Luiz Arthur Pagani

IMPLEMENTAÇÃO EM PROLOG DE ANALISADORES GRAMATICAIS PARA ALGUMAS LÍNGUAS DE **INTRODUCTION TO MONTAGUE SEMANTICS**

UNICAMP

2001

Luiz Arthur Pagani

IMPLEMENTAÇÃO EM PROLOG DE ANALISADORES GRAMATICAIS PARA ALGUMAS LÍNGUAS DE **INTRODUCTION TO MONTAGUE SEMANTICS**

Tese apresentada ao curso de Lingüística do Instituto de Estudos da Linguagem da Universidade Estadual de Campinas como requisito parcial para obtenção do título de Doutor em Lingüística

Orienntador: Prof. Dr. Edson Françozo

Campinas
Instituto de Estudos da Linguagem
2001

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA IEL - UNICAMP

Pagani, Luiz Arthur
P14i Implementação em Prolog de analisadores gramaticais para algumas línguas de **Introduction to Montague Semantics** / Luiz Arthur Pagani. - - Campinas, SP: [s.n.], 2001.

Orientador: Edson Françozo
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Estudos da Linguagem.

1. Semântica. I. Françozo, Edson. II. Universidade Estadual de Campinas. Instituto de Estudos da Linguagem. III. Título.

Prof. Dr. Edson Françozo = Orientador

Prof. Dr. Rodolfo Ilari

Prof. Dr. José Borges Neto

Prof. Dr. Jacques Wainer

Prof. Dr. Elias Humberto Alves

Este exemplar corresponde à redação final da tese defendida por Luiz Arthur Pagani

e aprovada pela Comissão Julgadora em 05/02/2002.

Esta tese é dedicada a Cristiane Branco Paiva Pagani e a Ana Luísa Paiva Pagani, com todo o carinho que elas merecem.

Agradecimentos

Em todas as suas etapas, esta tese deve bastante a muitas pessoas e instituições, sem as quais o seu projeto não teria sequer iniciado.

Começando pelos agradecimentos institucionais, devo um agradecimento ao programa de pós-graduação em Lingüística do Instituto de Estudos da Linguagem, da Universidade Estadual de Campinas, e em especial aos professores e aos meus colegas desse programa. No ambiente de estudos e pesquisas oferecido por este programa sempre pude encontrar estímulo e compreensão para enfrentar os desafios estabelecidos pelo doutoramento.

Ainda no Instituto de Estudos da Linguagem, tive a felicidade de participar de várias atividades desenvolvidas no Laboratório de Fonética Acústica e Psicolingüística Experimental (LAFAPE). Assim, agradecendo a Eleonora Cavalcante Albano, coordenadora do LAFAPE, espero estar também agradecendo a todos os seus membros. No entanto, merecem lembrança especial Paula Lens Costa Lima, João Luís Garcia Rosa, Jorge Bidarra e Bruno Dallari.

Devo ainda ao Departamento de Letras Vernáculas e Clássicas, do Centro de Ciências Humanas, da Universidade Estadual de Londrina, especialmente a Ivone Alves de Lima e a Esther Gomes de Oliveira (respectivamente, chefe do departamento e diretora do centro), pela concessão da licença total de 3 anos que me permitiu concluir os créditos do doutoramento e usufruir da bolsa-sanduíche durante um ano.

À Capes, eu devo dois agradecimentos: um pela bolsa de doutorado, concedida de março de 1996 a fevereiro de 1998, e o outro pela bolsa-sanduíche, concedida de março de 1998 a fevereiro de 1999. Sem esse auxílio financeiro, teria sido impossível trilhar o caminho percorrido até aqui.

O último agradecimento institucional é para o Linguistics Department, da New York Univeristy, que me recebeu durante o ano de 1998, proporcionando condições apropriadas para completar minha formação em Prolog, além de oferecer ainda algumas oportunidades, como a de assistir a uma palestra de Gennaro Chierchia, que eu não tive no Brasil.

Além dos agradecimentos institucionais, cabem também alguns agradecimentos acadêmicos. O primeiro e fundamental agradecimento é para Edson Françozo, o meu orientador (na verdade, depois de tanto tempo de convivência, muito mais do que orientador: um companheiro com quem compartilhei alguns bons momentos junto com muitos cafés). Ele acompanha pacientemente minha formação de pesquisador desde 1990, quando ingressei no mestrado, do qual ele também foi o orientador; isso sem contar o fato dele ter sido o meu primeiro professor de lingüística.

Em segundo lugar, José Borges Neto foi um precursor e um grande interlocutor. Em seu trabalho pioneiro de ensino de lingüística a alunos de computação na Universidade Federal de Curitiba, onde ensinava alguns fundamentos da semântica de Montague e incentivava os alunos a desenvolverem projetos em Prolog sobre o assunto, o prof. Borges estava estabelecendo as bases do trabalho ao qual eu também tenho me dedicado. Alguns programas desenvolvidos por alguns de seus alunos influenciaram profundamente a minha escolha por este mesmo tema. Acho que a melhor forma de agradecer a ele é poder estar colaborando com algum avanço nessa área.

Como o critério de ordenamento aqui não é tanto o de importância lingüística, mas o de paciência com temas computacionais, o terceiro a ser agradecido aqui é Rodolfo Ilari. Devo ao Ilari a primeira menção ao nome de Montague, quando eu o procurei, ainda no começo do meu mestrado, solicitando sua orientação. Minha intenção era estudar a expressão do tempo em português, um assunto no qual ele provavelmente é a maior autoridade no Brasil; foi nessa oportunidade que ele me recomendou estudar semântica de Montague. Infelizmente, por motivos nada acadêmicos (a menos que considerássemos a sociologia da academia), o Ilari não pôde ser meu orientador naquela oportunidade. De qualquer forma, devo a ele toda a minha formação em semântica formal.

Da área de ciência da computação, devo agradecer em especial a Ariadne Maria Brito Rizzoni Carvalho e a Jacques Wainer. À primeira, devo minha introdução à programação em Prolog; ao segundo, devo boas discussões e sugestões bibliográficas sobre processamento de linguagem natural (PLN); e a ambos, devo um curso de introdução ao PLN, do qual resultou um grupo de estudo que nós, alunos daquele curso, ainda mantivemos por mais seis meses além do curso. Muita coisa da parte

computacional, que não teria como aprender sozinho, eu aprendi nessas oportunidades, junto com a Ariadne, o Jacques e os alunos daquele curso.

A Ray C. Dougherty, preciso agradecer não só pela acolhida institucional, oferecendo as melhores condições para que eu pudesse desenvolver os meus estudos sobre programação em Prolog, mas também por todo o suporte emocional quando vivi um período difícil de minha vida pessoal lá em Nova Iorque. Espero ter sido um bom seguidor do seu exemplo e do seu conselho.

