FUNDAÇÃO GETULIO VARGAS

ESCOLA DE MATEMÁTICA APLICADA - FGV/EMAp

CURSO DE GRADUAÇÃO EM MATEMÁTICA APLICADA

Semântica Computacional para Textos Normativos

por

Guilherme Paulino Passos

Rio de Janeiro

2016

FUNDAÇÃO GETULIO VARGAS

ESCOLA DE MATEMÁTICA APLICADA - FGV/EMAp

CURSO DE GRADUAÇÃO EM MATEMÁTICA APLICADA

Semântica Computacional para Textos Normativos

”Declaro ser o único autor do presente projeto de monografia que refere-se ao plano de trabalho a ser executado para continuidade da monografia e ressalto que não recorri a qualquer forma de colaboração ou auxílio de terceiros para realizá-lo a não ser nos casos e para os fins autorizados pelo professor orientador”

Guilherme Paulino Passos

Orientador: Prof. Dr. Alexandre Rademaker

Rio de Janeiro

2015

GUILHERME PAULINO PASSOS

Semântica Computacional para Textos Normativos

"Monografia apresentada à Escola de Matemática Aplicada - FGV/EMAp como requisito parcial para a obtenção do grau de Bacharel em Matemática Aplicada."

Aprovado em ____ de _____ de _____ .

Grau atribuido à Monografia: ____ .

Professor Orientador: Prof. Dr. Alexandre Rademaker

Escola de Matemática Aplicada

Fundação Getulio Vargas

Professor Tutor: Prof. Dr. Paulo Cezar Pinto Carvalho

Escola de Matemática Aplicada

Fundação Getulio Vargas

Sumário

1 Introdução **4**

1.1 Estudo Formal e Computacional da Linguagem 4

1.2 Proposta do Trabalho 6

1.3 Mapa de Leitura 8

2 Representação semântica **10**

2.1 Considerações Iniciais 10

2.2 Arquitetura da Gramática 12

2.3 Cálculo Lambda 13

2.3.1 Dificuldades – Ambigüidades de Escopo 15

2.4 Armazenamento de Cooper 16

2.4.1 Dificuldades 20

2.5 Armazenamento de Keller 21

2.6 *Hole Semantics* 23

3 Inferência **30**

4 Curt – Sistema de Diálogo e Wordnet **31**

4.1 Arquitetura do Curt 33

4.2 Integrando à Wordnet 40

4.3 Helpful Curt 48

5 Conclusão **49**

6 Referências **50**

1 Introdução

Trocar “pp.X-Y” por “pp.X–Y” nas citações

Colocar link para minha implementação e para os arquivos originais

1.1 Estudo Formal e Computacional da Linguagem

Linguagem natural é o termo utilizado para se referir à linguagem humana que não foi criada através de um planejamento consciente, como o Português, o Inglês, o Japonês, entre outros. O termo se contrapõe às linguagens construídas, como o Esperanto, e às linguagens formais, tais como as linguagens de programação ou linguagens lógicas.

Nas últimas décadas, o estudo computacional da linguagem natural passou por intenso desenvolvimento. Isto se deu pelo crescimento e grande aproximação (ou mesmo fusão) de áreas em diferentes departamentos, como Lingüística Computacional, em linguística, e Processamento de Linguagem Natural (*Natural Language Processing*, ou simplesmente NLP), em computação. (Jurafsky e Martin, 2009, pp. xxi, 10) [1]

Exemplos bem sucedidos de aplicações são a Siri, uma assistente do sistema operacional iOS que interage com o usuário utilizando linguagem natural; serviços de tradução automática, como o do Google, que apresentam constante melhora; e também diversas empresas relacionadas a inteligência de marketing ou empresarial (*marketing intelligence* e *business intelligence*) destinadas a fazer análise de dados a partir de textos em linguagem natural.

O uso de modelos matemáticos de diferentes formas e tradições foi um passo essencial para o desenvolvimento da ciência, bem como para a levar o conhecimento adquirido a aplicações. Historicamente, ocorreu uma tensão (ou, ao menos, um distanciamento)

[1] Preferimos a visão que diferencia conceitualmente a Lingüística Computacional do Processamento de Linguagem Natural. A primeira seria o estudo da linguagem humana através de modelos computacionais. Seria, portanto, de interesse científico, buscando explicação e compreensão do fenômeno. Já a segunda seria a disciplina de métodos computacionais relacionados à linguagem para resolução de problemas práticos. Assim, seria uma disciplina de engenharia, voltada para a aplicação. (Smith) Isto, entretanto, não significa que sejam comunidades apartadas ou que os métodos utilizados sejam distintos. A diferença conceitual na prática apenas se realiza por uma distinção de enfoque ou de finalidade.

Há a visão de tais nomes representariam a mesma área, ou mesmo de que um campo estaria incluído no outro. Por exemplo, Grishman (1986, pp. 1-9) utiliza o nome “Lingüística Computacional” para incluir tanto a abordagem de finalidade científica quanto a de finalidade prática.

entre dois paradigmas em NLP: o simbólico e o probabilístico.

Tal divisão existiu de modo particularmente notável do fim da década de 50 ao fim da de 60. Desta época, do paradigma simbólico participaram o trabalho de Noam Chomsky em linguagens formais e sintaxe gerativa, o trabalho de lingüistas e cientistas da computação em algoritmos de análise sintática (*parsing*), bem como os da área de inteligência artificial, como sistemas baseados em lógica formal e correspondência de padrões (*pattern matching*), influenciados pelo famoso *Logic Theorist* de Allen Newell, Herbert Simon e Cliff Shaw, um exemplo de sistema baseado em lógica e raciocínio automático.

Para Manning e Schütze (1999, pp. 4-7), esta tradição é representativa da escola do racionalismo, no embate filosófico entre racionalismo e empirismo. Aqui, racionalismo é a posição segundo a qual é possível, de modo significativo, adquirir conceitos e conhecimento independentemente da experiência dos sentidos. (Markie, 2015) No caso em questão, conhecimento lingüístico.

Na tradição estocástica, dois exemplos são o trabalho de Bledson e Browning de um sistema bayesiano para reconhecimento ótico de caracteres, bem como o uso de métodos bayesianos por Mosteller e Wallace para atribuir autoria de trechos d'*O Federalista*. Já nas décadas de 70 e 80, houve grande desenvolvimento de algoritmos de reconhecimento de fala, como o uso de Cadeias Ocultas de Markov. (Jurafsky e Martin, 2009, pp.10-11)

Assim, Manning e Schütze (1999, pp. 4-7) apresentam esta linha como representativa do empirismo – pressupondo menos conhecimento inato e ressaltando o aprendizado a partir de exemplos.

Métodos de lingüística computacional e processamento de linguagem natural foram aplicados para as mais diversas sub-áreas da lingüística. Algumas delas são: a fonologia (estudo dos sons), a morfologia (o estudo da formação e composição de palavras), a sintaxe (o estudo de como as palavras se combinam para formar orações e frases), a semântica (o estudo do significado) e a pragmática (o estudo de como o contexto influencia no significado). (Eijck e Unger, 2010, p. 2)

Em particular, uma linha de estudos de semântica é da chamada *semântica de teoria de modelos* ou *semântica formal*. Este método busca descrever o significado de linguagem natural através de um modelo, uma estrutura abstrata que codifica a informação passada. Particularmente, são usados modelos formais, isto é, matematicamente bem definidos.

Um fundamento por trás deste método é como se segue: Entre as funções da linguagem, está a de descrever a maneira que o mundo é (ou de comunicar tais descrições). Neste uso, uma afirmação feita em linguagem é *sobre* algo; em particular, usualmente algo do mundo real. Caso a afirmação corretamente reflita o modo pelo qual o mundo é, é dita *verdadeira*. Se não, é *falsa*. Dada a importância do uso da linguagem para troca de informações a respeito do mundo, em particular para a razão prática – decisões sobre como agir –, pareceria razoável que coubesse a uma teoria da semântica descrever a relação entre o significado de expressões e a determinação de sua verdade ou falsidade, colocando este aspecto como central. Um modo de realizar esta descrição seria pelo uso de modelos. Estes seriam uma representação abstrata do mundo, de modo que a relação entre a linguagem e a descrição do mundo possa ser feita de um modo tratável. (Kamp e Reyle, 1993, pp. 11-13)

Esse método é devido a Richard Montague, sendo a realização específica feita pelo mesmo hoje conhecida como *Gramática de Montague*. Montague foi um lógico e, com efeito, seu trabalho utilizou métodos da lógica para a descrição da linguagem natural. Em lógica, também a definição de significado (ou de verdade) se utiliza de modelos: a noção de modelo foi construída para definir os conceitos de verdade e de conseqüência lógica. O fundamento não é muito diferente: um modelo é uma representação abstrata de um estado de coisas no mundo. Apesar disto, o uso da lógica em linguagem não foi de acordo com a visão comum da época, segundo a qual as linguagens naturais não seriam sistemáticas o suficiente; com efeito, as lógicas formais teriam sido desenvolvidas exatamente para permitir a comunicação de afirmações científicas um modo rigoroso, que seria impossível para a linguagem natural. Apesar disto, um precursor deste método foi o alemão Gottlob Frege, criador da lógica de primeira ordem. (Kamp e Reyle, 1993, pp. 12,16-17,21-23)

1.2 Proposta do Trabalho

Com efeito, este é um trabalho na área de semântica computacional, em um sentido estrito. Denotamos aqui *semântica computacional* como a área que busca construir representações formais de modo algorítmico para o significado de expressões de linguagem natural. (Kamp, 2010, p. ix) Incluímos também na área o uso dessas representações para realizar inferências, isto é, extrair conclusões. É, assim, uma versão computacional da semântica formal.

Em mais detalhes, no estudo computacional da semântica, uma idéia central é a

de que é possível capturar o significado de expressões de linguagem natural a partir de estruturas formais. Como vimos, isto é a definição do campo de semântica formal. O intuito é relacionar estruturas lingüísticas com conhecimento sobre o mundo, que é representado de alguma maneira. São todas questões da semântica formal: a escolha de qual o modo de representar, quais as propriedades da representação e como associar palavras e frases a estruturas. O uso de estruturas formais tem utilidade para lingüistas por permitir que discutam significado de modo mais rigoroso, menos ambíguo. Esta tradição deriva diretamente dos trabalhos de Richard Montague. (Blackburn e Bos, 2005, p. xii)

Entretanto, um modo de expandir essa análise é caminhar em direção à semântica computacional, buscando realizar as tarefas da semântica formal por uso de um computador. Isso expande a utilidade de modelos formais para além da análise por um humano. As representações formais tornam possível que um computador consiga acessar o significado e trabalhar com ele, o utilizando para finalidade distintas. Em especial, para a atividade de *inferência*, isto é, tornar explícita informação que estava implícita. Portanto, são objetivos centrais da área a automatização de construção de representações a partir de textos em linguagem natural, bem como a automatização da extração de inferências a partir de representações formais já feitas.

Aqui, seguimos o livro de Blackburn e Bos (2005), revisando seu conteúdo e realizando alguns de seus exercícios. Os autores apresentam código em Prolog para os desenvolvimentos feitos no texto. Nosso trabalho é feito por modificações neste código original.Em particular, ao fim integramos o sistema Curt, apresentado pelos autores, à Wordnet. A Wordnet é um banco de dados do léxico inglês, sendo uma fonte adequada de informação semântica de diversas palavras, para integração ao conhecimento do sistema. (Fellbaum, 1998) Assim, o inglês será a linguagem-objeto deste texto.

