [Sentence] → [PosTree IWord]
cleanTrees trees ls = map cleanElements $ map cleanTags $ improveTrees trees ls

-- Tags leafs with indicies
improveTrees :: [PosTree Word] → [Sentence] → [PosTree IWord]
improveTrees trees ls = [improve t l s | (t, l, s) ← zip3 trees ls [0..]]
        where improve t l s = substitute t (zip l (zip (repeat s) [0..]))

-- simplifies certain tags with tedious information
cleanTags :: PosTree a → PosTree a
cleanTags = treemap f id
        where
                f "VBD" = "VB"
                f "VBG" = "VB"
                f "VBN" = "VB"
                f "VBP" = "VB"
                f "VBZ" = "VB"
                f "NNS" = "NN"
                f other = other

-- deletes certain nodes with tedious information
cleanElements :: PosTree a → PosTree a
cleanElements tree = fromJust (deleteTag "." tree)

-- Maybe this is not useful afterall
rearrangeCCs :: PosTree a → PosTree a
rearrangeCCs (L t w)              = L t w
rearrangeCCs (P t xs) | null bs   = P t ccxs
                                  | otherwise = P t [P "CC" [P t as, P t (tail bs)]]
                      -- | otherwise = P ("CC"++t) [P t as, P t (tail bs)]
        where
                ccxs = map rearrangeCCs xs
                (as, bs) = break f ccxs
                f (L t _) = t == "CC"
                f (P _ _) = False


-- deletes all nodes with a given tag
deleteTag :: PosTag → PosTree a → Maybe (PosTree a)
deleteTag t1 (P t2 xs) | t1 == t2  = Nothing
                       | otherwise = Just (P t2 (mapMaybe (deleteTag t1) xs))
deleteTag t1 (L t2 w)  | t1 == t2  = Nothing
                       | otherwise = Just (L t2 w)

-- map over tags and leaf values
```

```
treemap :: (PosTag → PosTag) → (b → c) → PosTree b → PosTree c
treemap f g (L tag word) = L (f tag) (g word)
treemap f g (P tag subs) = P (f tag) (map (treemap f g) subs)

-- Replaces every leaf value with a value from a list, in order
substitute :: PosTree a → [b] → PosTree b
substitute tree list = snd (substitute_ tree list)
        where
                substitute_ (L tag x) (l:ls)  = (ls, L tag l)
                substitute_ (P tag []) ls     = (ls, P tag [])
                substitute_ (P tag (t:ts)) ls = (ls'', P tag (s:ss))
                        where (ls', s)       = substitute_ t ls
                              (ls'', P _ ss) = substitute_ (P "" ts) ls'

------------------

-- Extract the deptree for each sentence
deps :: Element → String → [DepTree]
deps doc name = map (buildTree deps) roots
        where
                deps = listDeps doc name
                lefts = nub [gov | (typ, gov, dep) ← deps]
                rights = nub [dep | (typ, gov, dep) ← deps]
                roots = lefts \\ rights

-- The dependency information comes in tuples of (label, arrow_from, arrow_to)
-- This function transforms a set of tuples into an algebraic tree structure
buildTree :: [(String, Int, Int)] → Int → DepTree
buildTree deps root = Dep root [(name, buildTree deps dep) | (name, gov, dep) ← deps, gov == root]

-- Extract dependecy tuples form xml
listDeps :: Element → String → [(String, Int, Int)]
listDeps doc name = map parseDep deps
        where
                Just dep = findElement (unqual name) doc
                deps = findElements (unqual "dep") dep

-- Extract single tuple from 'dep' element
parseDep :: Element → (String, Int, Int)
parseDep e = (typ, read gov - 1, read dep - 1)
        where
                Just typ = findAttr (unqual "type") e
                Just gov = findChild (unqual "governor") e >>= (findAttr (unqual "idx"))
                Just dep = findChild (unqual "dependent") e >>= (findAttr (unqual "idx"))

------------------

-- Calls 'corefs' to extract a map from word-index to variable name.
-- In case the index has no assigned variable, a globally free variable is returned
allRefs :: Element → [Ref] → Store
allRefs doc vs = searcher
        where (vs', comap) = corefs doc vs
              searcher i = findWithDefault (vs' !! diagonal i) i comap
              diagonal (j,k) = (j+k)*(j+k+1) `div` 2 + j