Finalmente, o último agradecimento acadêmico cabe a Carlos Franchi. Mais do que agradecimento, deixo aqui o reconhecimento não só por tudo o que o Franchi influenciou na minha formação de lingüista, mas pelo seu trabalho de formação de quase todos os lingüistas em atividade atualmente. Nesse momento, em que a dor de sua perda ainda é bastante recente, tenho a esperança de retribuir o que ele fez por todos nós, tentando fazer o que eu acho que ele gostaria de ver sendo feito: colaborar para a formação de novos lingüistas conscientes não só dos melhores recursos científicos para se praticar a lingüística, mas de todas as implicações sociais de nossos gestos.

Depois dos agradecimentos acadêmicos, restam apenas os agradecimentos pessoais.

A Paula Lens Costa Lima, além dos agradecimentos acadêmicos já feitos, pois quase tudo o que eu aprendi sobre psicolingüística experimental foi junto com ela, durante a nossa convivência no LAFAPE, devo também agradecimentos pessoais. Por mais que eu tenha aprendido junto com ela, o mais marcante mesmo foi a atenção e a compreensão quase maternal com que ela nós acolhe. Com isso, a Paula é uma das poucas pessoas que conseguem enxergar além das barreiras que a gente cria para nos proteger no intrincado mundo dos debates acadêmicos.

Agradeço ainda aos meus pais, Nilda Ventura Pagani e Luiz Pagani Sobrinho (também falecido durante a redação desta tese), por todo o suporte quando eu nem imaginava o que isso iria significar.

Ao meu irmão e à minha cunhada, Carlos Eduardo Pagani e Luciana de Souza Pagani (casados na etapa final de redação desta tese; afinal, um trabalho como esse não

poderia contar apenas com perdas), eu agradeço pela hospedagem incondicional toda vez que preciso estar em Campinas.

Ainda sobre hospedagem, não poderia deixar de agradecer ao casal Luís Gonçales Bueno de Camargo e Patrícia Cardoso da Silva, bem como a suas filhas Lívia e Carmen, porque eles sempre me abrigam em Curitiba. Na verdade, o agradecimento que eu devo é muito maior do que apenas o de hospedagem, e eles sabem bem disso.

O último agradecimento para hospedagem é para Aparecida de Lourdes Branco Paiva, que me acolhe como a um filho e me permite ocupar desordenadamente o quintal de sua casa para praticar um hobbie que, de outra forma, eu não teria como me dedicar a não ser precariamente.

O último agradecimento, e também o mais importante de todos, vai para Cristiane Branco Paiva, a quem eu conheci quando este trabalho estava pela sua metade, e que aceitou trocar o seu nome para Cristiane Branco Paiva Pagani e depois nos deu Ana Luísa Paiva Pagani, a quem também preciso agradecer por quase não ter chorado nas madrugadas em que eu ainda estava escrevendo e revisando esse texto. Sem as duas, a minha vida não teria metade do sentido que tem hoje.

Não poderia terminar esses agradecimentos sem pedir desculpas a todos os que eu tenha esquecido de agradecer aqui; afinal, são tantas as pessoas que acabam colaborando com o nosso trabalho, direta ou indiretamente, que é quase impossível fazer justiça a todas elas. De qualquer forma, mesmo que as tenha esquecido nesse momento, elas terão sempre minha gratidão incondicional; contava também que a conclusão desse trabalho pudesse ser demonstração de reconhecimento a todas as pessoas que me auxiliaram.

Finalmente, seria desnecessário dizer que eu sou o único responsável por todos os erros que cometi durante esse trabalho, e que ainda devem aparecer nesse texto, apesar da insistência das várias pessoas que tentaram me alertar para eles.

Sumário

1. Resumo

Nesta tese foram desenvolvidos analisadores gramaticais para as seguintes línguas apresentadas no livro **Introduction to Montague Semantics** (Dowty, Wall & Peters 1981): L_0, L_{0E}, L_1 e L_{type}. Nesse manual, a apresentação da semântica de Montague é feita primeiro através da apresentação de uma parte do cálculo de predicados, seguida de sua correspondente adaptação para um fragmento do inglês; e aos poucos vão sendo introduzidas complexidades que resultam nas novas línguas. A língua L_0 corresponde à parte do cálculo de predicados apenas com constantes (nomes de indivíduos e predicados); a língua L_1 corresponde à parte que contém as variáveis para indivíduos (chamada de primeira ordem); e em L_{type} a língua é acrescida de variáveis para qualquer categoria sintática (o que é conhecido como ordem superior).

Estamos empregando o termo "analisador gramatical" como tradução do termo em inglês "*parser*"; mas, ao contrário do que muitas vezes acontece, o analisador gramatical aqui, além de incluir um analisador sintático, inclui também um interpretador semântico. A única língua que não teve implementação completa foi L_{type}, para a qual não foi feito o seu interpretador semântico.

No entanto, essa falta fica justificada porque durante a tentativa de implementação de seu interpretador, constatou-se um defeito na especificação formal de suas regras semânticas. Assim, ao invés dessa implementação, optou-se por descrever e propor uma solução para esse problema.

Palavras-chave:

- Semântica de Montague
- Análise gramatical como dedução
- Prolog

2. Apresentação

A história dessa tese é um pouco mais antiga do que este doutorado; ela começou, na verdade, durante o meu mestrado, ainda quando eu precisava escolher um tema, e eu cogitei, durante muito pouco tempo, a possibilidade de reunir Prolog e semântica de Montague. Naquela época, apesar do meu interesse por questões de modelamento e simulação do conhecimento lingüístico, e principalmente no uso do computador para fazer isso, não demorou muito para perceber que este seria um objetivo muito pretensioso frente às restrições de prazo que começavam a vigorar.

Assim, mais por incapacidade do que por gosto, precisei abandonar a idéia de implementar a semântica de Montague em Prolog, por mais que isso inicialmente me parecesse promissor. Mas não era difícil prever que não seria nada fácil para um mestrando em lingüística dominar as complexidades técnicas tanto de uma, quanto de outra área: naquele momento, minha formação não me permitia dominar minimamente os recursos lógicos e computacionais que este projeto exigiria.

No entanto, passado o mestrado e tendo ingressado no doutorado, eu achei que agora poderia retomar este antigo projeto. E como no meu mestrado eu já tinha refletido sobre algumas relações entre a cognição e a linguagem, achei ingenuamente que não seria demais acrescentar às minhas pretensões a discussão dessas questões sobre modelamento da linguagem através desta implementação.