Atualizado

Do ponto de vista científico, a semântica computacional traz novos métodos para explorar a teoria da semântica formal. Exemplos podem ser testados em grande quantidade, permitindo melhor verificação empírica e simulações. Já da visão das aplicações, o processamento da semântica ainda é um desafio, oferecendo novos métodos de valor. Um processamento semântico adequado é útil para diversas tarefas, como para sistemas de responder questões, extração de informações, resumo automático de textos, tradução automática, entre outros. (Dagan et al., 2012, pp. 1-2,10-14)

Assim, a utilidade da teoria e dos métodos pode ser colocado em teste tanto através

do poder explicativo em linguagens naturais, quanto pela utilidade em aplicações práticas. A respeito da questão semântica, há uma série de desafios chamada *PASCAL Recognizing Textual Entailment (RTE) Challenges*. Neles, são apresentados exemplos de *implicação textual* (*textual entailment*):

Dados dois fragmentos de texto, a tarefa é reconhecer se o significado de um pode ser inferido a partir do significado do outro. Mais especificamento, dado um par de expressões textuais — T, o texto base, e H, a hipótese — dizemos que T acarreta H se o significado de H pode ser inferido do significado de T, de acordo com o que seria tipicamente interpretado por falantes da língua. (Dagan et al., 2006, p. 1) Apesar desta definição parecer problemática, há resultados que mostram que existe consistência suficiente nos julgamentos humanos, validando a proposta. (Dagan et al., 2012, p. 3)

Dois exemplos são:

Texto	Hipótese	Implicação Textual
Sessões no Clube Caverna pagaram aos Beatles £15 à noite e £5 na hora do almoço.	Os Beatles tocaram no Clube Caverna na hora do almoço.	Verdadeiro
A American Airlines começou a demitir centenas de comissários de bordo na terça-feira após um juiz ter rejeitado a proposta da União de bloquear as perdas de empregos.	A American Airlines chamará de volta centenas de comissários de bordo para aumentar o número de vôos que opera.	Falso

Versões mais avançadas dos métodos estudados aqui foram utilizadas para atacar o problema em Bos e Markert (2005) e Bos e Markert (2006), inclusive com resultados de precisão superior à do melhor resultado feito durante a RTE-1. (Dagan et al., 2012, p. 89)

1.3 Mapa de Leitura

No capítulo 2, apresentaremos o modelo de representação semântico. Usaremos a lógica de primeira ordem como linguagem formal para capturar frases completas. Isto será complementado com ferramentas que permitam uma combinação mais natural do significado de partes para formar as sentenças, o que será feito formalmente pelo formalismo

cálculo lambda. Também apresentamos representações mais complexas, desenvolvidas para resolver as chamadas ambigüidades des escopo.

No capítulo 3, apresentaremos rapidamente métodos de inferência estudados no trabalho. Veremos o método do tableau, bem como a resolução. Em nosso sistema final, no entanto, usaremos ferramentas já prontas, de maior sofisticação, além do escopo do trabalho.

Já no capítulo 4.3, exibiremos a combinação dos métodos de representação e de inferência no Curt, um pequeno sistema de diálogo desenvolvido por Blackburn e Bos (2005). Também mostraremos a integração por nós realizada deste sistema com a Wordnet.

Por fim, no capítulo 5, apresentaremos conclusões e próximos passos possíveis nesta linha de trabalho.

Estou em dúvida se vale a pena desenvolver este capítulo, já que o Curt só usa provadores de teorema já prontos.

2 Representação semântica

2.1 Considerações Iniciais

Desejamos associar a cada expressão de linguagem natural um significado formal, simbólico. Além disso, desejamos fazê-lo de modo algorítmico, que possa ser reproduzido por um computador. Portanto, um primeiro passo importante é encontrar um modo adequado de representar o significado, que satisfaça nossos objetivos lingüísticos, bem como que seja manipulável por nossas ferramentas computacionais.

A linguagem formal que utilizaremos para representar o significado de frases é *lógica de primeira ordem*. Jurafsky e Martin (2009) apresentam como propriedades interessantes para representações: verificabilidade, não-ambigüidade, existência de uma forma canônica, capacidade de inferência, uso de variáveis e expressividade. Todas estas são possuídas pela lógica de primeira ordem, ao menos até certo ponto. Além disso, é um sistema bem compreendido e bastante flexível.

Adicionar “Iremos apresentar formalmente esta lógica na seção seguinte.”, bem como escrever esta seção seguinte.

Ainda que tenhamos escolhido a lógica de primeira ordem para ser a linguagem das representações semânticas para frases, isto não nos informa qual deve ser a representação semântica de palavras e expressões menores. Fórmulas desta lógica definem sentenças completas (no máximo, abertas à interpretação de variáveis livres). Talvez algumas possibilidades poderiam ser feitas através de termos, mas não está de todo claro qual seria o significado de uma expressão como “*to run*” (“*correr*”) ou “*that walks*” (“*que anda*”).

Em nossos pressupostos, adotaremos o *Princípio da Composicionalidade*, usualmente atribuído a Gottlob Frege. (Blackburn e Bos, 2005, p. 94) Segundo o mesmo, o significado de expressões complexas é função das expressões mais simples que a compõem. Em um exemplo como “*Caim kills Abel*”, isto nos informa que o significado desta frase depende do significado de “*Caim*”, “*kills*” e “*Abel*”. Entretanto, isto não nos diz como funciona esta dependência, ou a função que leva o significado das expressões simples ao da expressão complexa.

Por exemplo, podemos entender que o significado de “*kills*” é o predicado binário $kill(\ldots,\ldots)$, onde convencionamos que o primeiro argumento é o agressor (isto é, aquele que mata) e o segundo argumento é a vítima (aquele que é morto). Também podemos

entender os significados de "*Caim*" e "*Abel*" como as constantes $caim$ e $abel$, respectivamente. Assim, apesar de $kill(abel, caim)$ ser formada com o significado destes três termos, respeitando a composicionalidade, esta não é a expressão que queremos, e sim $kill(caim, abel)$.

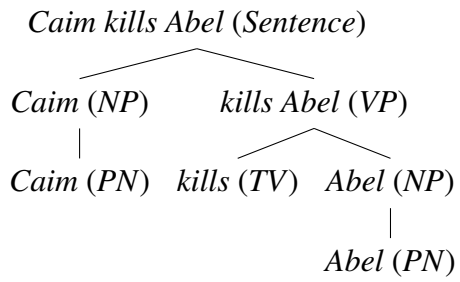
O que nos falta é a *sintaxe*. A sintaxe é o conjunto de regras e processos que organizam a estrutura de frases.

achar uma boa referência

Assim, as palavras em uma frase existem em relação a uma certa estrutura, que é essencial para capturar o significado. No inglês, com a estrutura usual de *Sujeito - Verbo - Predicado*, entendemos que "*Caim kills Abel*" significa $kill(caim, abel)$, e não $kill(abel, caim)$.

O foco deste trabalho não é na sintaxe, de modo que utilizamos uma sintaxe simples: a gramática é implementada pelo mecanismo de Gramática de Cláusulas Definidas (*Definite Clause Grammar* - DCG). A análise sintática é feita na forma de uma árvore cujos nós que são folhas são categorias sintáticas básicas (tais como sujeito (*noun*), verbo transitivo (*transitive verb*) e quantificador (*quantifier*, considerado caso particular de *determiner*). Já os nós que não são folhas representam categorias sintáticas complexas (tais como sintagma nominal (*noun phrase*) ou sintagma verbal (*verb phrase*). (Blackburn e Bos, 2005, p. 58)

Um exemplo de tal árvore, para a frase "*Caim kills Abel*", seria:



Aqui, temos as classes sintáticas:

NP – noun phrase (sintagma nominal)

PN – proper noun (nome próprio)

VP – verb phrase (sintagma verbal)

TV – transitive verb (verbo transitivo)

A decomposição parece lingüisticamente razoável, bem como útil para a compreensão do significado. Resta saber, assim, como podemos elaborar a construção da

semântica de uma frase completa a partir de tal análise sintática e dos significados dos termos mais elementares. Portanto, devemos construir nossa gramática.

2.2 Arquitetura da Gramática

Blackburn e Bos (2005, p. 86) apontam três princípios a serem observados na construção da gramática: modularidade – divisão da gramática em componentes com finalidades bem definidas e que interagem de modo transparente –, extensibilidade – ser fácil de complementá-la – e reusabilidade – capacidade de reaproveitar grandes partes da gramática, mesmo alterando as representações semânticas.

Para seguí-los, o método usado foi dividir a gramática segundo dois critérios: se o componente trata do léxico ou de regras gramaticais (isto é, combinando expressões para criar outras maiores) e se é sintático ou semântico. Assim, um dos componentes da gramática é um léxico que enumera as palavras e expressões aceitas, afirmando determinadas propriedades. Um segundo trata da semântica dos léxicos, isto é, da representação semântica dada a palavras individualmente ou expressões básicas (aqui, isto é principalmente feito se utilizando da classificação sintática, isto é, a classe sintática e a palavra em si determinam a semântica). Um terceiro item trata das regras da sintaxe, isto é, da classificação sintática de expressões complexas a partir da classificação de termos menos (incluindo aspectos como concordância de gênero ou número). Ademais, um quarto componente trata das regras semânticas, isto é, como combinar o significado de expressões menores para formar o significado de expressões maiores. Por fim, utilizamos uma quinta parte que se utiliza de todas as anteriores para fazer a relação entre o texto recebido e a representação semântica desejada. Um exemplo pode ser visto na tabela 1, tratando do exemplo da representação semântica mais simples, a do cálculo lambda, que veremos a seguir.

LAMBDA.PL	sintaxe	semântica
léxico	ENGLISHLEXICON.PL	SEMLEXLAMBDA.PL
gramática	ENGLISHGRAMMAR.PL	SEMRULESLAMBDA.PL

Tabela 1: Gramática para semântica lambda

Ao alterarmos a representação semântica, de fato nossas principais alterações estarão nos componentes semânticos.

2.3 Cálculo Lambda

Para realizar um método sistemático de composição dos significados, é introduzido o formalismo do *cálculo lambda*. Aqui, ele será uma extensão da linguagem da lógica de primeira ordem. Dois símbolos novos serão introduzidos: o símbolo de abstração "λ" e o de aplicação "@".

O símbolo "λ" será um operador sobre variáveis, permitindo a "captura" das mesmas, do mesmo modo a que um quantificador (como "$\forall$"). Por exemplo, sendo $man(x)$ uma fórmula de primeira ordem, $\lambda x.man(x)$ é uma fórmula do nosso cálculo lambda, em que a variável x está capturada pelo operador λ; alternativamente, $\lambda x.$ está *abstraindo sobre* x.

Por sua vez, o símbolo "@", que conecta duas fórmulas de cálculo lambda, representa uma *aplicação*. Assim, se F e A são duas fórmulas de cálculo lambda, $F@A$ é também uma fórmula de cálculo lambda, chamada uma *aplicação funcional* de F em A, ou uma aplicação na qual F é um *funtor* e A é o *argumento*. Por exemplo, em $\lambda x.man(x)@john$, o funtor é $\lambda x.man(x)$ e o argumento é $john$.

Uma expressão de aplicação funcional representa o comando de aplicar o argumento no funtor, que usualmente será prefixado por uma abstração. A interpretação desse comando é: retire o prefixo de abstração do funtor e, em toda ocorrência da variável abstraída, a substitua pelo argumento da aplicação. Por exemplo, em $\lambda x.man(x)@john$, o funtor é $\lambda x.man(x)$ e a interpretação do comando é de retirar o prefixo $\lambda x.$ e substituir toda ocorrência de x no funtor pelo argumento $john$, o que produz o resultado de $man(john)$. Transformar uma aplicação em sua fórmula resultante após o processo de aplicação é uma operação chamada de β-*redução*, β-*conversão* ou λ-*conversão*. (Blackburn e Bos, 2005, p. 67)

Destacamos que aplicações podem ser subfórmulas de outras fórmulas, com a β-redução da fórmula maior sendo a β-redução de suas subfórmulas, bem como que não é necessário ser um termo ou uma variável para ser um argumento de uma aplicação. Veja este exemplo: É bem formada a fórmula $(\lambda P.P@mia)@\lambda x.woman(x)$. Em uma primeira etapa de β-redução, chegamos à fórmula $\lambda x.woman(x)@mia$ e aí, mais uma vez realizando a operação, chegamos à sua β-redução final $woman(mia)$.

Um cuidado a se ter é que pode ser necessário trocar o símbolo das variáveis em uma aplicação. É suficiente trocar todas as variáveis ligadas (isto é, capturadas por um

operador) do funtor por variáveis novas, não utilizadas até então. A operação de substituir todas as variáveis ligadas por outras é chamada de α-*conversão*, enquanto se uma fórmula pode ser gerada através de α-conversão de outra, as duas fórmulas são ditas α-*equivalentes*. Para um exemplo em que não realizar a α-conversão antes de uma β-conversão pode gerar problemas, basta realizar a β-conversão da seguinte expressão: $\lambda x.\exists y.not\_equal(x, y)@y$. O resultado incorreto seria $\exists y.not\_equal(y, y)$, enquanto o resultado adequado seria $\exists y.not\_equal(z, y)$.