-- Builds a map from word-index to variable name for a single coreference element.
-- Additionally returns a list of unused variable names.
```

```
corefs :: Element → [Ref] → ([Ref], Map Index Ref)
corefs doc vs = (vs', unions [singleton i v | (men, v) ← zip ments vs,
                                              indices ← map parseMention men,
                                              i ← indices])
        where
                corefs = findElements (unqual "coreference") doc>>=elChildren
                ments = map (findElements (unqual "mention")) corefs
                vs' = drop (length corefs) vs

-- Extracts indicies governed by a single mention element
parseMention :: Element → [Index]
parseMention men = [(toint sentence-1, i) | i ← [toint start-1..toint end-2]]
        where
                Just sentence = findChild (unqual "sentence") men
                Just start = findChild (unqual "start") men
                Just end = findChild (unqual "end") men
                toint = read . strContent

--------------------

-- An infinite list of possible variable names
variables :: [Ref]
variables = alphabet ++ [b++a | b ← variables, a ← alphabet]
        where alphabet = [c:"" | c ← "abcdefghijklmnopqrstuvwxyz"]

-- A list of tempoary filenames for the produced xml files
filenames = ["/tmp/donkey_input" ++ show i | i ← [0..]]

-- Runs the Stanford toolchain on each of a list of Strings.
run :: [String] → IO [(Store, [PosTree IWord])]
run sentences =
        do
                sequence (zipWith writeFileLn names sentences)
                writeFileLn listname (intercalate "\n" names)
                (_, _, _, h) ← createProcess (shell (stanfordpath++" -outputDirectory /tmp -filelist "
                    ++listname))
                exitCode ← waitForProcess h
                putStrLn ("Exit code: " ++ show exitCode)
                sequence [runOnFile (name ++ ".xml") | name ← names]
        where
                stanfordpath = "../stanford-corenlp-full-2012-11-12/corenlp.sh"
                listname = "/tmp/donkey_inputlist"
                names = take (length sentences) filenames
                writeFileLn path s = writeFile path (s ++ "\n")

-- Like 'run', but doesn't actually call the Stanford toolchain.
-- It is assumed the previous tempoary xml files still eixst
runDry :: [String] → IO [(Store, [PosTree IWord])]
runDry sentences =
        do
                sequence [runOnFile (name ++ ".xml") | name ← names]
        where
                names = take (length sentences) filenames

-- Extracts the usable information from a single generated xml file
runOnFile :: FilePath → IO (Store, [PosTree IWord])
runOnFile name =
```

```
        do
                f ← readFile name
                case parseXMLDoc f of
                        Nothing → error ("Unable to parse XML " ++ name)
                        Just xml → do
                                return (
                                        allRefs xml variables,
                                        cleanTrees (postrees xml) (lemmas xml))
```

Listing 4: src/ToLogic.hs

```
module ToLogic (montague) where

import Fol (Ref)
import Visser (Visser(..), cleanVariables)
import Stanford (IWord, Store, PosTree(..))

import Data.List
import Data.Map hiding (lookup, mapMaybe, map)
import Data.Maybe

type Predicate = Ref → Visser
type Predicate2 = Ref → Ref → Visser

-- Runs a montague grammar transformation on a constituent tree and a
-- word-index to variable name map. The produced logic is a
-- polarity-switch + scope-modifier donkey monoid
montague :: (Store, [PosTree IWord]) → Visser
montague (refs, tree) = cleanVariables $ intRoot tree refs


intRoot :: [PosTree IWord] → Store → Visser
intRoot ((P "S" s):ss) r = intS s r `VC` intRoot ss r
intRoot ((P "NP" np):ss) r = intNP np r (λx → VT) `VC` intRoot ss r
intRoot [] r = VT

intS :: [PosTree IWord] → Store → Visser
intS [P "NP" np, P "VP" vp] r = intNP np r (intVP vp r)
intS ((P "SBAR" [(L "IN" ("if",_)), (P "S" s1)]) : (L "," _) : s2) r =
              VQ0 (VSw `VC` intS s1 r `VC` VSw `VC` VQ1 (intS s2 r))
intS [P "S" s1, L "CC" ("and",_), P "S" s2] r = intS s1 r `VC` intS s2 r
intS [P "S" s1, L "," _, L "CC" ("and",_), P "S" s2] r = intS s1 r `VC` intS s2 r
intS other r = error ("Unsuported by intS: " ++ show other)