Dessa forma, o objetivo inicial para o presente trabalho seria o de desenvolver uma tese que pudesse ser classificada segundo os moldes de uma psicolingüística computacional. Segundo Crocker (1996: 6), essa nova área da Ciência Cognitiva se ocuparia em estudar "o vocabulário representacional e informacional da faculdade lingüística (definida pela teoria sintática atual), os algoritmos para se chegar a estas representações (lingüística computacional), a organização desses processos dentro da mente e o seu funcionamento (psicolingüística)". Por 'faculdade lingüística', o autor se refere à busca por uma gramática universal nos moldes do programa chomskyano. Isso fica claro quando ele menciona os três princípios chomskyanos para o estudo empírico do conhecimento da língua (Chomsky 1986, apud Crocker 1996: 7):

- O que constitui o conhecimento da língua?
- Como o conhecimento da língua é adquirido?
- Como o conhecimento da língua é posto em uso?

Ao contrário de Crocker, porém, minha pretensão era de explorar também o aspecto semântico da língua, e não apenas o sintático. Por isso, a faculdade lingüística não seria apenas "definida pela teoria sintática atual", mas também por uma teoria semântica pertinente. Além disso, eu sentia também que há na definição de psicolingüística dada por Crocker alguma imprecisão no uso do termo "representação" (e em seus derivados), típica da gramática gerativa: ao postular uma suposta realidade psicológica para a teoria gerativa, esconde-se a diferença entre representação mental e representação teórica, que é justamente o que precisa ser empiricamente comprovado pela psicolingüística.

Mas, como o projeto de mestrado, este também precisou ser revisto, ainda que não tão radicalmente. No decorrer do que eu achava que seria uma etapa de preparação para a construção do modelamento, acabei me deparando novamente com o que eu acreditei que fosse mais incapacidade minha (apesar de ter me preparado um pouco mais dessa vez) em relação aos recursos lógicos empregados na interpretação da língua L_{type}, mas que depois se converteu no que eu acredito ser a descoberta de um pequeno defeito no manual de Dowty, Wall & Peters 1981 (daqui por diante apenas DWP).

Antes de encarar o desafio de simular algum aspecto do comportamento semântico, eu achei que tentar simplesmente implementar analisadores gramaticais[1] para as línguas desse manual seria um bom exercício, do qual eu esperava inclusive que surgissem algumas questões importantes na controvérsia entre as partes declarativa e procedimental do Prolog. Para isso, eu contava com a crença de que, devido ao seu largo uso, esse manual não apresentasse nenhuma grande contradição.

[1] Estou traduzindo por "analisador gramatical" o que em inglês normalmente é chamado de *parser*.

Assim, iniciei o que eu achava que seria apenas o começo de minha tese, aproximando duas áreas que sempre estiveram formalmente próximas, mas que nunca foram explicitamente reunidas: a semântica de Montague e o Prolog.[2]

Tanto a semântica de Montague quanto o Prolog têm como base o cálculo de predicados. A primeira não só assume que as línguas naturais podem receber o mesmo tratamento formal dado às linguagens artificiais (como o do próprio cálculo de predicados), mas chega mesmo a empregar uma linguagem intensional como mediadora da interpretação das expressões das línguas naturais. A segunda se pretende uma linguagem de programação que reproduza computacionalmente uma parte do cálculo de predicados, permitindo provas mecânicas de teoremas.

Assim, esse então primeiro objetivo pode ser representado pela seguinte pergunta: o que é preciso para implementar em Prolog a semântica de Montague? Essa pergunta se deve principalmente à constatação de que o Prolog, apesar de toda a sua suposta declaratividade, ainda continua preso à procedimantalidade dos seus interpretadores e dos computadores nos quais ele é posto para funcionar.[3] A semântica de Montague, por sua vez, não se preocupa diretamente com questões sobre a heurística da prova, e por isso não se interessa pela sua computabilidade: tudo o que ela exige é que haja uma prova para os teoremas, já uma implementação computacional precisa se preocupar com a maneira dessa prova ser atingida.

---

[2] As duas únicas implementações computacionais conhecidas (Friedman & Warren 1978 e Warren & Friedman 1982) que mencionam explicitamente a semântica de Montague não mencionam o Prolog. Talvez a única exceção seja o artigo de um grupo de argentinos (Hack, Gonzalez, Catuogno, Moure & Campbell 1990) que, provavelmente por uma falha editorial, não fazia nenhum comentário mais explícito sobre o programa em si e, o mais grave, não apresentava nenhum endereço para contato, apesar dos autores oferecerem o seu programa.

[3] Supostamente, o Prolog seria uma linguagem de programação declarativa, na qual o programador deveria se ocupar exclusivamente com a descrição do problema a ser computado; uma vez que o problema estivesse adequadamente descrito, as soluções para o problema seriam automaticamente encontradas pelo mecanismo de inferência embutido no Prolog. Num lado diametralmente oposto estariam as linguagens de programação procedimental, nas quais o programador, além de descrever adequadamente o problema, precisa ainda construir uma máquina inferencial para que a solução possa ser encontrada. Como fica evidente, assim que se começa a programar em Prolog, é que não se pode confiar completamente nessa declaratividade: existem algumas restrições atribuídas principalmente a restrições técnicas, que obrigam o programador a se preocupar também com a procedimentalidade do Prolog.

Portanto, apesar da proximidade entre as duas áreas, também existe entre elas uma distância que precisava ser estreitada. Dessa forma, pode-se dizer que esta parte da tese seria um exercício de lingüística computacional, na medida em que ela vai buscar na ciência da computação, em geral, e na programação em Prolog, em particular, técnicas para permitir essa implementação computacional da semântica de Montague. Mas se deve entender aqui o termo computacional num sentido mais amplo do que ele encontra normalmente na lingüística computacional, onde as questões de eficiência estão entre as mais importantes (os algoritmos precisam não apenas funcionar, mas funcionar da melhor forma possível); mais importante do que eficiência, nesse início parecia mais importante algumas condições mínimas de computabilidade. Em alguns momentos algumas questões sobre a eficiência das implementações serão até comentadas, mas elas terão aqui um caráter apenas indicativo: apesar de todo o esforço, esta continua sendo a tese de um lingüista, e não de um cientista da computação.

Mas se esta, com certeza, não é a melhor das implementações possíveis, não há nenhuma desculpa para que ela não seja, por outro lado, a mais montagoveana delas. Na lingüística computacional, para que os programas funcionem, muitas vezes costuma-se sacrificar a fidedignidade teórica em favor da eficiência, de modo que algumas vezes fica difícil reconhecer neles a teoria que os inspirou; nesses casos, é comum se falar eufemisticamente em "equivalência fraca" entre o aplicativo desenvolvido e a teoria empregada. Para essa equivalência fraca, é suficiente que o aplicativo apresente os mesmos resultados que são previstos pela teoria, ainda que ele não execute exatamente os mesmos passos determinados pela teoria. Aqui nesta tese, ao contrário, busca-se uma equivalência forte: não basta chegar aos mesmos dados previstos pela teoria, é preciso que a própria teoria seja reconhecível no código do programa.