Desse modo, temos o cálculo lambda como uma "linguagem de cola", permitindo fazer composições de expressões até gerar verdadeiras expressões de primeira ordem. A abordagem então é criar, de algum modo, a representação semântica a nível de léxico (isto é, a nível de classes sintáticas básicas), bem como montar a representação semântica a nível da gramática, pela composição de termos mais simples, de algum modo compatível com a semântica a nível lexical. Vejamos alguns exemplos:

Para nomes próprios (*proper names*), a semântica é: $\lambda u.u@symbol$, onde $symbol$ representa o símbolo do nome próprio (por exemplo, $john$).

Por sua vez, para verbos transitivos temos a semântica $\lambda k.\lambda y.k@(\lambda x.symbol(y, x))$, onde mais uma vez $symbol$ reprenta o símbolo específico da palavra (por exemplo, $kill$).

Pensemos agora no sintagma verbal (*verb phrase*) "*kills Abel*". Um modo natural de pensar na composição é, sendo A a expressão semântica de "*kill*" e B, a de "*Abel*", realizar a aplicação $A@B$. Com efeito, fazendo isso teríamos:

$$(\lambda k.\lambda y.k@(\lambda x.kill(y, x)))@\lambda u.u@abel$$
$$\lambda y.((\lambda u.u@abel)@(\lambda x.kill(y, x)))$$
$$\lambda y.(\lambda x.kill(y, x)@abel)$$
$$\lambda y.kill(y, abel)$$

Agora, podemos juntar o sintagma nominal (e também nome próprio) "*Caim*" e o sintagma verbal "*kills Abel*", aplicando a semântica do segundo na do primeiro, de onde teríamos:

$$(\lambda u.u@caim)@(\lambda y.kill(y, abel))$$
$$(\lambda y.kill(y, abel))@caim$$
$$kill(caim, abel)$$

Assim, chegamos a uma representação da frase "*Caim kills Abel*" que é uma expressão de lógica de primeira ordem, utilizando o cálculo lambda como ferramenta para composição sistemática do sentido de expressões menores.

2.3.1 Dificuldades – Ambigüidades de Escopo

Apesar deste método produzir resultados interessantes, ele não é suficiente. Uma característica particular é que, do modo que realizamos, cada decomposição sintática está associada a apenas uma possibilidade semântica. Isto não quer dizer que o modelo até então não consegue tratar de ambigüidades.

Em primeiro lugar, as ambigüidades lexicais podem ser tratadas colocando em nosso sistema todos os sentidos possíveis de determinada expressão. Assim, homógrafos (palavras com a mesma grafia mas significados distintos) podem ser considerados como entradas distintas em nosso banco de dados da semântica lexical. Um uso interessante da linguagem Prolog está no fato de que a mesma possibilita a geração de diversos resultados possíveis, pelo mecanismo de *backtracking*. Assim, a implementação em Prolog permite que a semântica a nível léxico seja capturada. Em segundo lugar, ambigüidades por diferentes possibilidades de decomposição sintática de uma mesa frase também podem ser tratadas pelo modelo até então. Novamente, a implementação se beneficia do mecanismo de *backtracking* do Prolog, de modo que diferentes decomposições sintáticas e seus significados associados podem ser gerados sucessivamente.

Checar ambigüidades sintáticas.... Funciona mesmo? Exemplo?

Entretanto, podemos apontar um tipo de ambigüidade que, até então, nosso modelo é incapaz de tratar: as ditas *ambigüidades de escopo*. (Blackburn e Bos, 2005, p. 105-109) As ambigüidades de escopo são melhor explicadas através de exemplos.

Analisemos a frase:

"*Every man loves a woman.*"

Esta frase parece ter duas interpretações possíveis: na primeira, para cada homem existe uma mulher amada por aquele. Possivelmente, são mulheres distintas. Já na segunda leitura, existe uma mulher específica que é amada por todos os homens.

Essa dúvida parece ser gerada pelo *escopo* dos quantificadores "*every*" e "*a*". Caso o quantificador "*every*" seja *mais externo* (ou *out-scoping*) ao quantificador "*a*", então

teremos a primeira leitura. Neste caso, também dizemos que o quantificador "*every*" tem *escopo sobre* o quantificador "*a*". Por outro lado, caso o quantificador "*a*" tenha escopo sobre o quantificador "*every*", a leitura será a segunda. Perceba que, ao que parece, as ambigüidades de escopo não são geradas por, realmente, análises sintáticas distintas, mas sim por uma dificuldade de atribuição de significado à uma decomposição sintática em particular.

Que o nosso sistema atual não é capaz de representar esse tipo de ambigüidade pode ser visto pelo fato de que a representação semântica é única, dados o sentido dos termos mais simples e a decomposição sintática. Precisamos, assim, aprimorar o modelo.

Para termos um olhar em direção à solução, podemos notar que a ocorrência de quantificadores gera seus problemas na função sintática de sintagma nominal (*noun phrase*), pois a combinação quantificador e substantivo (*determiner* + *noun*) ocorre apenas nela. Isso sugere que alteremos o modo pelo qual tratamos a semântica dos sintagmas nominais com quantificadores.

2.4 Armazenamento de Cooper

Para o problema das ambigüidades de escopo, a solução computacional proposta é o uso de *armazenamentos*. Nesta abordagem, a representação semântica de cada expressão deixa de ser a de uma simples fórmula em cálculo lambda, para ser a de uma representação de múltiplas formas possíveis.

Em particular, começaremos com o *armazenamento de Cooper*. Esta é uma técnica desenvolvida por Robin Cooper para lidar com ambigüidades de escopo de quantificadores. (Blackburn e Bos, 2005, p. 113) Intuitivamente, a idéia está em adicionar a possibilidade de substituir uma representação mais detalhada de um sintagma nominal por uma nova variável e "armazenar" a representação completa deste sintagma nominal para uso posterior. Ao fim, as representações podem ser "resgatadas" do armazenamento, em qualquer ordem. Ao se "resgatar" uma representação após alguma outra, o quantificador do sintagma nominal resgatado posteriormente poderá ter escopo mais externo do que um quantificador da representação "resgatada" anteriormente. Desse modo, ao se possibilitar os "resgates" em ordens distintas, diferentes representações são formadas.

Agora cada expressão (isto é, cada nó da árvore de análise sintática (*parse*)) é associada a uma n-upla chamada "armazenamento". O primeiro elemento do armazenamento

será uma fórmula de cálculo lambda, bem como antes. É uma representação "nuclear" da expressão. Com efeito, chamaremos este elemento de *núcleo* do armazenamento. Por sua vez, os outros elementos da n-upla serão pares (β, i), em que β é uma representação semântica para um sintagma nominal e i é um indice para este sintagma. Estes pares são denominados *operadores de ligação indexados* (*indexed binding operators*).

Com mais detalhes, *a priori* as representações não diferem muito de como eram. Os nós das folhas, não sendo nenhum um sintagma nominal quantificado, são análogos ao modo anterior, sendo armazenamentos com apenas uma entrada. Já um nó não-terminal pode ter sua representação montada de um modo "usual": ele tem como núcleo uma combinação dos núcleos de cada um de seus filhos na árvore; isto é, é a combinação dos núcleos dos armazenamentos dos termos que compõem a expressão mais complexa. Esta combinação é exatamente do mesmo modo como era feito até então. O restante do armazenamento do nó não-terminal é a justaposição (*append*) do restante dos armazenamentos de cada um dos termos filhos. Em suma: quando a expressão é composta por outras na análise sintática, tudo ocorre de modo análogo a como ocorria na representação "pura" por cálculo lambda, preservando os operadores de ligação indexados de todas as sub-expressões que compõem a expressão maior.

Caso o nó não-terminal não seja um sintagma nominal quantificado, a represnetação "usual" é a sua única possível. Entretanto, o processo possui uma diferença quando o nó não-terminal é um sintagma nominal quantificado. Além da composição "usual" para outros nós, há uma segunda representação possível. Isso merece ser destacado:

Armazenagem (Cooper)

Seja o armazenamento $\langle \phi, (\beta, j), \ldots, (\beta', k) \rangle$ a representação semântica "usual" para um sintagma nominal quantificado. O armazenamento $\langle \lambda u.(u@z_i), (\phi, i), (\beta, j), \ldots, (\beta', k) \rangle$, onde i é um índice único[2]também é uma representação para este sintagma nominal quantificado.

Isto significa que sintagmas nominais quantificados podem ter suas representações montadas de dois modos. Neste ponto, nosso algoritmo terá uma escolha de aplicar ou não a regra de armazenagem. Ao se desejar saber a representação de uma frase em específico, esperaremos que nosso sistema nos retorne todas as representações possíveis. Perceba

[2]isto é, não utilizado até então

também que a regra não é recursiva. Há apenas duas opções: manter a representação "usual" ou realizar a operação de armazenagem.

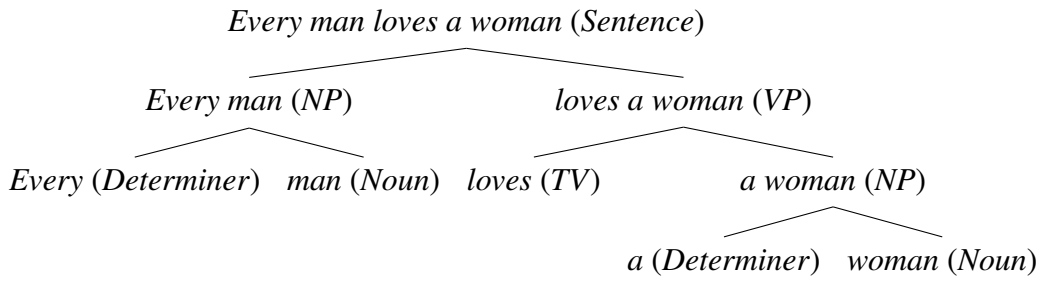
Após todo este processo, teremos uma frase cuja representação é um armazenamento. É necessário lidar com isto de algum modo, pois o que desejamos é que uma frase possa ser representada por expressões de lógica de primeira ordem, não por um armazenamento. Aqui é que poderemos "resgatar" nossos operadores de ligação indexado, que foram previamente armazenados. Para isso, usaremos a seguinte regra de resgate:

Resgate (Cooper)

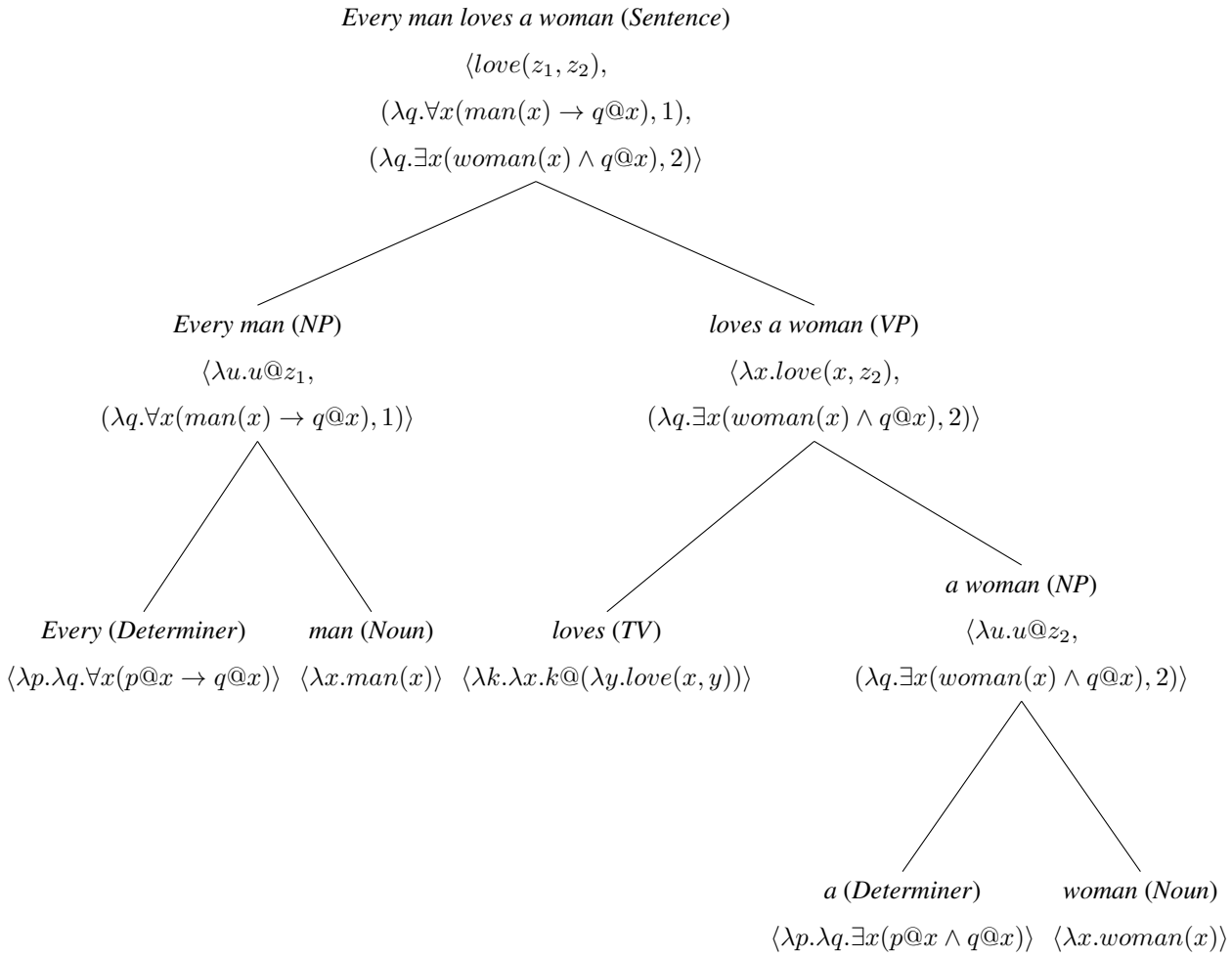
Sejam σ_1 e σ_2 duas seqüências (possivelmente vazias) de operadores de ligação. Se o armazenamento $\langle \phi, \sigma_1, (\beta, i), \sigma_2 \rangle$ a uma frase (*sentence*), então o armazenamento $\langle \beta @ \lambda z_i . \phi, \sigma_1, \sigma_2 \rangle$ também é associado a esta frase.