intNP :: [PosTree IWord] → Store → Predicate → Visser
intNP [L "PRP" (prp,i)] r vb = vb (r i)
intNP [L "DT" dt, L "NN" nn] r vb = intDT dt r (intNN nn r) vb
intNP [L "DT" dt, L "NN" nn1, L "CC" ("and",_), L "NN" nn2] r vb = intDT dt r (λx → intNN nn1 r x `VC
     ` intNN nn2 r x) vb
intNP [P "NP" np1, L "CC" ("and",_), P "NP" np2] r vb = intNP np1 r (λx → intNP np2 r (inner x))
        where inner x y = VQ0 (VSw `VC` VE (x++y) `VC` VP "eq-on-of" [(x++y),x,y] `VC` VSw `VC` vb (x++y)
             ) -- hopefully x++y is free
intNP other r vb = error ("Unsuported by intNP: " ++ show other)

intVP :: [PosTree IWord] → Store → Predicate
intVP [L "VB" ("be",_), what] r = λv → VP (stringify [what]) [v]
intVP [L "VB" vb] r = intVB vb r
```

```
intVP [L "VB" vb, P "NP" np] r = λv → intNP np r (intVB2 vb r v) -- it could be nice to 'swap'
     subject and object here, to make it read more like the original sentence
intVP [P "VP" vp1, L "CC" ("and",_), P "VP" vp2] r = λv → intVP vp1 r v `VC` intVP vp2 r v
intVP [L "VB" vb1, L "CC" ("and",_), L "VB" vb2, P "NP" np] r = λv1 → (intNP np r (λv2 → (intVB2 vb1
      r v1 v2) `VC` (intVB2 vb2 r v1 v2)))
intVP n@[L "VB" vb, P "PP" pp] r = intVB (stringify n,(0,0)) r
intVP [L "VB" (vb,i), P "NP" np, P "NP-TMP" nptmp] r = intVP [L "VB" (vb++"-"++stringify nptmp,i), P "NP
     " np] r
intVP [L "VB" ("do",_), L "RB" ("not",_), P "VP" vp] r = λv → VQ0 (VSw `VC` intVP vp r v `VC` VSw `VC
     ` VF)
intVP other r = error ("Unsuported by intVP: " ++ show other)

-- Stanford parser doesn't seperate transitive verbs, so our type has to accept any number of
     arguments
-- Hvis de to n;ste ikke bliver mere komplicerede kan vi ogs[ bare proppe dem ind i VP ovenfor.
intVB :: IWord → Store → Predicate
intVB (vb,_) r = λv → VP vb [v]
intVB2 :: IWord → Store → Predicate2
intVB2 (vb,_) r = λv1 v2 → VP vb [v1, v2]

intNN :: IWord → Store → Predicate
intNN (nn,_) r = λv → VP nn [v]

intDT :: IWord → Store → Predicate → Predicate → Visser
intDT ("some", i) r = λp q → VSc `VC` VE m `VC` p m `VC` VSc `VC` q m where m = r i
intDT ("a", i) r = intDT ("some", i) r
intDT ("the", i) r = λp q → q m where m = r i
intDT ("every", i) r = λp q → VQ0 (VSw `VC` VSc `VC` VE m `VC` p m `VC` VSc `VC` VSw `VC` q m) where
     m = r i
intDT ("all", i) r = intDT ("every", i) r
intDT ("no", i) r = λp q → VQ0 (VSw `VC` VE m `VC` p m `VC` q m `VC` VSw `VC` VF) where m = r i
intDT other r = error ("Unsuported by intDT: " ++ show other)

-- Future:

-- Adjectives:
-- (NP (JJ big) (NN sausage))
-- What about adjective phrases? ADJP
-- (ADJP (RB very) (JJ big))

-- Also:
-- ADVP - Adverb Phrase.
-- CONJP - Conjunction Phrase
-- PP - Prepositional Phrase.
-- Different WH phrases

flatten :: PosTree a → [a]
flatten (P _ xs) = xs >>= flatten
flatten (L _ x) = [x]

stringify :: [PosTree IWord] → String
stringify ts = intercalate "-" words
    where words = map fst (ts >>= flatten)
```

Listing 5: src/Dpl.hs

```
module Dpl (Dpl(..), tex, pretty, simplify, Assignment, intDpl, intDpl2, intDpl3) where
```