Despretensiosamente, poderíamos propor então uma nova nomenclatura: poderíamos chamar a lingüística computacional clássica de "lingüística Computacional", enquanto que aqui estaríamos praticando uma "Lingüística computacional". E a brincadeira vale nas duas direções: enquanto, na primeira, na qual "computação" está em maiúscula e "lingüística" em minúscula, as questões de computabilidade devem ser as mais relevantes; na segunda, ao contrário, as questões lingüísticas é que são as mais pertinentes. Mas é claro que essa ironia, na verdade, esconde (ou revela) a incapacidade

das primeiras gerações de pesquisadores de áreas interdisciplinares para manter o equilíbrio entre as muitas aptidões exigidas.

Como, do ponto de vista computacional, a programação em Prolog foi escolhida como guia, para dar rumo ao aspecto lingüístico, optou-se pela escolha do manual de DWP. Mais especificamente, a intenção seria a de ir implementando seqüencialmente cada uma das linguagens apresentadas nesse manual; apesar de relativamente antigo, ele ainda continua sendo o material didático mais citado para a iniciação ao assunto. Textos mais recentes, como o manual de Chierchia & McConnell-Ginet (1990, que no ano de 2000 recebeu uma segunda edição revista) e o de Cann (1993), apesar de didaticamente mais avançados, são tecnicamente mais condescendentes (algumas complicações técnicas são sacrificadas pela simplicidade expositiva); ainda montagoveano, mas tecnicamente ainda mais distante, é o manual de Chierchia (1997).[4] Além disso, em todos esses manuais, seus autores reconhecem explicitamente terem se espelhado no primeiro.

Contudo, durante a implementação do analisador gramatical para a língua L_{type} comecei a achar que minha formação não tinha sido suficiente para enfrentar o problema que eu estava encarando. (Apesar de todo o esforço para aprender um pouco de programação em Prolog, eu sei que eu continuo sendo um lingüista que sabe um pouco de Prolog, e estou longe de chegar a ser um programador nessa linguagem; é muito pouco provável inclusive que eu venha a ser um.) Por causa disso, acabei despendendo muito esforço tentando descobrir qual era a deficiência de programação que me impedia de implementar o interpretador semântico de L_{type}, antes de finalmente começar a desconfiar de algum improvável defeito no próprio manual de DWP. Mas essa foi a minha conclusão final.

Quando eu percebi, meus objetivos iniciais estavam "corrompidos". Novamente devido às restrições de prazo para o término do doutorado, seria impossível tentar construir ainda um modelo de alguma capacidade semântica, até porque uma segunda parte do projeto inicial seria, depois de terminados os exercícios de implementação das

[4] Esse último está sendo traduzido em conjunto por mim, Rodolfo Ilari e Lígia Negri.

línguas de DWP, adaptar o analisador gramatical para a língua portuguesa (a língua natural de DWP é o inglês, que foi mantida nessa etapa inicial de implementação).

De qualquer forma, acredito que os resultados obtidos são suficientes para se classificar o meu trabalho como tese de doutoramento: apesar de não ter discutido questões relativas à simulação de algum aspecto semântico de uma língua natural através da implementação computacional de uma teoria semântica (que era o objetivo inicial da tese), foi justamente algo que eu achava que seria uma parte mais "burocrática" dela (os exercícios de implementação dos analisadores para as línguas de DWP, que deveriam constituir apenas uma preparação para a construção do modelamento de algum aspecto da semântica do português) que acabou resultado na descoberta e na sugestão de solução de um equívoco cometido pelos autores do manual, um erro que parece não ter sido relatado anteriormente na literatura sobre a semântica de Montague.

Dessa maneira, o texto apresentado aqui é uma espécie de relatório desse percurso que começa com a apresentação do Prolog para os lingüistas, no capítulo 1. (Apenas para lembrar, o capítulo 2 é esta própria Apresentação, e o o capítulo 1 é o Resumo da tese.) Na seqüência, o capítulo 4, o maior da tese e sua parte principal, descreve cada uma dessas línguas de DWP até L_{type}, tanto em seu aspecto sintático quanto semântico, bem como a implementação em Prolog do seu respectivo analisador; esse capítulo termina com a minha sugestão para corrigir o deslize na determinação das regras semânticas de L_{type}. A tese se encerra no capítulo 5, no qual apresento a minha conclusão em relação ao trabalho relatado aqui, e minhas expectativas sobre possíveis conseqüências do que foi descoberto e começado aqui.

Depois do texto propriamente dito, além das Referências bibliográficas utilizadas nesta tese (capítulo 7), foram incluídos também um capítulo com os Apêndices (capítulo 8, no qual o leitor pode encontrar alguns programas que acabaram não sendo diretamente utilizados na apresentação do trabalho) e um resumo em inglês (capítulo 6, imediatamente antes das Referências bibliográficas).

Antes de passar definitivamente aos assuntos técnicos, uma última observação: exatamente como Chierchia e McConnell-Ginet (2000: xi) admitem em relação à

semântica formal, eu também me dedico à "Lingüística computacional" com a mesma esperança de um dia ficar rico e famoso!

3. Prolog

Uma das principais motivações que levaram à criação do Prolog, e de boa parte do que se conhece por programação lógica, segundo dois de seus precursores, estava ligada à necessidade de se processar computacionalmente uma língua natural:

> Quase que desde a sua origem, o desenvolvimento da programação lógica tem estado intimamente relacionado à busca por um formalismo computacional para expressar análises sintáticas e semânticas das sentenças das línguas naturais. Um dos principais propósitos do desenvolvimento do Prolog era o de criar uma linguagem na qual as regras de estrutura sintagmática e de interpretação semântica para um sistema de pergunta-e-resposta em língua natural pudessem ser facilmente expressas.
>
> (Pereira & Shieber 1987: 2)

Apesar de sua aplicação em outras áreas da Inteligência Artificial (como na prova automática de teorema), o Prolog sempre esteve fundamentalmente ligado ao processamento de línguas naturais (PLN).[5] A partir da suposição de que as questões relativas à análise sintática e à interpretação semântica pudessem ser expressas através de uma língua como a da lógica de primeira ordem, acreditou-se que as estruturas sintáticas e semânticas das línguas naturais poderiam ser descritas como teoremas a serem provados.