Um armazenamento composto apenas por um núcleo, após sucessivas aplicações da regra de resgate, será uma fórmula bem formada de primeira ordem, como desejávamos.

Para visualizarmos este processo, vamos para um exemplo:



Esta é a árvore de análise sintática. Construindo os significados a partir das folhas e subindo, uma das possíveis árvores que podemos alcançar é:



Agora, o que resta é converter o armazenamento representativo da frase completa nas possíveis fórmulas de primeira ordem através da operação de resgate.

Inicialmente:

$$\langle love(z_1, z_2),$$
$$(\lambda q.\forall x(man(x) \rightarrow q@x), 1), (\lambda q.\exists x(woman(x) \wedge q@x), 2)\rangle$$

Resgatando o operador de ligação 1:

$$\langle \lambda q.\forall x(man(x) \rightarrow q@x)@(\lambda z_1.love(z_1, z_2)),$$
$$(\lambda q.\exists x(woman(x) \wedge q@x), 2)\rangle$$

β-convertendo:

$$\langle \forall x(man(x) \rightarrow love(x, z_2)),$$
$$(\lambda q.\exists x(woman(x) \wedge q@x), 2)\rangle$$

Resgatando o operador de ligação 2:

$$\langle (\lambda q.\exists x(woman(x) \wedge q@x))@(\forall x(man(x) \rightarrow love(x, z_2)))\rangle$$

α-convertendo e β-convertendo:

$$\langle \exists x(woman(x) \wedge \forall y(man(y) \rightarrow love(y, x)))\rangle$$

Assim, chegamos a uma das duas interpretações: a de que todos os homens amam uma mesma mulher. Se resgatarmos o operador de ligação 2 e só depois resgatarmos o operador de ligação 1, teremos a outra leitura: $\forall y(man(y) \rightarrow \exists x(woman(x) \wedge love(y, x)))$

Portanto, desenvolvemos um método sistemático que pode capturar as ambigüidades de escopo, produzindo as leituras possíveis. O que nos resta agora é a pergunta: será que nosso método é de fato capaz de capturar todas as ambigüidades de escopo? Infelizmente, deve estar claro que não. Iremos apontar duas frases nas quais o método não é suficiente.

2.4.1 Dificuldades

A primeira frase é: "*Every criminal with a gun is dangerous.*" Aplicando nosso método, teremos os seguintes resultados:

1. $\forall x((criminal(x) \wedge \exists y(gun(y) \wedge with(x, y))) \rightarrow smoke(x))$
2. $\exists y(gun(y) \wedge \forall x(criminal(x) \wedge with(x, y) \rightarrow smoke(x)))$

3. $\forall x((criminal(x) \wedge with(x, y)) \rightarrow \exists z(gun(z) \wedge smoke(x)))$

Apesar dos resultados 1 e 2 serem perfeitamente razoáveis, sendo as interpretações que desejávamos, a interpretação 3 possui uma variável livre, não sendo uma sentença de primeira ordem. Isso nos mostra que há um problema com o nosso método. Como isso surgiu?

Realizando nosso procedimento e optando sempre por colocar a representação do sintagma nominal no armazenamento, montaremos a árvore:

Por sua vez, a segunda frase é: "*Every man doesn't love a woman*". A presença da negação traz elementos interessantes. Em primeiro lugar, ela em si é uma fonte possível de ambigüidades de escopo. Entretanto, o método de armazenamento de Cooper não tratou a negação de nenhum modo especial. Além disso, esse exemplo mostra o interesse em manter a operação de armazenamento como opcional. Esta frase pode ser interpretado de seis modos:

1. $\neg\forall x(man(x) \rightarrow \exists y(woman(y) \wedge love(x, y)))$
2. $\neg\exists y(woman(y) \wedge \forall x(man(x) \rightarrow love(x, y)))$
3. $\forall x(man(x) \rightarrow \neg\exists y(woman(y) \wedge love(x, y)))$
4. $\exists y(woman(y) \wedge \neg\forall x(man(x) \rightarrow love(x, y)))$
5. $\forall x(man(x) \rightarrow \exists y(woman(y) \wedge \neg love(x, y)))$
6. $\exists y(woman(y) \wedge \forall x(man(x) \rightarrow \neg love(x, y)))$

Apesar disso, nosso método apenas gerará três desses modos: 3, 5 e 6. Assim, a presença da negação de fato afeta a capacidade de nosso sistema produzir todas as interpretações.

2.5 Armazenamento de Keller

Para lidar especificamente com o primeiro problema do armazenamento de Cooper, Bill Keller propôs uma alteração: permitir armazenamentos aninhados. Assim, cada operador de ligação passa a ser composto não mais por uma fórmula de cálculo lambda e um índice único, mas sim por um armazenamento e um índice único. Isto altera a regra de armazenagem:

Armazenagem (Keller)

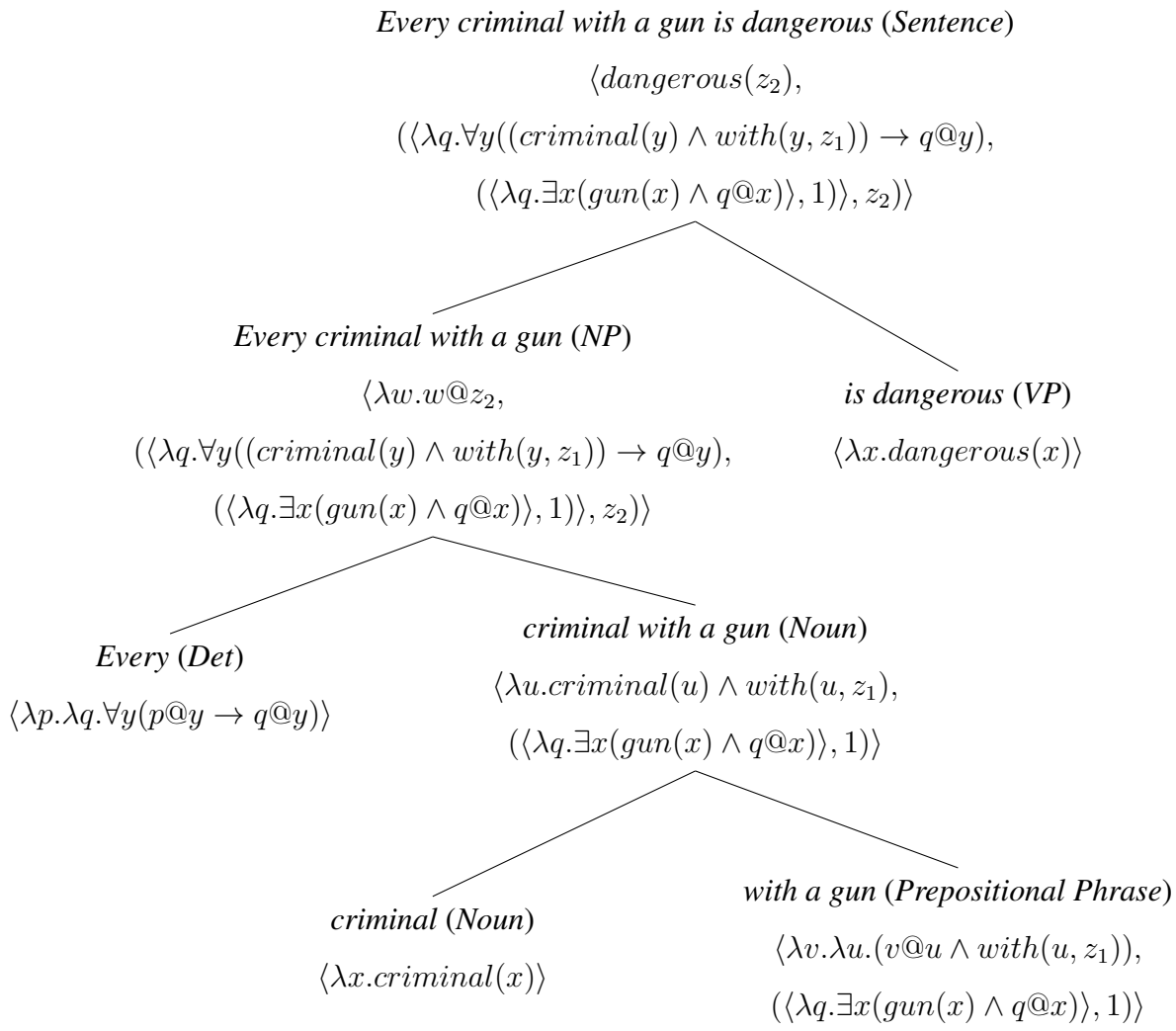
Sendo σ uma seqüência (possivelmente vazia) de operadores de ligação, se o armazenamento $\langle \phi, \sigma \rangle$ é a representação semântica "usual" para um sintagma nominal quantificado, então o armazenamento $\langle \lambda u.(u@z_i), (\langle \phi, \sigma \rangle, i) \rangle$, onde i é um índice único, também é uma representação para este sintagma nominal quantificado.

Por sua vez, também o resgate é alterado. Um operador de ligação só pode ser resgatado para aplicação do núcleo do armazenamento se todos os armazenamentos externos a ele já tiverem sido aplicados. Isto garante que, caso os sintagmas nominais estejam aninhados, então o sintagma nominal mais interno só terá seu operador resgatado após o resgate do sintagma nominal mais externo, evitando o tipo de problema que observamos. Portanto, nossa regra é:

Resgate (Keller)

Sejam σ, σ_1 e σ_2 seqüências (possivelmente vazias) de operadores de ligação. Se o armazenamento $\langle \phi, \sigma_1, ((\beta, \sigma), i), \sigma_2 \rangle$ é uma representação para uma frase (*sentence*), então $\langle (\beta @ \lambda z_i.\phi), \sigma_1, \sigma, \sigma_2 \rangle$ também o é.

Podemos então aplicar isto para o nosso exemplo:



Agora, realizando a extração, podemos fazê-la apenas de um modo: $\exists x(gun(x) \wedge \forall y((criminal(y) \wedge with(y, x)) \rightarrow dangerous(y)))$. Isto é a interpretação correta, não tendo sido gerado nenhum problema. As outras opções de (não-)extração funcionam de modo semelhante.

Assim, o problema dos sintagmas nominais é resolvido. Apesar disso, o segundo problema apontado, do escopo das negações, persiste. As interpretações geradas são as mesmas de antes, pelo armazenamento de Cooper. Portanto, o método de Keller aprimora os resultados de Cooper, sem resolver todas os obstáculos gerados por ambigüidades de escopo.