```
import Fol (Ref, Sigma, Fol(..), Entity, Domain, Predicate)

import Data.List (intersperse)
import Utils (wrap, replace)
import Control.Monad (>=>)

-------------------------------------------------------
-- Syntax

data Dpl =
    DT | DF
  | DC Dpl Dpl
  | DD Dpl Dpl
  | DN Dpl
  | DI Dpl Dpl
  | DE Ref
  | DP Ref [Ref]
  deriving (Eq, Show)

-- Printing functions

tex :: Dpl → String
tex = fst . tex' where
  tex' DT = ("\\top", 0)
  tex' DF = ("\\bot", 0)
  tex' (DC p q) = (wrap 1 (tex' p) ++ "␣\\cdot␣" ++ wrap 1 (tex' q), 1)
  tex' (DD p q) = (wrap 2 (tex' p) ++ "␣\\vee␣" ++ wrap 2 (tex' q), 2)
  tex' (DI p q) = (wrap 2 (tex' p) ++ "␣\\rightarrow␣" ++ wrap 2 (tex' q), 3)
  tex' (DN p) = ("\\neg␣" ++ wrap 0 (tex' p), 0)
  tex' (DE k) = ("\\exists␣" ++ k, 0)
  tex' (DP f ks) = (replace "-" "\\_" f ++ "(" ++ concat (intersperse "," ks) ++ ")", 0)

pretty :: Dpl → String
pretty = fst . pretty' where
  pretty' DT = ("⊤", 0)
  pretty' DF = ("⊥", 0)
  pretty' (DC p q) = (wrap 1 (pretty' p) ++ "·" ++ wrap 1 (pretty' q), 1)
  pretty' (DD p q) = (wrap 2 (pretty' p) ++ "∨" ++ wrap 2 (pretty' q), 2)
  pretty' (DI p q) = (wrap 2 (pretty' p) ++ "⇒" ++ wrap 2 (pretty' q), 3)
  pretty' (DN p) = ("¬" ++ wrap 0 (pretty' p), 0)
  pretty' (DE k) = ("∃" ++ k, 0)
  pretty' (DP f ks) = (f ++ "(" ++ concat (intersperse "," ks) ++ ")", 0)


simplify :: Dpl → Dpl
simplify p = iterate simp p !! 10

simp :: Dpl → Dpl
simp (DC p DT) = simp p
simp (DC DT p) = simp p
simp (DC p DF) = DF
simp (DC DF p) = DF
simp (DI p DF) = DN p
-- scope related
simp (DI DT (DI DT p)) = DT `DI` simp p
simp (DN (DN p)) = DT `DI` simp p
```

```
-- Fall throughs
simp (DC p q) = simp p `DC` simp q
simp (DD p q) = simp p `DD` simp q
simp (DI p q) = simp p `DI` simp q
simp (DN p) = DN (simp p)
simp p = p
-- See the paper on why the following disjunction rules are invalid
-- simp (DD p DT) = DT
-- simp (DD DT p) = DT
-- simp (DD p DF) = simp p
-- simp (DD DF p) = simp p

------------------------------------------------------------
-- Interpretation

type Assignment = [(Ref, Entity)]
type Interpretation1 = (Sigma, Domain, Ref → Predicate)
type IDpl = Interpretation1 → Assignment → Assignment → Bool

intDpl :: Dpl → IDpl
intDpl DT _ x y                    = x == y
intDpl DF _ x y                    = False
intDpl (DC p q) i@(sigma,dom,_) x y  = or [x `r` z && z `s` y | z ← assignments]
    where (r, s) = (intDpl p i, intDpl q i)
          assignments = sequence [[(k,v) | v ← dom] | k ← sigma]
intDpl (DN p) i x y                = x == y && not (x `r` y) where r = intDpl p i
intDpl (DI p q) i x y              = intDpl (DN (p `DC` (DN q))) i x y
intDpl (DE var) i x y              = hide var x == hide var y where hide key = filter (λ(x,y) → x
    ≠ key)
intDpl (DP name args) (_,_,intp) x y = x == y && intp name (map (lookup' x) args)