3.1. Programação Lógica

Como foi dito acima, o Prolog é uma linguagem de programação lógica. Por **programação lógica**, entende-se que um programa de computador pode ser concebido como um problema que, por sua vez, pode ser expresso através da lógica simbólica e resolvido através de um procedimento de inferência. Desse ponto de vista, programar é

---

[5] Normalemente, a abreviação "PLN" está associada ao termo "processamento de linguagem natural"; como na Lingüística o termo "linguagem" geralmente remete à capacidade lingüística, que é apenas um dos aspectos das línguas, vou preferir traduzir "natural language processing" por "processamento de língua natural".

resolver um problema através da inferência de uma conclusão adequada a partir de um conjunto de premissas pertinentes:

> Um dos principais objetivos da lógica simbólica tem sido o de capturar a noção de conseqüência lógica através de recursos formais e mecânicos. Se as condições para uma determinada categoria de problemas puderem ser formalizadas por uma lógica adequada, então uma solução poderia ser encontrada na construção de uma prova formal para a asserção do problema, a partir das premissas.
>
> (Pereira & Shieber 1987: 1-2)[6]

Conseqüentemente, um programa em Prolog é basicamente uma base de dados composta por asserções feitas num formato muito semelhante à notação usada no cálculo de predicados. Essa base de dados, que estabelece todas as relações válidas entre os objetos envolvidos no problema, é o conjunto de premissas que permitirá chegar à conclusão pretendida:

> Um procedimento construtivo de prova que não apenas elabore as provas, mas que também atribua valores para as incógnitas da asserção do problema, pode então ser concebido como um dispositivo computacional para a determinação dessas incógnitas. Dessa perspectiva, as premissas podem ser concebidas como um programa, a asserção do problema como uma invocação desse programa (com determinados valores de entrada e com as incógnitas de resultado), e a prova como uma computação do programa.
>
> (Pereira & Shieber 1987: 2)

Essa maneira de programar através de definições desse tipo é chamada de **declarativa**, e se opõe à maneira **procedimental**, onde o programador precisava não só descrever o problema, como também estabelecer um procedimento de resolução para ele. Assim,

---

[6] Parece que os autores confundem um pouco a questão ao falarem em "*formal, mechanical, means*", como se 'mecânico' e 'formal' fossem a mesma coisa. O primeiro termo parece ter sido herdado da lógica e não tem nenhuma relação com se chegar a uma solução, apenas com a garantia de que, se existir uma, deve haver uma prova dela; já o segundo parece ter vindo da computação, e sugere a obtenção efetiva da solução. Um exemplo dessa distinção aparece na limitação que as definições recursivas à esquerda, apesar de logicamente perfeitas, impõem a alguns algoritmos para a computação da prova.

> o Prolog pode ser considerado uma linguagem bastante *declarativa*; ou seja, um programa em Prolog pode ser considerado como uma asserção do *que* está sendo computado, independentemente de qualquer método específico para a computação. O Pascal, por outro lado, é *procedimental*, à medida que aquilo que um programa em Pascal computa pode ser definido apenas em termos de como ele executa a computação. Obviamente, o Prolog também apresenta uma interpretação procedimental; ele recorre a um método específico para computar as relações que vão permitir considerar que um programa esteja asserindo declarativamente. Além disso, determinadas partes "impuras" do Prolog não apresentam nenhuma interpretação declarativa. Mas o Prolog, enquanto primeiro passo na direção de uma linguagem de programação lógica, pode ser amplamente considerado como uma linguagem declarativa.
>
> (Pereira & Shieber 1987: 3-4)

Uma das principais comodidades da programação declarativa é a de que o programador quase não precisa se preocupar com o procedimento de inferência, porque esse tipo de tarefa já está contido nas especificações iniciais das linguagens de programação lógica.[7]

3.2. Mecanismo de prova do Prolog

No Prolog, esse mecanismo inferencial funciona descendentemente (*top-down*) e em profundidade (*depth-first*). Isso significa que, para provar uma conclusão, o Prolog toma a base de premissas de cima para baixo e da esquerda para a direita. Com um conjunto de premissas como:[8]

---

[7] Essa afirmação poderia sugerir que o Prolog fosse o paraíso da programação de computador, mas isso não é bem assim. Devido ao funcionamento da sua máquina de inferência, o Prolog não lida bem com definições com recursividade à esquerda (cf. discussão na próxima seção), e por isso o programador ou precisa evitar definições desse tipo, ou então interferir no procedimento de inferência. Para essa segunda solução, apesar de não ser possível alterar o procedimento descendente e em profundidade do Prolog, existem algumas técnicas de programação que simulam outras estratégias de solução (sobre essas várias técnicas de busca, ver Le 1993: 272-307).

[8] O exemplo está escrito numa notação ainda muito mais própria da lógica simbólica do que da sintaxe mais específica do Prolog, que será apresentada mais detalhadamente adiante; esse é apenas um exemplo para ilustrar o processo de inferência do Prolog. Convém observar também que as variáveis aqui aparecem em itálico, e que o símbolo de implicação material (normalmente representado por "→"), aqui é escrito em

1. pai(Luiz, Arthur)
2. pai(Antônio, Luiz)
3. pai(Ventura, Nilda)
4. mãe(Nilda, Arthur)
5. mãe(Catharina, Luiz)
6. mãe(Áurea, Nilda)
7. ancestral(x, y) ← genitor(x, y)
8. ancestral(x, y) ← genitor(x, z) & ancestral(z, y)
9. genitor(x, y) ← pai(x, y)
10. genitor(x, y) ← mãe(x, y)

para saber quem é ancestral de quem, bastaria solicitar que fosse provada a expressão "ancestral(x, y)". Percorrendo o conjunto de premissas de cima para baixo, a primeira expressão que se adequa à expressão a ser provada é 7, que para ser satisfeita precisa apenas que se satisfaça "genitor(x, y)". Consultando mais uma vez descentemente o conjunto de premissas, agora é a premissa 9 que servirá como próxima meta, que por sua vez exige a satisfação de "pai(x, y)". Finalmente, nesse ponto, a premissa 1 pode ser atingida e a solução para o problema é "ancestral(Luiz, Arthur)".

Mas essa é apenas a primeira solução que o Prolog consegue apresentar, ele ainda tem condições de sugerir algumas outras: na seqüência, retrocedendo (*backtracking*) apenas na busca da premissa 1, ainda se chegaria a "ancestral(Antônio, Luiz)" e "ancestral(Ventura, Nilda)" como soluções para a premissa 9. Com isso, esgotam-se todas as possibilidades de retrocesso para esta premissa; mas nem todas foram esgotadas para a 7: depois de 9, a premissa 10 também oferece soluções para ela, que são seqüencialmente "ancestral(Nilda, Arthur)", "ancestral(Catharina, Luiz)" e "ancestral(Áurea, Nilda)".