2.6 *Hole Semantics*

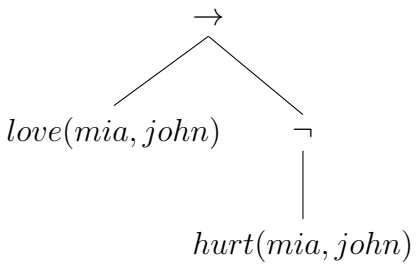
Apesar de ser possível criar um novo mecanismo para capturar a ambigüidade de escopo gerada pela negação , abordagens *ad hoc* para novas dificuldades não são muito

comentar o modo pelo qual eu fiz isso?

desejadas, criando uma falta de harmonização dos métodos usados, possivelmente proliferando uma diversidade de construções muito distintas entre si. Se possível, gostaríamos de possuir uma abordagem mais uniforme. Na realidade, não apenas a negação traz ambigüidades de escopo. Por exemplo, uma frase como "*If a man walks then he jumps and a woman is happy*" é ambígua. Bos (1996) Podemos imaginar uma interpretação na qual "*a woman is happy*" é parte do conseqüente da implicação e outra na qual não o é, sendo uma afirmação separada da implicação.[3] Em razão destas dificuldades, e também de modo a ganhar maior flexibilidade na representação, analisaremos uma outra forma de representação semântica, não baseada em armazenamentos.

Assim como nos métodos baseados em armazenamentos, uma frase não será associada uma expressão de primeira ordem, mas a uma representação abstrata, que então é associada a um conjunto de expressões em primeira ordem. Entretanto, o modo pelo qual isso é feito aqui é distinto. Em *Hole Semantics*, uma idéia essencial é a de *restrições*: podemos pensar na representação como um conjunto de restrições, de modo que qualquer fórmula de primeira ordem que satisfaça as restrições será uma interpretação possível para a frase. (Blackburn e Bos, 2005, p. 129) A representação será referida por representação subespecificada (*USR*, de *underspecified representation*).

Uma fórmula de primeira ordem possui uma decomposição única como uma árvore, em razão pelo modo como é montada. Por exemplo, para a frase "*If Mia loves John then Mia does not hurt John*" tem associada à sua semântica a fórmula $love(mia, john) \rightarrow (\neg hurt(mia, john))$, que pode ser decomposta na árvore:



As restrições serão sobre o modo de construir a fórmula. Dito de outro modo, a USR será um modo de falar a respeito da árvore de cada interpretação possível. Ao invés de montarmos uma única árvore (o que corresponderia a uma única fórmula), a USR pode ser pensada como uma "árvore incompleta", isto é, uma árvore com "buracos", justificando o nome desse método. Entretanto, estes buracos não poderão ser preenchidos de qualquer

[3]Esta construção não ocorre em nosso programa, por não haver orações coordenadas com "*and*".

modo, havendo relações de *dominância*. Um buraco deverá dominar um nó quando estiver acima do mesmo na representação da árvore. A partir daí, as subfórmulas irão compôr a fórmula completa através de um "preenchimento" dos buracos. Este "preenchimento" será feito *encaixando* algum nó (junto com sua sub-árvore) no buraco.

Nos métodos de armazenamentos, a semântica das frases era representada por um vetor que continha um núcleo e os quantificadores guardados em (um aninhamento de) operadores de ligação. Em *Hole Semantics*, nosso modo de representar será bem distinto. Na realidade, nós usaremos uma linguagem lógica para essa representação, chamada *linguagem de representação subespecificada* (*URL*, do inglês *underspecified representation language*). A linguagem original, que aqui é alguma forma de lógica de primeira ordem, será referida por *linguagem de representação semântica* (*SRL*, *semantic representation language*). Pode causar algum espanto o fato de que a URL será, ela própria, uma linguagem de primeira ordem! Seu vocabulário será definido do seguinte modo:

1. Predicados binários :NOT e $\leq$

2. Predicados ternários :IMP, :AND, :OR, :ALL, :SOME e :EQ

3. Cada constante no vocabulário da SRL também é uma constante no vocabulário da URL.

4. Para cada predicado n-ário $pred$ na SRL, :PRED é um predicado $(n+1)$-ário na URL.

A lógica de primeira ordem utilizada é *tipada*, havendo três tipos. O primeiro deles é o dos *buracos*, cujas variáveis serão denotadas por h, h', h_1, h_2, etc. O segundo é o tipo dos *rótulos*, cujas variáveis são escritas l, l', l_1, etc. Cada rótulo marcará um vértice na árvore que não é um buraco, sendo um modo de se referir aos símbolos da SRL. Por fim, o terceiro tipo é o das *meta-variáveis*, escritos v, v', v_1, v_2, etc. As meta-variáveis têm a função de se referir às variáveis da SRL.

Dizemos que algo é um *nó* se for um buraco ou um rótulo. Dizemos que algo é um *meta-termo* da URL caso seja uma *meta-variável* ou uma constante da URL.

Agora, iremos definir as USRs básicas:

1. Se l é um rótulo e h é um buraco, então $l \leq h$ é uma USR básica.

2. Se l é um rótulo e n e n' são nós, então l:NOT(n), l:IMP(n, n'), l:AND(n, n') e l:OR(n, n') são USRs básicas.

3. Se l é um rótulo enquanto t e t' são meta-termos, então l:EQ(t, t') é uma USR básica.

4. Se l é um rótulo, S é um símbolo n-ário na linguagem SRL e $t_1, \ldots, t_n$ são meta-termos, então l:S$(t_1, \ldots, t_n)$ é uma USR básica.

5. Se l é um rótulo, v é uma meta-variável, n é um nó, então l:SOME(v, n) e l:ALL(v, n) são USR básicas.

6. Nada mais é uma USR.

Observe aqui que o espaço a mais criado pela subida de aridade nos predicados e conectivos é preenchido pela variáveis de rótulo. Observe que o item **??** é o único que utiliza o símbolo de $\leq$. USRs básicas desta forma são ditas *restrições de dominância*. Por fim, podemos definir o restante das USRs:

1. Toda USR básica é uma USR.

2. Se ϕ é uma USR e n é um nó, então $\exists n\phi$ é uma USR.

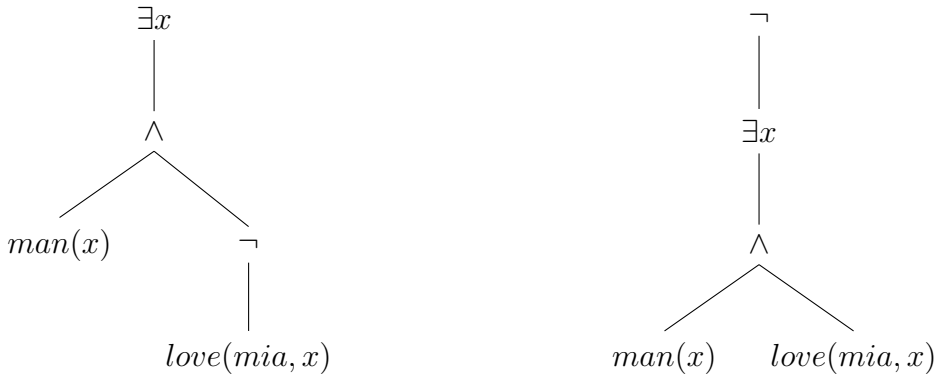
3. Se ϕ é uma USR e v é uma meta-variável, então $\exists v\phi$ é uma USR.

4. se ϕ e ψ são USRs, então $\phi \wedge \psi$ é uma USR.

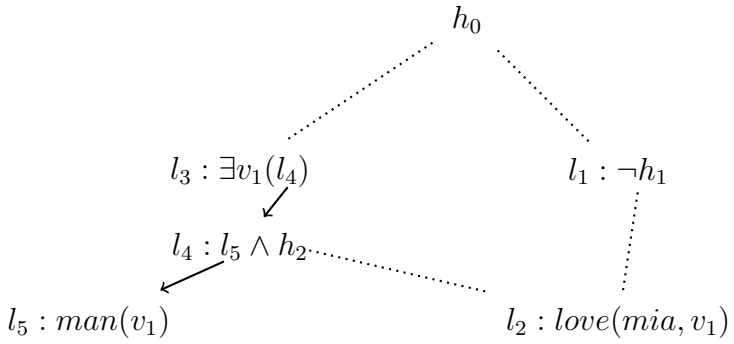
5. Nada mais é uma USR.

É de ser notado que nem todos os conectivos e formas da lógica da primeira ordem foram empregados nesta definição. Na realidade, apenas são fórmulas conjuntos pequenos de formas conjuntivas e existencialmente fechadas. Entretanto, esse fragmento da linguagem é suficiente para nossos propósitos. (Blackburn e Bos, 2005, p. 131)

Podemos avançar então para um exemplo. Consideremos a frase "*Mia does not love a man*". Uma interpretação é aquela na qual Mia não ama um homem específico, que pode ser formalizada como $\exists x : man(x) \wedge \neg love(mia, x)$. Outra, é aquela na qual Mia não ama homem algum, isto é, $\neg(\exists x : man(x) \wedge love(mia, x))$. Suas árvores são:



Já a representação subespecificada busca capturar o que há em comum entre as árvores possíveis. A USR desta frase é: $\exists h_0 \exists h_1 \exists h_2 \exists l_1 \exists l_2 \exists l_3 \exists l_4 \exists l_5 \exists v_1 (l_1 : \text{NOT}(h_1) \wedge l_2 : \text{LOVE}(mia, v_1) \wedge l_3 : \text{SOME}(v_1, l_4) \wedge l_4 : \text{AND}(l_5, h_2) \wedge l_5 : \text{MAN}(v_1))$. Porém, as USRs se tornam melhor compreensíveis através de sua representação gráfica, estando abaixo aquela relativa à nossa frase considerada:



Podemos ver a intuição desta representação. É criado um buraco h_0 correspondente ao nó mais alto da árvore. As linhas pontilhadas representam restrições de dominância entre buracos e nós. Por sua vez, as linhas preenchidas mostram quais nós são pais de outros. A relação de parentesco também representa dominância: se um nó é pai de outro, é certo que o filho não pode ter escopo mais externo que o pai, uma vez que deve ser subfórmula do mesmo. Entretanto, neste caso a posição está fixa: necessariamente a relação de parentesco será aquela. Por sua vez, na dominância entre buracos e nós, não é isto que occore. Basta que o nó dominado esteja no escopo do nó dominante, não necessariamente sendo filho do mesmo. Ou seja, basta ser descendente.

Agora, a nossa análise de frases é feita ainda decompondo sintaticamente, e então, para cada termo, criando uma representação na forma de uma USR. Ainda utilizamos o cálculo lambda para fazer combinações de expressões. A representação final da frase é

feita por combinações das representações das partes que as constituem. Por exemplo, a representação para o determinante "*a*" é: $\lambda x.\lambda y.\lambda h.\lambda l.\exists h_1 \exists l_1 \exists l_2 \exists l_3 \exists v_1 (l2{:}\textsc{all}(v_1, l_3, \wedge l3{:}\textsc{and}(l_1, h_1) \wedge l \leq h_1 \wedge l_2 \leq h \wedge x@v_1@h@l_1 \wedge y@v_1@h@l))$

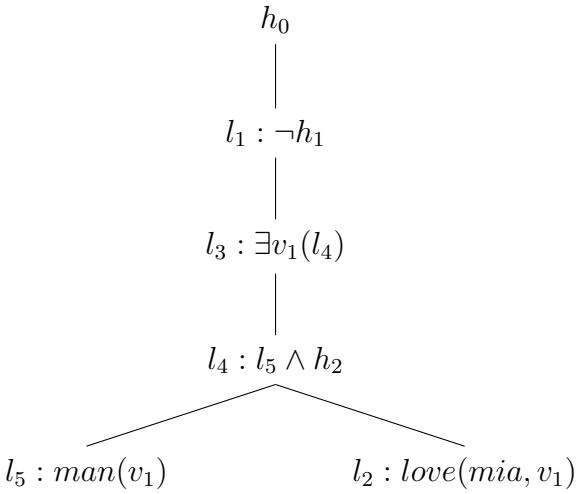
Por sua vez, para o substantivo "*woman*" é: $\lambda v.\lambda h.\lambda l.(l{:}\textsc{woman}(v) \wedge l \leq h)$

Assim, o sintagma nominal *"a woman"* fica, aplicando a segunda representação na primeira e beta-reduzindo: $\lambda y.\lambda h.\lambda l.\exists h_1 \exists l_1 \exists l_2 \exists l_3 \exists v_1 (l2{:}\textsc{all}(v_1, l_3, \wedge l3{:}\textsc{and}(l_1, h_1) \wedge l \leq h_1 \wedge l_2 \leq h \wedge l_1{:}\textsc{woman}(v_1) \wedge l_1 \leq h \wedge y@v_1@h@l))$.