-- Like 'lookup', but throws an error in case the entity was not available.
-- We can use this version since our formulas are guarenteed to be valid.
lookup' :: Assignment → Ref → Entity
lookup' xys key | Just e ← lookup key xys = e
                | otherwise               = error ("Variable not in scope: " ++ key)



type Interpretation2 = (Domain, Ref → Predicate)
type IDpl2 = Interpretation2 → Assignment → [Assignment]

intDpl2 :: Dpl → IDpl2
intDpl2 DT _                   = return
intDpl2 DF _                   = const []
intDpl2 (DC p q) i             = intDpl2 p i>=>intDpl2 q i
intDpl2 (DN p) i               = λx → if null (intDpl2 p i x) then [[]] else []
intDpl2 (DI p q) i             = intDpl2 (DN (p `DC` (DN q))) i
intDpl2 (DE var) (dom,_)       = λx → [set var val x | val ← dom]
intDpl2 (DP name args) (_,pint) = λx → if pint name (map (lookup' x) args) then [] else [x]

set :: Ref → Entity → Assignment → Assignment
set k v xys = (k,v) : filter (λ(x,y) → x ≠ k) xys



intDpl3 :: Dpl → Fol
```

```
intDpl3 p = addDpl T [p]

addDpl :: Fol → [Dpl] → Fol
addDpl p [] = p
addDpl p (DT:ds) = addDpl p ds
addDpl p (DF:ds) = F
addDpl p ((DC d1 d2):ds) = addDpl p (d2:d1:ds)
addDpl p ((DN d):ds) = addDpl (And (Not (addDpl T [d])) p) ds
addDpl p ((DI d1 d2):ds) = addDpl p ((DN (DC d1 (DN d2))):ds)
addDpl p ((DE k):ds) = addDpl (Exists k p) ds
addDpl p ((DP f ks):ds) = addDpl (And (Predi f ks) p) ds
addDpl p ((DD d1 d2):ds) = Or (addDpl p (d1:ds)) (addDpl p (d2:ds))


-- More possible interpretation includes Visser's disjunction interpretation
```

Listing 6: src/Visser.hs

```
module Visser (Visser(..), simplify, tex, pretty, cleanVariables, intVisser1, intVisser2, intVisser3)
    where

import Fol (Ref, Fol(..))
import Dpl (Dpl(..))
import Stanford (variables)

import Data.List (intersperse, nub)
import Utils (replace)

-- Syntax

data Visser =
    VT | VF
  | VC Visser Visser
  | VD Visser Visser
  | VE Ref
  | VP Ref [Ref]
  | VSw
  | VSc
  | VQ0 Visser
  | VQ1 Visser
  deriving (Eq, Show)

tex :: Visser → String
tex VT = "\\top"
tex VF = "\\bot"
tex VSw = "\\Bowtie"
tex VSc = "\\triangle"
tex (VC p q) = tex p ++ "␣\\cdot␣" ++ tex q
tex (VD p q) = "(" ++ tex p ++ ")␣\\vee␣(" ++ tex q ++ ")"
tex (VE k) = "\\exists␣" ++ k
tex (VP f ks) = replace "-" "\\_" f ++ "(" ++ concat (intersperse "," ks) ++ ")"
tex (VQ0 p) = "?_0(" ++ tex p ++ ")"
tex (VQ1 p) = "?_1(" ++ tex p ++ ")"

pretty :: Visser → String
pretty VT = "⊤"
pretty VF = "⊥"
```

```
pretty VSw = "⋈"
pretty VSc = "△"
pretty (VC p q) = pretty p ++ "·" ++ pretty q
pretty (VD p q) = "(" ++ pretty p ++ ")∨(" ++ pretty q ++ ")"
pretty (VE k) = "∃" ++ k
pretty (VP f ks) = f ++ "(" ++ concat (intersperse "," ks) ++ ")"
pretty (VQ0 p) = "?0(" ++ pretty p ++ ")"
pretty (VQ1 p) = "?1(" ++ pretty p ++ ")"

simp :: Visser → Visser
simp (VC p VT) = simp p
simp (VC VT p) = simp p
--simp (VC p VF) = VF
--simp (VC VF p) = VF
-- TODO: Get rid of conjunctive, equal stream devices
-- Fall throughs
simp (VC p q) = simp p `VC` simp q
simp (VD p q) = simp p `VD` simp q
simp (VQ0 p) = VQ0 (simp p)
simp (VQ1 p) = VQ1 (simp p)
simp p = p