No entanto, nem todas as soluções possíveis para o problema inicial foram atingidas; esgotaram-se apenas as soluções para a premissa 7. Retrocedendo-se mais uma vez, agora que todas as possibilidades de inferência para a premissa 7 foram percorridas, o Prolog ainda encontra a premissa 8, que apresentará seqüencialmente as seguintes

---

sentido inverso ("←") para se adequar ao sentido de busca do Prolog. Com isso, uma premissa como 7, poderia ser lida como 'x é ancestral de y se x for genitor de y'; na lógica simbólica, normalmente, a expressão seria "$\forall x,y$[genitor(x, y) → ancestral(x, y)]", que poderia ser lida como ' para todo x e todo y, se x é genitor de y, então x é ancestral de y'.

soluções: "ancestral(Antônio, Arthur)", "ancestral(Ventura, Arthur)", "ancestral(Catharina, Arthur)" e "ancestral(Áurea, Arthur)".[9]

Ainda antes de passar à apresentação da sintaxe do Prolog, cabe um comentário sobre a observação de uma limitação do seu mecanismo de inferência (já mencionada na nota 7): caso a premissa 8 fosse reescrita como "ancestral(x, y) ← ancestral(z, y), genitor(x, z)" e fosse ainda trocada de lugar com a premissa 7, apesar dessas mudanças serem imateriais na lógica simbólica, o Prolog não conseguiria chegar a uma solução, pois ele entraria numa recursão infinita tentando satisfazer sempre a mesma premissa, sem jamais conseguir enxergar o fato que garantiria o seu término; essa recursão infinita é causada pela recursividade à esquerda da regra associada ao procedimento de busca descendente e da esquerda para a direita do Prolog.[10]

3.3. Sintaxe do Prolog

3.3.1. Fatos

Um **fato** ou **asserção** simples é constituído por um predicado, terminando necessariamente em um ponto.

O **predicado**, por sua vez, pode ser construído usando-se qualquer combinação de letras minúsculas como seu funtor, e vem seguido por uma série (possivelmente vazia) de argumentos (escritos dentro de parêntesis logo após o funtor e separados por vírgula);[11] geralmente por motivos mnemônicos, costuma-se dar aos funtores e aos argumentos

---

[9] Nesse ponto o leitor deve ser capaz de, se desejar, fazer para essas soluções o mesmo percurso de prova que foi apresentado para as soluções anteriores. Por isso, me desincumbo de apresentá-las aqui, de forma que elas possam servir de exercício para esse leitor mais interessado.

[10] Sobre esse assunto, ver Pereira & Shieber 1987: 22-26, de onde o exemplo acima foi adaptado. Também aqui não assumirei a tarefa de apresentar passo a passo o problema; fica novamente o leitor convidado a assumir mais esse exercício, caso ele se interesse.

[11] Para uma definição mais precisa da ortografia dos termos (ou átomos) que constituem os funtores, ver Clocksin & Mellish 1981: 24-5. Além das letras minúsculas, um dos elementos que podem ser empregados é o *underscore* ("_"), que geralmente simula o que seria o espaço na ortografia tradicional, que não é permitido como parte do funtor. Na verdade, a exigência para os funtores é a de que pelo menos a sua primeira letra seja minúscula, mas é pouco comum encontrar funtores grafados com alguma letra maiúscula.

nomes que recordem a relação que se quer definir. Cada um dos argumentos, quando forem constantes, também respeitarão a mesma ortografia dos funtores.

Assim, os seguintes exemplos são possíveis fatos para o Prolog:

- "a.": predicado sem argumentos, composto pelo funtor 'a';
- "azul(ceu).": predicado composto pelo funtor 'azul', cujo único argumento é 'ceu',[12]
- "menor_que(formiga,elefante).": predicado composto pelo funtor 'menor_que', seguido de dois argumentos: 'formiga' e 'elefante'.[13]

Em Prolog, a distinção das noções de funtor e predicado corresponde ao fato de que o **funtor** é o termo que se encontra diante dos parênteses, enquanto que o **predicado** é a combinação do funtor com a sua **aridade** (quantidade de argumentos).[14] Assim, apesar de apresentarem o mesmo funtor, se o número de argumentos não for o mesmo, tem-se dois predicados diferentes. Como em "pai(antonio)" e "pai(antonio,luiz)", por exemplo, ainda que ambas as expressões compartilhem o mesmo funtor ('pai'), elas constituem dois predicados[15] diferentes ('pai/1' e 'pai/2'), que poderíamos, arbitrária mas mnemonicamente, associar a 'Antônio é pai' e 'Antônio é pai de Luiz', respectivamente.

3.3.2. Regras

O tipo de asserção que foi definido acima é simples porque é composto por um único predicado. No entanto, esta não é a única forma com que podemos exprimir as asserções. Um outro tipo de asserção em Prolog, chamado de **regra**, é constituído por um

---

[12] Apesar de alguns interpretadores/compiladores de Prolog reconhecerem as letras acentuadas (como o Sicstus, o SWI e o Arity), muitos não reconhecem; por isso, o uso da ortografia regular do português muitas vezes não será respeitado.

[13] Observe que não é necessário incluir nenhum espaço entre a vírgula e o argumento que a segue; no entanto, isso pode ser feito sem prejuízo: "menor_que(formiga, elefante).".

[14] É comum expressar o predicado através do seu funtor e da sua aridade separados por uma barra inclinada ("/"), por exemplo: pai/1, pai/2 etc. Algumas vezes, no entanto, como em Matthews 1998: 34, a distinção não é feita tão claramente.

[15] Ainda não podemos chamar esses nossos exemplos de fatos porque eles não terminam com um ponto.

predicado inicial (chamado de **cabeça** (*head*) da regra), seguido por um conjunto de predicados (chamados de **corpo** ou **cauda** (*tail*) da regra), que também são separados por vírgulas, como os argumentos. A cabeça e o corpo de uma regra são separados pelo sinal ":-", que corresponde à implicação do cálculo de predicados invertida; e, como toda asserção em Prolog, ela também termina em um ponto. Um exemplo de regra seria "filho(luiz,antonio) :- pai(antonio,luiz).".

Segundo Matthews (1998: 54), "através de regras, definem-se novas relações em termos de outras que já foram previamente definidas." Assim, é através das regras que ocorre a programação propriamente dita em Prolog, pois é a sua unificação que irá permitir a generalização e a recursividade das definições.

3.3.3. Variáveis

Da forma como foram apresentadas até aqui, porém, as regras não têm muito valor porque elas ainda contêm apenas constantes; para que uma regra cumpra a sua função de generalização e possa definir economicamente outras relações, é necessário que ela contenha pelo menos uma variável. Uma **variável** em Prolog tem basicamente a mesma função que as variáveis do cálculo de predicados; ou seja, são posições vazias que devem ser preenchidas por constantes que satisfaçam o predicado que as contém (**unificação**).[16] Em Prolog, elas são indicadas pelo emprego de pelo menos uma letra maiúscula no início da expressão;[17] assim, "X" (mas não "x"), "Filho" e "PaiDe" são exemplos de variáveis.