Definindo as USRs para cada função sintática e o modo de combiná-las, obtemos a USR da frase. Com isto em mãos, precisamos ser capazes de construir as árvores possíveis. Isso é feito por meio de *encaixes*[4]. Para cada buraco, achamos um rótulo candidato para preenchê-lo: este rótulo será encaixado no buraco. Mais formalmente, um encaixe é uma função injetiva dos buracos aos rótulos. Entretanto, nem todo encaixe nos satisfaz. Evidentemente, queremos satisfazer duas condições: queremos que o resultados seja uma árvore (portanto, acíclica e conexa), bem como queremos que, se existe uma restrição de dominância de um buraco H sobre um rótulo L (ou seja, se $L \leq H$), então L será descendente de H na árvore.

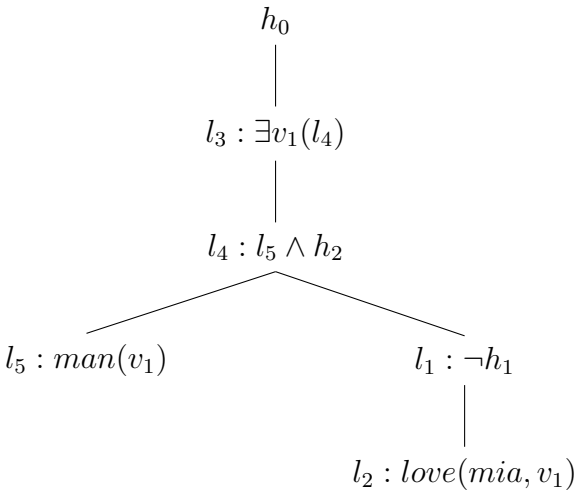
Para o exemplo que vimos, dois encaixes são possíveis: $P_1(h_0) = l_1, P_1(h_1) = l_3, P_1(h_2) = l_2$ e $P_2(h_0) = l_3, P_2(h_1) = l_2, P_2(h_2) = l_1$.

Assim, duas árvores são formadas, cada uma gerando uma interpretação possível. Pelo encaixe P_1 obtemos:



Já pelo P_2, temos:

[4]Em inglês, o termo usado é "*plug*", por isso a letra utilizada é P.



Com efeito, estas são de fato as árvores que havíamos construído antes, para cada interpretação.

3 Inferência

4 Curt – Sistema de Diálogo e Wordnet

De posse dos nossos métodos de representação e de uma gramática capaz de se utilizar deles, precisamos agora extrair informação a partir destas representações. Ou seja, precisamos ser capazes de fazer inferência. Há três importantes tarefas inferenciais que desejamos cumprir.

A primeira é chamada *consulta* (*querying*). Suponha que tenhamos uma representação de como o mundo é – um modelo M. Este modelo pode ter sido construído de diferentes modos: por um banco de dados pré-existente, extraindo dados por meios diversos (como a partir da leitura de imagens) ou mesmo através de um diálogo. Agora, suponha que tenhamos uma *descrição* do mundo, uma afirmação cujo significado pode ser capturado por uma fórmula ϕ. A tarefa de consulta consiste em verificar se esta descrição é verdadeira no mundo; no nosso caso, no modelo de mundo que temos. Portanto, em saber se ϕ é satisfeita em M.

A segunda é a tarefa de *verificação de consistência*. Dizemos que uma afirmação ou discurso é consistente quando "faz sentido", "não é contraditório" ou descreve algo "imaginável" ou "possível". Por exemplo, é claramente inconsistente dizer que "*João é cego e viu Paula atravessar a rua.*", afinal, cegos não são capazes de ver. Essa consistência em linguagem natural pode ser capturada pela noção de satisfabilidade em lógica: uma afirmação é consistente exatamente quando sua representação formal é satisfatível, isto é, quando existe um modelo possível para a representação formal.

Já a terceira tarefa é a de *verificação de informatividade*. Se dissermos que "*Zeus é marido de Hera.*", seria por certo estranho se complementássemos com "*Zeus é casado.*". A informação contida nesta segunda frase já estava presente na primeira, de modo que fazer a nova afirmação é redundante. Dito de outro modo, a segunda frase não é informativa, ainda que isso seja um diálogo possível (se, por exemplo, queremos realçar o fato de Zeus ser casado). De posse da representação lógica, podemos avaliar a informatividade de uma afirmação pela idéia de conseqüência lógica: se o significado da nova afirmação for conseqüência lógica do discurso feito até então, a nova afirmação não é informativa. Formalmente, se nosso discurso até então pode ser representado por uma fórmula ϕ e a afirmação a ser avaliada, pela fórmula ψ, então não-informatividade ocorre quando $\phi \models \psi$. Pelo teorema da dedução, isto equivale a $\models \phi \rightarrow \psi$. Assim, a verificação de informatividade equivale à verificação de validade lógica.

É importante de se notar que as duas últimas tarefas são interdefiníveis. Uma fórmula ϕ é consistente se e somente se $\neg\phi$ é informativa (isto é, logicamente inválida), ao mesmo tempo que ϕ é informativa se e somente se $\neg\phi$ é consistente.

Computacionalmente, a tarefa de consulta é a mais fácil das três. Ela é decidível, o que significa que existem métodos capazes de resolver o problema. Infelizmente, isto não significa que seja tão fácil: na realidade, é um problema da classe de complexidade PSPACE. Já as outras duas tarefas, interdefiníveis, na realidade são *indecidíveis*. Do ponto de vista da tarefa de consistência, uma vez que existe um número infinito de modelos possíveis, a maioria deles sendo infinito, temos um problema de busca computacionalmente complicado. Isto é um fato que teremos de aceitar: nossos métodos não serão capazes de confirmar a consistência e a informatividade em todos os casos. Isto não será um impedimento para nossos exemplos, mas outras abordagens foram desenvolvidas em lógicas menos complexas, algumas decidíveis, que podem ser vistos como fragmentos da lógica de primeira ordem, como lógicas modais ou lógicas de descrição. (Blackburn e Bos, 2005, pp. 50–54)

Para resolver o par difícil de problemas, teremos duas ferramentas: provadores de teorema e construtores de modelos. Provadores de teorema verificam validade: eles são capazes de achar demonstrações e, caso achem a demonstração de uma fórmula, está será válida. Entretanto, eles nada dizem sobre invalidade. Por sua vez, construtores de modelo tentam encontrar um modelo que satisfaça a fórmula a ele apresentada. Contudo, a incapacidade de encontrar um modelo não garante que tal modelo não exista (por exemplo, o modelo pode ser infinito ou maior do que considerado pelo construtor). Estes instrumentos são utilizados dos seguinte modo: para uma fórmula ϕ, caso uma prova seja achada, saberemos que ela é válida, o que significa que ela é não-informativa, bem como que $\neg\phi$ é insatisfatível (e assim, inconsistente). Por sua vez, caso um modelo para ϕ seja achado, saberemos que ela é satisfatível (consistente) e que $\neg\phi$ não é válida, isto é, informativa.

Assim, tome θ como representativo do discurso atual. Seja ϕ a representação da nova informação. Caso um provador consiga mostrar a validade de $\theta \rightarrow (\neg\phi)$, então a nova afirmação não será consistente com o discurso anterior. Por sua vez, caso consiga mostrar que $\theta \rightarrow \phi$, a nova afirmação não será informativa.

De outro lado, caso um construtor de modelos consiga encontrar um modelo para $\theta \wedge \phi$, então a nova afirmação é consistente com o discurso. Já se encontrar um modelo

para $\theta \wedge (\neg\phi)$, a afirmação será informativa.

Isto é, provadores de teorema podem fornecer respostas negativas para o teste de informatividade e para o teste de consistência, enquanto construtores de modelos podem fornecer respostas positivas. Neste trabalho, usamos um provador de teoremas e um construtor de modelos prontos, chamados de OTTER e MACE, respectivamente.

colocar referência a eles?

Estabelecidas as tarefas que desejamos cumprir e como realizá-las com nossas representações semânticas, apresentaremos como fazer isto através do sistema Curt, de Blackburn e Bos (2005). Curt significa "*clever use of reasoning tools*" ("*uso esperto de ferramentas de raciocínio*"). É um sistema no qual o usuário pode fazer afirmações em inglês que serão avaliadas pelo programa. Ele será capaz de fazer nossas tarefas inferenciais, notificando caso haja algum problema, bem como de construir modelos das informações passadas e de responder algumas perguntas simples.

4.1 Arquitetura do Curt

O Curt integra as ferramentas de representação com as de inferência. A arquitetura da leitura e representação é como descrevemos na seção 2.2. Já a tarefa de inferência é principalmente organizada pelo arquivo CALLINFERENCE.PL, colocando o problema em um formato lido pelo provador de teoremas e pelo construtor de modelos.

O Curt é um sistema de diálogo rudimentar, no qual o usuário pode fazer afirmações em linguagem natural a serem avaliadas. A cada afirmação, Curt encontra os sentidos possíveis para a frase e tenta integrá-la ao restante de sua leitura do diálogo. Esta é uma fórmula de primeira ordem: quando uma nova frase é dita, para cada interpretação possível é formada uma nova leitura pela conjunção da leitura anterior com a interpretação da nova frase. Além disso, são encontrados modelos possíveis para cada leitura. Posteriormente, todas as leituras alternativas são descartadas, sendo mantida uma única. Do mesmo modo, apenas é mantido um modelo.

Alguns comandos são possíveis, como **`history`**, que apresenta a lista das afirmações feitas até então; **`readings`**, que apresenta as leituras possíveis do diálogo até então; **`new`**, que recomeça o diálogo, **`select`**, que permite a escolha de uma das leituras possíveis para ser mantida pelo sistema e **`models`**, que mostra os modelos construídos para o diálogo.

Nossas ferramentas de inferência são usadas no modo pelo qual Curt lê criticamente

as afirmações feitas. Caso uma leitura construída seja inconsistente, ela é descartada. Não havendo nenhuma leitura consistente, o sistema reclamará que não acredita na afirmação feita. Por outro lado, também é avaliada a informatividade. Caso a afirmação sendo feita não seja informativa em relação ao diálogo até então, Curt reclamará que a afirmação feita é óbvia.

Mostremos um exemplo no SensitiveCurt, um modelo de Curt que ainda não utiliza conhecimento preliminar. Por hora, não discutiremos os testes de consistência e informatividade.

```
> John is a gorilla

Message (consistency checking): mace found a result.
Message (informativity checking): mace found a result.
Curt: OK.

> readings

1 some A (gorilla102480855(A) & john = A)

> models

1 D=[d1]
  f(0,c1,d1)
  f(1,gorilla102480855,[d1])
  f(0,john,d1)


> John eats a banana

Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Message (informativity checking): mace found a result.
Message (informativity checking): mace found a result.
```

```
Curt: OK.

> readings

1 (some A (gorilla102480855(A) & john = A) & some B (banana107753592(B)
2 (some A (gorilla102480855(A) & john = A) & some B (banana112352287(B)

> models

1 D=[d1]
  f(0,c1,d1)
  f(1,gorilla102480855,[d1])
  f(0,john,d1)
  f(0,c2,d1)
  f(1,banana107753592,[d1])
  f(2,eat,[ (d1,d1)])

2 D=[d1]
  f(0,c1,d1)
  f(1,gorilla102480855,[d1])
  f(0,john,d1)
  f(0,c2,d1)
  f(1,banana112352287,[d1])
  f(2,eat,[ (d1,d1)])
```

Nosso sistema identificou dois sentidos possíveis para "*banana*". Consultando a glosa, veremos que um dos sentidos é a fruta, enquanto o outro sentido é a bananeira, a árvore de banana. O sentido que queremos, de fruta, é **banana107753592** Entretanto, o que é estranho nos dois modelos formados é que há apenas um elemento no domínio, que é tanto *John* quanto a banana! Nosso sistema é incapaz de dizer que nenhum gorila é uma banana. Teremos de dizer isto a ele:

```
> No gorilla is a banana

Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Message (informativity checking): mace found a result.
Message (informativity checking): mace found a result.
Message (informativity checking): mace found a result.
Message (informativity checking): mace found a result.
Curt: OK.

> readings

1 and(and(some(A,and(gorilla102480855(A),eq(john,A))),some(B,and(banana
2 and(and(some(A,and(gorilla102480855(A),eq(john,A))),some(B,and(banana
3 and(and(some(A,and(gorilla102480855(A),eq(john,A))),some(B,and(banana
4 and(and(some(A,and(gorilla102480855(A),eq(john,A))),some(B,and(banana
```

Selecionaremos aqui a terceira interpretação, pois possui o escopo correto (a negação está fora do escopo do quantificador existencial) e usa a interpretação correta de banana, com o *synset* 107753592.

```
> select 3

> models

1 D=[d1,d2]
  f(0,c1,d1)
  f(1,gorilla102480855,[d1])
  f(0,john,d1)
  f(0,c2,d2)
  f(1,banana107753592,[d2])
```

```
  f(2,eat,[ (d1,d2)])
```

Desse modo, Curt agora mostrou um modelo em que, de fato, há duas entidades: o gorila *John* e a banana comida pelo mesmo.