simplify :: Visser → Visser
simplify p = iterate simp p !! 2

-- Renames variables to [a..], removing gaps.
cleanVariables :: Visser → Visser
cleanVariables p = foldl rename p (zip used im ++ zip im variables)
  where
    used = nub (listem p)
    im = map (++"0") used
    listem (VE k) = [k]
    listem (VP f ks) = ks
    listem (VC p q) = listem p ++ listem q
    listem (VD p q) = listem p ++ listem q
    listem (VQ0 p) = listem p
    listem (VQ1 p) = listem p
    listem p = []
    rename (VE k) (x,y) = if k == x then VE y else VE k
    rename (VP f ks) (x,y) = VP f [if k == x then y else k | k ← ks]
    rename (VC p q) (x,y) = VC (rename p (x,y)) (rename q (x,y))
    rename (VD p q) (x,y) = VD (rename p (x,y)) (rename q (x,y))
    rename (VQ0 p) (x,y) = VQ0 (rename p (x,y))
    rename (VQ1 p) (x,y) = VQ1 (rename p (x,y))
    rename p (x,y) = p



-- Interpretation

type IVisser1 = (Dpl, Dpl, Integer)

intVisser1 :: Visser → IVisser1
intVisser1 VT = (DT, DT, 1)
intVisser1 VF = (DT, DF, 1)
intVisser1 VSw = (DT, DT, -1)
intVisser1 (VP p rs) = (DT, DP p rs, 1)
```

```
intVisser1 (VE r) = (DT, DE r, 1)
intVisser1 (VC vis1 vis2) = compVisser1 (intVisser1 vis1) (intVisser1 vis2)
intVisser1 (VQ0 vis) = (DT, qm `DI` qp, 1) where (qm, qp, _) = intVisser1 vis
intVisser1 (VQ1 vis) = error "?1 is not defined in Visser1"

compVisser1 :: IVisser1 → IVisser1 → IVisser1
compVisser1 (qm, qp, 1) (rm, rp, b) = (qm`DC`rm, qp`DC`rp, b)
compVisser1 (qm, qp, -1) (rm, rp, b) = (qm`DC`rp, qp`DC`rm, -b)



type IVisser2 = (Dpl, Dpl, Dpl, Dpl, Integer, Integer)

intVisser2 :: Visser → IVisser2
intVisser2 VT = (DT, DT, DT, DT, 1, 0)
intVisser2 VF = (DT, DT, DT, DF, 1, 0)
intVisser2 VSw = (DT, DT, DT, DT, -1, 0)
intVisser2 VSc = (DT, DT, DT, DT, 1, 1)
intVisser2 (VP p rs) = (DT, DT, DT, DP p rs, 1, 0)
intVisser2 (VE r) = (DT, DT, DT, DE r, 1, 0)
intVisser2 (VC vis1 vis2) = compVisser2 (intVisser2 vis1) (intVisser2 vis2)
intVisser2 (VQ0 vis) = (qm1, DT, qp1, qm0 `DI` qp0, 1, 0)
    where (qm1, qm0, qp1, qp0, a, i) = intVisser2 vis
intVisser2 (VQ1 vis) = (DT, DT, DT, (qm1 `DC` qm0) `DI` (qp1 `DC` qp0), 1, 0)
    where (qm1, qm0, qp1, qp0, a, i) = intVisser2 vis

compVisser2 :: IVisser2 → IVisser2 → IVisser2
compVisser2 (qm1, qm0, qp1, qp0, 1, 0) (rm1, rm0, rp1, rp0, b, j)  = (qm1`DC`rm1, qm0`DC`rm0, qp1`DC`
     rp1, qp0`DC`rp0, b, j)
compVisser2 (qm1, qm0, qp1, qp0, 1, 1) (rm1, rm0, rp1, rp0, b, j)  = (qm1`DC`rm0, qm0`DC`rm1, qp1`DC`
     rp0, qp0`DC`rp1, b, 1-j)
compVisser2 (qm1, qm0, qp1, qp0, -1, 0) (rm1, rm0, rp1, rp0, b, j) = (qm1`DC`rp1, qm0`DC`rp0, qp1`DC`
     rm1, qp0`DC`rm0, -b, j)
compVisser2 (qm1, qm0, qp1, qp0, -1, 1) (rm1, rm0, rp1, rp0, b, j) = (qm1`DC`rp0, qm0`DC`rp1, qp1`DC`
     rm0, qp0`DC`rm1, -b, 1-j)



intVisser3 :: Visser → Dpl
intVisser3 p = (qm1 `DC` qm0) `DI` (qp1 `DC` qp0)
    where (qm1, qm0, qp1, qp0, a, i) = intVisser2 p