Empregando variáveis, podemos generalizar a regra sobre filhos acima, de forma a tornar a nossa base de dados menos redundante; basta reescrevê-la como "filho(Filho, Pai) :- pai(Pai, Filho).", que é a forma com que se traduz em Prolog expressões como 'se

---

[16] Essa jeito de falar da função das variáveis é chamado de substituicional; além desse, um outro estilo mais formalmente adequado é o atribuicional. Vamos falar mais sobre isso quando falarmos sobre a Semântica de L_{type}, a partir da página 134.

[17] Além de iniciar com letra maiúscula, as variáveis também podem começar com um *underscore*; assim, "_filho" também seria uma variável no Prolog, e "_" corresponde à variável anônima (usada quando ela não precisa estar ligada a outra posição também vazia). Pode-se também empregar outras maiúsculas para

um indivíduo é pai de outro, este é filho daquele', em linguagem corrente, ou "$\forall x,y$[pai(x,y) → filho(y,x)]", na notação tradicional do cálculo de predicados.[18]

3.3.4. Listas

Além de constantes e variáveis, as posições argumentais podem conter ainda uma lista. Em Prolog, uma **lista** é uma seqüência ordenada[19] de termos (podendo ser inclusive uma seqüência sem nenhum termo; ou seja, uma lista vazia), separados por uma vírgula, reunidos de forma a contarem como se fossem um único termo.

Como "as listas são seqüências de itens que normalmente – ainda que não necessariamente – apresentam alguma relação entre si" (Matthews 1998: 65),[20] elas podem ser usadas para designar um determinado tipo de agrupamento. Assim, se quiséssemos utilizar o predicado "pai/2", por exemplo, para indicar os filhos de uma pessoa, sem alterar o predicado, ainda poderíamos registrar os filhos como segundo argumento, mas em forma de lista: "pai(antonio, [maria, pedro])." – que poderia corresponder a 'Antônio é pai de Maria e de Pedro'; um outro exemplo ainda, seria o de declarar não apenas o pai, mas também a mãe; nesse caso, bastaria transformar o primeiro argumento numa lista de dois elementos: "pai([antonio, ana], [maria, pedro])." – correspondendo, por exemplo, a 'Antônio e Ana são pais de Maria e de Pedro'. E como uma lista é um conjunto ordenado, poderíamos atribuir alguma propriedade pertinente à seqüência em que os elementos aparecem (como a de ordem de nascimento, na segunda posição argumental; e de par ordenado de marido e esposa, na primeira).

---

facilitar a leitura da variável, como em "divide(Dividendo,Divisor,Resultado,RestoDaDivisao)". Para mais detalhes, ver Clocksin & Mellish 1981: 25.

[18] Como o Prolog não reproduz exatamente o cálculo de predicados de primeira ordem, mas apenas um subconjunto dele (conhecido como cláusulas de Horn), não existem operadores universais ou existenciais. Sobre os processos de simplificação e extração desses operadores em Prolog, ver Clocksin & Mellish 1981: 269-72.

[19] Assim, as listas se distinguem dos conjuntos, já que estes são agrupamentos não-ordenados. Na verdade, pode-se considerar uma lista como um tipo de conjunto: um conjunto ordenado. Além disso, ao contrário dos conjuntos, as listas podem conter elementos repetidos.

[20] Não parece haver motivo para essa restrição, já que o simples fato de estarem na mesma lista é bastante para garantir que os itens apresentem alguma relação entre si: a de pertencerem à mesma lista.

Apesar da lista ter sido apresentada como uma seqüência de termos separados por vírgula ("[a, b, c, d]", por exemplo), sua estrutura em Prolog, na verdade, é a de um elemento (**cabeça** da lista) seguido de outra lista (**cauda** ou **resto** da lista, ainda que esta seja a lista vazia), separados um do outro por uma barra vertical ("|"). Assim, a lista anterior seria considerada por um interpretador Prolog como sendo "[a | [b | [c | [d | []]]]]"; ou seja, por uma lista encabeçada por "a" e seguida por outra lista, que por sua vez é encabeçada por "b" e seguida por uma outra lista, que começa por "c" e é seguida por uma quarta lista, composta por "d" e seguida, finalmente, por uma lista vazia. No entanto, quase todos os interpretadores de Prolog reconhecem o formato apresentado inicialmente.

Uma lista em Prolog, na verdade, tem a estrutura de uma pilha:

> Por analogia, imagine uma lista como uma pilha de pratos. O prato no topo é a cabeça da pilha. Se esse prato for retirado, o restante – a cauda – ainda será uma pilha de pratos (supondo que ainda tenha restado mais que um). Observe que o único prato que está diretamente acessível é o que está no topo da pilha – a cabeça –, isso vale igualmente para uma lista. Para atingir pratos abaixo na pilha, os pratos superiores precisam ser removidos antes [...] Finalmente, o único lugar em que um novo prato pode ser incluído na pilha é no topo, como uma nova cabeça; isso também vale para uma lista.
>
> (Matthews 1998: 65)

No entanto, essa analogia não funciona sempre: numa pilha de pratos, se quisermos chegar a um prato que não esteja na segunda posição, pode-se retirar de uma única vez todos os pratos do topo da pilha até ele; já numa lista isso não acontece, a menos que a sua cabeça seja explicitamente definida como uma outra lista. Além disso, ainda que seja tecnicamente correto falar em pilha de um único prato, dificilmente empregaríamos esse termo no uso cotidiano para isso; mas uma lista é uma pilha no seu sentido técnico.