Vejamos agora outro exemplo, para ilustrar o uso dos testes de consistência e informatividade.

Diremos inicialmente que todo crocodilo é réptil.

```
> Every crocodile is a reptile

Message (consistency checking): mace found a result.
Message (consistency checking): mace found a result.
Message (informativity checking): mace found a result.
Message (informativity checking): mace found a result.
Curt: OK.

> readings

1 some A (reptile101661091(A) & all B (crocodile101697178(B) > B = A))
2 all A (crocodile101697178(A) > some B (reptile101661091(B) & A = B))
```

Perceba que a segunda leitura é a que queremos, pois a primeira é a de que há um réptil específico que é todo crocodilo, o que é absurdo. Assim, selecionaremos a segunda leitura.

```
> select 2

> Paul is a crocodile

Message (consistency checking): mace found a result.
Message (informativity checking): mace found a result.
Curt: OK.
```

```
> No reptile is Paul

Message (consistency checking): otter found a result.
Curt: No! I do not believe that!
```

Assim, dizendo que *Paul* é um crocodilo, Curt percebe o absurdo de dizer que Paul não é um réptil e enfaticamente reclama.

Por outro lado:

```
> If John is a crocodile then John is a reptile.

Message (consistency checking): mace found a result.
Message (informativity checking): otter found a result.
Curt: Well, that is obvious!
```

Dizer que se *John* é um crocodilo garante que *John* é um réptil também é algo que Curt reclama: o sistema já sabia disso, por saber que todo crocodilo é um réptil, apontando portanto a obviedade da afirmação.

O que desejamos é que Curt seja capaz de fazer este tipo de inferência sem que tenhamos de ensiná-lo que todo crocodilo é um réptil. Este é um conhecimento sobre o mundo que Curt poderia já ter. De fato, uma vez integrado à Wordnet, teremos uma consulta deste modo:

```
> John is a crocodile

Message (consistency checking): mace found a result.
Message (informativity checking): mace found a result.
Curt: OK.

> John is a reptile

Message (consistency checking): mace found a result.
Message (informativity checking): otter found a result.
Curt: Well, that is obvious!
```

```
> models

1 D=[d1]
  f(1,physicalentity100001930,[d1])
  f(1,entity100001740,[d1])
  f(1,physicalobject100002684,[d1])
  f(1,object100002684,[d1])
  f(1,unit100003553,[d1])
  f(1,whole100003553,[d1])
  f(1,animatething100004258,[d1])
  f(1,livingthing100004258,[d1])
  f(1,being100004475,[d1])
  f(1,organism100004475,[d1])
  f(1,fauna100015388,[d1])
  f(1,animal100015388,[d1])
  f(1,animatebeing100015388,[d1])
  f(1,beast100015388,[d1])
  f(1,brute100015388,[d1])
  f(1,creature100015388,[d1])
  f(1,chordate101466257,[d1])
  f(1,diapsidreptile101661818,[d1])
  f(1,diapsid101661818,[d1])
  f(1,reptile101661091,[d1])
  f(1,reptilian101661091,[d1])
  f(1,crocodilian101696633,[d1])
  f(1,crocodilianreptile101696633,[d1])
  f(1,craniate101471682,[d1])
  f(1,vertebrate101471682,[d1])
  f(1,crocodile101697178,[d1])
  f(0,c1,d1)
  f(0,john,d1)
```

```
f(0,c2,d1)
```

Apenas pela afirmação de que *John* é um crocodilo, Curt construirá um modelo com tudo que sabe sobre crocodilos: *John*, o crocodilo, é necessariamente um réptil, é vertebrado, é um animal, é um organismo e uma entidade física.

4.2 Integrando à Wordnet

A Wordnet considera quatro categorias sintáticas de palavras: substantivos, verbos, adjetivos e advérbios. (Fellbaum, 1998, p. 5) Dentro de cada categoria, os conceitos são distribuídos em conjuntos de sinônimos (*synsets*, de "*synonym sets*"). Entre estes são definidas relações semânticas, como hiponímia (relação de subconjunto), meronímia (relação de ser parte) e implicação. Felizmente, a Wordnet pode ser baixada em um formato já preparado par a linguagem Prolog.

Por exemplo, o arquivo WN_S.PL apresenta um predicado no formato

```
s(synset_id,w_num,"word",ss_type,sense_number,tag_count).
```

Cada cláusula corresponde a um significado de uma palavra. Os argumentos relevantes para nossa análise são o **synset_id**, que identifica a qual synset aquele sentido da palavra pertence, **w_num**, que diz qual o número daquela palavra dentro do synset, **"word"**, que diz qual palavra é e **ss_type**, que diz a qual categoria sintática aquele synset pertence.

Um pequeno trecho do arquivo é este:

```
s(104565375,1,'weapon',n,1,29).
s(104565375,2,'arm',n,3,1).
s(105563770,1,'arm',n,1,104).
```

Notamos aqui a palavra "*arm*" tendo um sentido que pertence ao *synset* 104565375, que é o mesmo de um sentido de *weapon*, bem como tendo um outro sentido que pertence ao *synset* 105563770. Consultando o arquivo WN_G.PL, temos a glosa sobre cada *synset*, isto é, um comentário. O trecho relativo é:

```
g(104565375,'any instrument or instrumentality used in fighting
        or hunting; ''he was licensed to carry a weapon"').
```

```
g(105563770,'a human limb; technically the part of the superior limb
        between the shoulder and the elbow but commonly
        used to refer to the whole superior limb').
```

Ou seja, os "*synsets*" correspondem, respectivamente, às palavras "*arma*" e "*braço*", que são dois sentidos possíveis de "*arm*". Um outro exemplo é:

```
s(110388924,1,'owner',n,1,15).
s(110388924,2,'proprietor',n,1,11).
s(110389398,1,'owner',n,2,9).
s(110389398,2,'possessor',n,1,0).
```

A palavra "*owner*" possui ao menos dois sentidos: o de "*proprietor*", representado no *synset* 110388924, bem como o de "*possessor*", representado no *synset* 110389398. Esta diferenciação também existe no português, onde "*dono*" pode significar tanto "*proprietário*" quanto "*possuidor*".

Há dois pontos distintos no qual precisamos integrar a Wordnet à nossa arquitetura. O primeiro é na extensão do vocabulário, isto é, fazer com que nosso leitor de entradas seja capaz de aceitar frases que, respeitando as regras sintáticas entendidas pelo mesmo, usem palavras contidas na lista da Wordnet. O segundo ponto é na utilização das relações semânticas entre *synsets* para complementar o conhecimento prévio do Curt. Iremos especificar como realizamos cada passo.

Para o primeiro problema, um modo natural na nossa arquitetura é inserir no arquivo ENGLISHLEXICON.PL a conexão com a Wordnet. Tal arquivo é onde as entradas do léxico estão especificadas, de modo que poderíamos adicionar uma cláusula que considera como entradas do léxico palavras presentes na Wordnet. De fato, este foi o primeiro modo que implementamos, inserindo no arquivo a seguinte cláusula:

```
lexEntry(noun,[symbol:Sym,syntax:Syn]) :-
    Ss_type = n,
    s(Synset,_,Word,Ss_type,_,_),
    downcase_atom(Word,Word2),
    atomic_list_concat(Syn,' ',Word2),
```

```
checkWords([Word2],[Expression]),
atom_concat(Expression,Synset,Sym).
```

Explicaremos com mais cuidado. Dizemos que existe uma entrada no léxico de um substantivo (*noun*) com símbolo Sym (isto é, representação semântica usando o símbolo Sym) e sintaxe Syn (isto é, aparecendo no texto na forma Syn) quando as condições no corpo da cláusula são satisfeitas.

Em primeiro lugar, o tipo de entrada na Wordnet deve ser um substantivo, portanto **`Ss_type = n`**. Depois, em **`s(Synset,_,Word,Ss_type,_,_)`**, tentamos achar um candidato. Verificamos a sintaxe. Na Wordnet, as palavras não estão pré-processadas do modo que queremos: em minúsculo e quebrando expressões de mais de uma palavra em listas. Por exemplo, "$human\ foot$" deve ficar no formato $[human, foot]$. Os predicados **`downcase_atom/2`** e **`atomic_list_concat/2`** fazem este papel, construindo a representação sintática. Quanto ao símbolo semântico, usamos o **`checkWords`** também para normalizar a expressão (desta vez em uma única string) e concatenamos o resultado com o número do "*synset*". Perceba que isto garante uma expressão única na nossa linguagem lógica para um sentido específico de uma palavra.

Por exemplo, veja o código abaixo e o que nos é retornado:

```
?- lexEntry(noun,[symbol:Sym,syntax:[dinosaur]]).
Sym = dinosaur101699831
```

Em uma consulta (por exemplo, usando HOLESEMANTICS.PL), obtemos:

```
?- holeSemantics.
> Vincent is a dinosaur

1 some(A,and(hole(A),some(B,and(label(B),some(C,some(D,
some(E,some(F,some(G,and(hole(C),and(label(D),and(label(E),
and(label(F),and(some(E,G,F),and(and(F,D,C),and(leq(B,C),
and(leq(E,A),and(and(pred1(D,dinosaur101699831,G),leq(D,A)),
and(eq(B,G,vincent),leq(B,A))))))))))))))))))))))))

[plug(C,B),plug(A,E)]
```

```
1 some(G,and(dinosaur101699831(G),eq(G,vincent)))
```

Apesar de cumprir o objetivo e adequado à arquitetura, infelizmente este método não é adequado: ele é extremamente lento. Investigando, a causa não é de todo surpreendente: a decomposição sintática é feita testando possibilidades. Encontrando um candidato a substantivo, o único modo não só de achar a entrada léxica, caso seja, mas de refutar essa possibilidade é percorrendo toda a lista da Wordnet. Isto faz com que a decomposição sintática seja imensamente demorada, com várias consultas ao banco de dados léxicos, que é extenso. Assim, desenvolvemos uma abordagem alternativa.

O lampejo está em se pensar quantas consultas à Wordnet são necessárias para a leitura de uma frase. Podemos imaginar um modo de, conhecida a lista de palavras da frase (normalmente, uma lista bastante pequena), identificar os synsets necessários percorrendo a Wordnet apenas uma vez: basta avançar na Wordnet, verificando a cada etapa se a palavra é igual à forma sintática desejada. Sendo igual, podemos criar uma entrada no léxico com esta informação. [5]

Podemos implementar isso. Inicialmente, em ENGLISHLEXICON.PL, declaramos o predicado **`lexEntry/2`** como dinâmico. Modificaremos então o arquivo KELLERSTORAGE.PL (o mesmo pode ser feito em HOLESEMANTICS.PL, depende de qual forma de representação será usada para encontrar as leituras possíveis):

```
kellerStorage:-
   readLine(Sentence),
   wordnetLexicon(Sentence,_),
   setof(Sem,t([sem:Sem],Sentence,[]),Sems1),
   filterAlphabeticVariants(Sems1,Sems2),
   printRepresentations(Sems2).
```

A única linha que adicionamos foi **`wordnetLexicon(Sentence,_),`**. Com isso, nosso léxico será atualizado a depender da frase lida, pela Wordnet. Descreveremos como isso é feito:

[5] Um problema deste método é que, do modo descrito, não captura adequadamente palavras compostas. Entretanto, soluções podem ser pensadas, como utilizar não apenas as palavras individualmente, mas também pares de palavras adjacentes, na ordem da frase.

```
wordnetLexicon(L,SymList) :-
    findall(Sym,findWordnetLex(L,Sym),SymList).

findWordnetLex(L,Sym) :-
    s(Synset,_,Expression,n,_,_),
    member(Expression,L),
    atom_concat(Expression,Synset,Sym),
    Syn = Expression,
    assert(lexEntry(noun,[symbol:Sym,syntax:[Syn]])).
```

O primeiro predicado garante que serão consideradas todas as possibilidades de leitura das palavras com base na Wordnet. Já o segundo, inicialmente encontra uma palavra na Wordnet que seja um substantivo e um significado possível para ela, pelo predicado **`s/6`**. Verifica-se se a palavra está presente na frase sendo lida. Caso o seja, então é inserido no léxico a entrada para a mesma, a partir do synset. Por exemplo, se a palavra "*bear*" estiver presente em uma frase, será inserido o predicado [6]

```
lexEntry(noun,[symbol:bear102131653,syntax:[bear]]).
```

Agora usaremos a Wordnet para geração de conhecimento prévio. Faremos primeiro para a relação de *sinonímia*, isto é, palavras de mesmo significado.