-- More possible interpretations include support for disjunction
-- and one directional switch
```

Listing 7: src/Tests.hs

```
import Test.Framework (defaultMain, testGroup)
import Test.Framework.Providers.HUnit
import Test.Framework.Providers.QuickCheck (testProperty)

import Test.QuickCheck
import Test.HUnit

import System.IO.Unsafe
```

```
import Data.List
import Data.Map hiding (map)
import Data.Char (chr, ord)

import Stanford
import Prop(Ref)

main = defaultMain tests

tests = [
    testGroup "Stanford tests" [
        testCase "test1" test_tomapping1,
        testCase "test2" test_tomapping2,
        testProperty "test3" prop_tomapping,
        testCase "test4" test_stanford
     ],
    testGroup "Dpl tests" [

     ]
 ]

instance Arbitrary Char where
  arbitrary = chr `fmap` oneof [choose (0,127), choose (0,255)]
  coarbitrary = coarbitrary . ord

instance Arbitrary PosTree where
    arbitrary = do
        n ← choose (0, 3) :: Gen Int
        n ← return $ if n == 0 then 0 else 1
        case n of
            0 → do
                k ← choose (1, 4) :: Gen Int
                subs ← sequence (replicate k arbitrary)
                tag ← oneof [elements ["NP"], arbitrary]
                return (Phrase tag subs)
            1 → do
                tag ← arbitrary
                word ← arbitrary
                return (Leaf tag word)
    coarbitrary a b = b

variables :: [Ref]
variables = [c:"" | c ← "abcdefghijklmnopqrstuvwxyz"]
toMapping2 :: PosTree → Map Int Ref
toMapping2 tr = snd (toMapping variables tr)

test_tomapping1 = (toMapping2 . postreeS) inp @?= out
    where inp = "(NP (DT The) (NN man))"
          out = fromList [(0, "a"), (1, "a")]

test_tomapping2 = (toMapping2 . postreeS) inp @?= out
    where inp = "(ROOT (S (NP (DT The) (NN man)) (VP (VBD did) (RB not) (VP (VB make) (S (NP (PRP$ his
        ) (NN donkey)) (VP (VB see))))) (. .)))"
          out = fromList [(0, "b"), (1, "b"), (5, "a"), (6, "a")]

prop_tomapping :: PosTree → Bool
prop_tomapping postree = cntNp postree ≥ (length . nub . elems . toMapping2) postree
```

```
  where cntNp (Leaf _ _) = 0
        cntNp (Phrase "NP" subs) = 1 + sum (map cntNp subs)
        cntNp (Phrase _ subs) = sum (map cntNp subs)

test_stanford = (sentence, refs, deps) @?= (sout, rout, dout)
    where (sentence, refs, deps) = unsafePerformIO (runOnFile "test.xml")
          sout = ["if", "a", "farmer", "own", "a", "donkey", ",", "he", "beat", "it"]
          rout = fromList [(1, "e"), (2, "e"), (4, "f"), (5, "f"), (7, "e"), (9, "f")]
          dout = Dep 8 [("advcl",Dep 3 [("mark",Dep 0 []),("nsubj",Dep 2 [("det",Dep 1 [])]),("dobj",
               Dep 5 [("det",Dep 4 [])])]),("nsubj",Dep 7 []),("dobj",Dep 9 [])]
```

Listing 8: src/Utils.hs

```
module Utils (wrap, replace) where

import Data.List (isPrefixOf)

-- This is a standard subtring replacement function
replace :: String → String → String → String
replace pat repl target
  | target == ""            = ""
  | pat `isPrefixOf` target = repl ++ replace pat repl (drop (length pat) target)
  | otherwise               = head target : replace pat repl (tail target)

-- This is used for pretty printing
wrap :: Int → (String, Int) → String
wrap n (p, k) | k > n     = "(" ++ p ++ ")"
              | otherwise = p
```

References

[1] Thomas Dybdahl Ahle. thomasahle / donkey - github. https://github.com/thomasahle/donkey. Accessed: May 2013.

[2] Noam Chomsky. *Syntactic structures*. de Gruyter Mouton, 2002.

[3] Marie-Catherine De Marneffe, Bill MacCartney, and Christopher D Manning. Generating typed dependency parses from phrase structure parses. In *Proceedings of LREC*, volume 6, pages 449–454, 2006.

[4] Jan Eijck and Fer-Jan Vries. Dynamic interpretation and hoare deduction. *Journal of Logic, Language and Information*, 1(1):1–44, 1992.

[5] Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning about knowledge*, volume 4. MIT press Cambridge, 1995.

[6] Gilles Fauconnier. Espaces mentaux; aspects de la construction du sens dans les langues naturelles. 1984.

[7] Peter Thomas Geach. *Reference and generality: An examination of some medieval and modern theories*, volume 88. Cornell University Press, 1962.

[8] Jeroen Groenendijk and Martin Stokhof. Dynamic predicate logic. *Linguistics and philosophy*, 14(1):39–100, 1991.

[9] Jeroen AG Groenendijk and MJB Stokhof. Dynamic montague grammar. 1990.

[10] Irene Heim. File change semantics and the familiarity theory of definiteness. 1983.

[11] Hans Kamp. A theory of truth and semantic representation. In *Formal methods in the study of language*, pages 277–322. Mathematical Centre, Amsterdam, 1981.

[12] Hans Kamp and Uwe Reyle. From discourse to logic: an introduction to modeltheoretic semantics, formal logic and discourse representation theory, 1993.

[13] Makoto Kanazawa. Weak vs. strong readings of donkey sentences and monotonicity inference in a dynamic setting. *Linguistics and philosophy*, 17(2):109–158, 1994.

[14] D. Klein and C.D. Manning. Accurate unlexicalized parsing. In *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics-Volume 1*, pages 423–430. Association for Computational Linguistics, 2003.

[15] Heeyoung Lee, Angel Chang, Yves Peirsman, Nathanael Chambers, Mihai Surdeanu, and Dan Jurafsky. Deterministic coreference resolution based on entity-centric, precision-ranked rules. *Computational Linguistics*, (Just Accepted):1–54, 2013.

[16] Heeyoung Lee, Yves Peirsman, Angel Chang, Nathanael Chambers, Mihai Surdeanu, and Dan Jurafsky. 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*, pages 28–34. Association for Computational Linguistics, 2011.

[17] James Malone, Ele Holloway, Tomasz Adamusiak, Misha Kapushesky, Jie Zheng, Nikolay Kolesnikov, Anna Zhukova, Alvis Brazma, and Helen Parkinson. Modeling sample variables with an experimental factor ontology. *Bioinformatics*, 26(8):1112–1118, 2010.

[18] Richard Montague. Universal grammar. *Theoria*, 36(3):373–398, 1970.

[19] Richard Montague. The proper treatment of quantification in ordinary english. *Approaches to natural language*, 49:221–242, 1973.

[20] Barbara Partee and Mats Rooth. Generalized conjunction and type ambiguity. *Formal Semantics: The Essential Readings*, pages 334–356, 1983.

[21] Vaughan R Pratt. Semantical consideration on floyo-hoare logic. In *Foundations of Computer Science, 1976., 17th Annual Symposium on*, pages 109–121. IEEE, 1976.

[22] Karthik Raghunathan, Heeyoung Lee, Sudarshan Rangarajan, Nathanael Chambers, Mihai Surdeanu, Dan Jurafsky, and Christopher Manning. A multi-pass sieve for coreference resolution. In *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, pages 492–501. Association for Computational Linguistics, 2010.

[23] William J Rapaport. September 2006.". *A History of the Sentence "Buffalo buffalo buffalo Buffalo buffalo."". Accessed*, 23, 2006.

[24] Mats Rooth and Barbara Partee. Conjunction, type ambiguity and wide scope or. In *Proceedings of the first west coast conference on formal linguistics*, pages 353–362, 1982.

[25] Stuart Jonathan Russell, Peter Norvig, Ernest Davis, Stuart Jonathan Russell, and Stuart Jonathan Russell. *Artificial intelligence: a modern approach*, volume 2. Prentice hall Englewood Cliffs, 2010.

[26] The Donkey Sanctuary. The donkey sanctuary: Together we can help end the suffering. http://www.thedonkeysanctuary.org. Accessed: March 2013.

[27] A. Visser. The donkey and the monoid: Dynamic semantics with control elements. *Artificial Intelligence Preprint Series*, 1, 1999.