Por exemplo, no *synset* 100064151, temos as expressões "*blockbuster*", "*megahit*", "*smash hit*". Esse *synset* representa o significado de algo de sucesso e popularidade, como um filme ou peça. [7]

Queremos então que nosso sistema detecte "*Titanic is a megahit.*" como equivalente a "*Titanic is a blockbuster*". Pensamos então em duas alternativas: Uma primeira opção é fazer as duas palavras corresponderem à mesma expressão lógica. Por exemplo, a $blockbuster(titanic)$ ou a $p100064151(titanic)$. Utilizar uma palavra específica para representar um *synset* não é uma boa alternativa, pois teríamos de escolher palavras não ambíguas ou tomar o cuidado de não representar dois *synsets* distintos pela mesma palavra. Por outro lado, utilizar algo como $p100064151$ como predicado faz com que

[6] O *synset* 102131653 representa o animal urso. Um outro sentido para a palavra na Wordnet é de um investidor pessimista.

[7] De fato, a glosa é "*an unusually successful hit with widespread popularity and huge sales (especially a movie or play or recording or novel)*".

a interpretação humana das fórmulas criadas seja mais trabalhosa (tendo de consultar a glosa), então não foi a forma que escolhemos.

Fizemos de outro modo. Associamos a cada significado de cada palavra um *synset*. Assim, "*Titanic is a megahit.*" fica $megahit100064151(titanic)$ e "*Titanic is a blockbuster*", $blockbuster100064151(titanic)$. Sendo predicados distintos, não há ainda uma relação lógica entre eles. Isto obriga que adicionemos explicitamente como axioma que $\forall x(megahit100064151(x) \leftrightarrow blockbuster100064151(x))$. Uma desvantagem de tal método é que cria um número maior de fórmulas a serem adicionadas no conhecimento prévio, o que pode prejudicar a computação. Por outro lado, torna as formulações mais legíveis e explícitas para um humano. Este é o modo pelo qual implementaremos.

Para dizermos que dois predicados são sinônimos, como $megahit100064151$ e $blockbuster100064151$, mas principalmente para encontrar os sinônimos dado um predicado, usamos a regra abaixo.

```
synonym(Sym1,Sym2) :-
    atom_concat(E1,Synset,Sym1),
    atom_number(Synset,SynsetNum),
    s(SynsetNum,_,Word,_,_,_),
    checkWords([Word],[E2]),
    \+ E1 = E2,
    atom_concat(E2,Synset,Sym2).
```

Nas duas primeiras linhas do corpo da regra, decompomos o primeiro predicado em sua parte léxica e em seu número do *synset*. Depois, procuramos uma palavra na Wordnet com o mesmo *synset*. Usamos **checkWords/2** apenas para normalizar a palavra (retirar espaços, colocar em caixa baixa e retirar símbolos diversos) e então conferimos se a palavra é distinta da palavra inserida. Se sim, então construímos o seguindo predicado pela concatenação da palavra com o *synset*. Com isto, podemos encontrar todos os predicados sinônimos.

Para a criação das regras, usamos:

```
wordnetKnowledge(Sym,Arity,Axiom) :-
    synonym(Sym,Sym2),
    (
```

```
        Arity = 0, Axiom = eq(Sym,Sym2)
    ;
        Arity = 1, F1 =.. [Sym,X], F2 =.. [Sym2,X],
        Axiom = all(X,and(imp(F1,F2),imp(F2,F1)))
    ;
        Arity = 2, F1 =.. [Sym,X,Y], F2 =.. [Sym2,X,Y],
        Axiom = all(X,all(Y,and(imp(F1,F2),imp(F2,F1))))
    ).
```

A divisão por aridade já era utilizada nas outras formas de conhecimento prévio no arquivo BACKGROUNDKNOWLEDGE.PL, tendo sido útil seguí-la. Inicialmente encontramos um sinônimo. Caso a expressão que desejamos consultar seja de aridade 0, isto é, uma constante, então o a regra a ser adicionada é a igualdade entre as constantes. Caso seja um predicado unário, basta dizer que para todo argumento, satisfazer um dos predicados implica satisfazer o outro. Caso seja binário [8], será o mesmo, mas para todo par de argumentos.

Colocamos tais regras no arquivo WORDNETKNOWLEDGE.PL, o relacionamos com o BACKGROUNDKNOWLEDGE.PL (inserindo entre as outras formas de conhecimento prévio) e então estará construído o conhecimento prévio de sinonímia.

A relação de hiponímia trata da idéia de subclasse: um conceito é dito hipônimo de outro quando o primeiro é mais específico que o do segundo. Por exemplo, "*banana*" é um hipônimo de "*fruta*", pois toda banana é uma fruta. Equivalentemente, dizemos que "*fruta*" é um hiperônimo de "*banana*".

A Wordnet traz a relação de hiponímia no arquivo WN_HYP.PL, definida tanto para substantivos quanto para verbos. Um exemplo é o seguinte:

```
?- s(X,_,angel,_,_,_), g(X,GX), hyp(X,Y),
g(Y,GY), findall(Z,s(Y,_,Z,_,_,_),SETZ).

X = 109538915,
GX = 'spiritual being attendant upon God',
Y = 109504135,
```

[8] Tanto neste trabalho quanto em Blackburn e Bos (2005), não são consideramos predicados com mais do que 2 argumentos.

```
GY = 'an incorporeal being believed to have powers
         to affect the course of human events',
SETZ = ['spiritual being', 'supernatural being'] .
```

O predicado de hiponímia é **hyp/2**, indicando que o primeiro argumento é hipônimo do segundo. Neste caso, o *synset* 109538915 é hipônimo de *109504135*. De acordo com a glosa e com as palavras consultadas, isto significa que "*angel*", no sentido de "*um ser espiritual a serviço de Deus*", é um hipônimo de "*um ser espiritual*".

Podemos então verificar como implementamos isto no sistema Curt:

```
hypernym(Sym1,Sym2) :-
    atom_concat(E1,Synset,Sym1),
    atom_number(Synset,SynsetNum),
    hyp(SynsetNum,SynsetHyp),
    s(SynsetHyp,_,Word,_,_,_),
    checkWords([Word],[E2]),
    \+ E1 = E2,
    atom_concat(E2,SynsetHyp,Sym2).
```

Esta regra encontra hiperônimos. Assim como a regra de achar sinônimos, utilizamos aqui um predicado da linguagem lógica, **Sym1**, na forma palavra-número. Precisamos decompô-lo, achar um hiperônimo e então construir o novo predicado do hiperônimo.

Para o conhecimento preliminar, usamos:

```
wordnetKnowledge(Sym,Arity,Axiom) :-
    hypernym(Sym,Sym2),
    (
        Arity = 1, F1 =.. [Sym,X], F2 =.. [Sym2,X],
        Axiom = all(X,imp(F1,F2))
    ;
        Arity = 2, F1 =.. [Sym,X,Y], F2 =.. [Sym2,X,Y],
        Axiom = all(X,all(Y,imp(F1,F2)))
    ).
```

Em primeiro lugar, é de se notar que aqui não aceitamos aridade 0, apenas 1 ou 2. Em ambos os casos, inserimos no conhecimento prévio a afirmação de que o hipônimo implica no hiperônimo, que é tudo que precisamos.

Para o uso do predicado **`wordnetKnowledge`**, transformamos o arquivo WORDNETKNOWLEDGE.PL em um módulo, modificamos o arquivo BACKGROUNDKNOWLEDGE.PL. Para isto, basta chamar o módulo e colocar o conhecimento prévio da Wordnet entre os outros no predicado **`computeBackgroundKnowledge/2`**:

```
findall(_,(memberList(symbol(Symbol,Arity),Symbols),
           wordnetKnowledge(Symbol,Arity,F),
           assert(knowledge(F))),_),
```

Uma diferença no que fizemos em relação aos outros tipos de conhecimento prévio é que analisamos a lista dos símbolos até então antes do processamento da Wordnet, o que é necessário, uma vez que nosso conhecimento prévio é construído com base nesta lista.

Com isso, está completa nossa integração com a Wordnet em relação aos substantivos, para as relações de sinonímia e de hiponímia.

modificar esse trecho, a depender de se colocarei outras implementações no texto ou não

4.3 Helpful Curt

Podemos usar o Curt mais avançado, Helpful Curt, a fim de utilizar seu mecanismo de responder perguntas. Podem ser respondidas por ele perguntas ditas *wh-questions*, como "*who*" ("quem"), "*what*" ("o quê") e "*which*" ("qual"). A técnica utilizada envolve utilizar uma representação quase-lógica para as perguntas: uma vez que perguntas não possuem valor-verdade (não sendo declarativas), não podemos atribuir a elas uma verdadeira expressão lógica. A técnica utilizada no Helpful Curt é uma de lacunas, mas não entraremos aqui em mais detalhes a respeito dela ou da semântica de questões. Para mais informações sobre estes tópicos, leitor pode consultar o original (Blackburn e Bos, 2005, pp. 293–300, 303–304) e os trabalhos referenciados no mesmo.

5 Conclusão

6 Referências

Patrick Blackburn e Johan Bos. *Representation and Inference for Natural Language. A First Course in Computational Semantics*. CSLI, 2005.

Johan Bos. Predicate logic unplugged. In *In Proceedings of the 10th Amsterdam Colloquium*, pages 133–143, 1996.

Johan Bos e Katja Markert. Recognising textual entailment with logical inference. In *Proceedings of the Conference on Human Language Technology and Empirical Methods in Natural Language Processing*, HLT '05, pages 628–635, Stroudsburg, PA, USA, 2005. Association for Computational Linguistics. doi: 10.3115/1220575.1220654. URL http://dx.doi.org/10.3115/1220575.1220654.

Johan Bos e Katja Markert. Recognising textual entailment with robust logical inference. In *Proceedings of the First International Conference on Machine Learning Challenges: Evaluating Predictive Uncertainty Visual Object Classification, and Recognizing Textual Entailment*, MLCW'05, pages 404–426, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33427-0, 978-3-540-33427-9. doi: 10.1007/11736790_23. URL http://dx.doi.org/10.1007/11736790_23.

Ido Dagan, Oren Glickman, e Bernardo Magnini. The pascal recognising textual entailment challenge. In *Proceedings of the First International Conference on Machine Learning Challenges: Evaluating Predictive Uncertainty Visual Object Classification, and Recognizing Textual Entailment*, MLCW'05, pages 177–190, Berlin, Heidelberg, 2006. Springer-Verlag. ISBN 3-540-33427-0, 978-3-540-33427-9. doi: 10.1007/11736790_9. URL http://dx.doi.org/10.1007/11736790_9.

Ido Dagan, Dan Roth, Fabio Zanzotto, e Graeme Hirst. *Recognizing Textual Entailment*. Morgan & Claypool Publishers, 2012. ISBN 1598298348, 9781598298345.

Jan van Eijck e Christina Unger. *Computational Semantics with Functional Programming*. Cambridge University Press, New York, NY, USA, 1st edição, 2010. ISBN 0521760305, 9780521760300.

Christiane Fellbaum. Introduction. In *WordNet: An Electronic Lexical Database*, Language, Speech, and Communication. MIT, 1998. ISBN 026206197X,9780262061971.

Ralph Grishman. *Computational Linguistics: An Introduction*. Studies in Natural Language Processing. Cambridge University Press, first edição, 1986. ISBN 0521325021,9780521325028.

Daniel Jurafsky e James H. Martin. *Speech and Language Processing (2nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009. ISBN 0131873210.

Hans Kamp. Foreword. *in Computational Semantics with Functional Programming*; Jan van Eijck and Christina Unger, 2010.

Hans Kamp e Uwe Reyle. *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Studies in Linguistics and Philosophy 42. Springer Netherlands, 1 edição, 1993. ISBN 978-0-7923-1028-0,978-94-017-1616-1.

Christopher D. Manning e Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, 1999. ISBN 0-262-13360-1.

Peter Markie. Rationalism vs. empiricism. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Summer 2015 edição, 2015.

Noah Smith. NLP/CL at Carnegie Mellon. URL `http://www.cs.cmu.edu/~nasmith/nlp-cl.html`.