Assim, numa lista como "[a, b, c, d]", a única forma de se chegar à quarta posição ("d") é pela remoção individual de cada um dos três itens anteriores: primeiro o "a", restando "[b, c, d]"; depois o "b", sobrando "[c, d]"; e finalmente o "c", chegando-se ao

"d".[21] Esse item só seria atingido em uma única operação caso a lista fosse "[[a, b, c], d]". Ou seja, uma lista é uma pilha da qual só podemos agir sobre o seu primeiro elemento.[22]

3.3.5. Definições

Outro conceito importante para a programação em Prolog, ainda que geralmente seja difícil encontrá-lo explicitamente apresentado, é o de 'definição'. Uma **definição** é um conjunto de asserções relativas a um mesmo predicado.[23] Dessa forma, em Prolog, qualquer encadeamento de fatos encabeçados pelo mesmo predicado constitui uma definição, que normalmente é chamada de "definição declarativa", pois ela é feita através da declaração exaustiva de asserções simples (no exemplo abaixo, pela asserção de todas as relações de paternidade pertinentes). Um exemplo de definição declarativa seria a tradução em Prolog das três primeiras premissas do nosso exemplo inicial:

```
pai(luiz, arthur).
pai(antonio, luiz).
pai(ventura, nilda).
```

3.3.6. Procedimentos

Uma simples definição declarativa já pode ser considerada um programa em Prolog; contudo, um programa desse tipo não é muito produtivo. Por esse motivo, o emprego da noção de definição geralmente está associado às definições compostas também por alguma regra; a esse tipo de definição dá-se o nome de procedimento. O que caracteriza um **procedimento**, então, é que ele contenha um conjunto de regras

---

[21] Para ser mais preciso, depois da terceira extração não temos "d", e sim "[d]". Para se chegar ao "d" precisamos de uma quarta extração, tendo como resto uma lista vazia ("[]"). Muitas vezes essa sutileza faz muita diferença num programa em Prolog.

[22] Mas isso não significa que uma lista não possa ser usada como outro dispositivo de armazenamento, como uma fila. Da mesma forma como é possível empregar o mecanismo descendente e em profundidade para simular outras estratégias de busca, ainda é possível simular uma fila através de uma lista; essa simulação aparece na técnica das listas diferenciais, cf. Le (1993: 219-23).

[23] Por "asserções relativas a um mesmo predicado", entende-se aqui 'asserções que apresentam o mesmo funtor inicial, sempre com a mesma aridade'.

recursivas (possivelmente uma única regra recursiva), combinado com o conjunto de condições (que também contenha pelo menos um fato) para o termino dessa recursividade. As duas regras abaixo, que também constituem um programa em Prolog, exemplificam a parte recursiva de um procedimento:[24]

```
ancestral(X, Y) :- pai(X, Y).
ancestral(X, Y) :- pai(X, Z), ancestral(Z, Y).
```

Junto com a definição declarativa para o predicado "pai/2" (que constitui o conjunto de condições de término para esse procedimento), essas duas regras relativas ao predicado "ancestral/2" compõem um procedimento que define a relação de ancestralidade.

3.4. Alguns predicados previamente definidos (built-in) e operadores

3.4.1. =..

O predicado "=.." (que é pronunciado como 'univ') serve para converter uma lista em uma estrutura de predicado e argumentos. Segundo Clocksin & Mellish (1981: 123), "'X =.. L' significa: 'L' é a lista composta pelo funtor de 'X' seguido pelos argumentos de X". Esse predicado pode ser usado em dois sentidos: 1) caso a variável 'X' seja instanciada com uma estrutura do tipo "funtor(arg1, arg2, ..., argn)", o interpretador Prolog constrói a lista "[funtor, arg1, arg2, ..., argn]" e tenta casá-la com a variável 'L', e 2) se a variável 'X' não estiver instanciada, a lista em 'L' é usada para construir a estrutura que será unificada com 'X' (nesse último caso, a cabeça da lista 'L' precisa ser um átomo, já que ele vai ser o funtor de 'X'). Para que esse predicado funcione, no entanto, é preciso que alguma das variáveis tenha sido instanciada: se ambas não estiverem instanciadas, o predicado falha; mas nada impede que ambas sejam

[24] Ela ainda é, fundamentalmente, uma tradução das premissas 7 e 8; a única modificação, para simplificar o exemplo, foi a substituição de 'genitor' por 'pai'.

instanciadas ao mesmo tempo (assim, o resultado será apenas o de comparação, e não o de construção, que foi sugerido acima).

Alguns exemplos apresentados por Clocksin & Mellish (1981: 123) são:

- "foo(a, b, c) =.. X.", resultando em 'X = [foo, a, b, c]'
- "X =.. [a, b, c, d].", resultando em 'X = a(b, c, d)'
- "a(b, c, d) =.. [a, b, c, d].", resultando em anuência

3.4.2. Name

Segundo Closksin & Mellish (1981: 124), "o predicado 'name' relaciona um átomo com a lista de caracteres (em ASCII) que o constituem". Assim, esse predicado pode ser usado para decompor um átomo numa lista de caracteres ou, alternativamente, para montar um átomo a partir da sua lista de caracteres em ASCII. Mais formalmente, "name(A, L)" estabelece que os caracteres do átomo 'A' são membros da lista 'L'; se o argumento 'A' estiver instanciado, o interpretador Prolog devolve em 'L' a lista de seus caracteres em ASCII; por outro lado, quando a lista 'L' é que estiver instanciada, o interpretador reúne em 'A' o átomo correspondente aos caracteres em ASCII de 'L'.

Como para "=..", aqui também pelo menos uma das variáveis precisa estar instanciada; se ambas não estiverem instanciadas, o predicado falha. E, mais uma vez, quando ambas estiverem instanciadas, o resultado é o de comparação.

Exemplificando:

- "name(name, X).", resultando em 'X = [110, 97, 109, 101]'
- "name(X, [110, 97, 109, 101]).", resultando em 'X = name'
- "name(name, [110, 97, 109, 101]).", resultando em anuência

3.4.3. Not

O predicado "not/1" (cujo funtor algumas vezes também pode ser "\+", lido como 'não é possível provar que') é uma "negação por falha" (*negation-as-failure*). Segundo

Covington, Nute & Vellino (1997: 20), "em Prolog, não se pode asseverar um fato negativo ('A Cíntia não é pai do Miguel'); só o que se pode fazer é concluir uma asserção negativa se não for possível concluir a asserção afirmativa correspondente. Mais precisamente, o computador não consegue saber que a Cíntia não é pai do Miguel; só o que ele consegue saber é que não há como provar que ela *seja* seu pai."

Assim, o significado operacional de "not(G)" é: se 'G' ocorre, então "not(G)" falha; se 'G' não ocorre, então "not(G)" não falha. Mas como Le (1993: 122) aponta: "*not(G)* não é capaz de apresentar nenhum resultado além de uma resposta confirmativa do tipo 'sim' ou 'não'. Portanto, *not(G)* só pode ser usado como teste, e nunca para encontrar uma solução." Isso ocorre porque a definição de "not/1" é fundamentada em outro predicado pré-definido "fail", que desfaz a unificação das variáveis ao ser satisfeito.[25] Assim, para ser bem empregado, o predicado "not/2" normalmente exige que todas as suas variáveis estejam instanciadas no momento de sua operação.[26]

3.4.4. ;

Além, do funtor "," que compõe cláusulas conjuntivas, o Prolog ainda dispõe do funtor ";" para construir cláusulas disjuntivas. Assim, "X; Y" é uma das maneiras de se exprimir em Prolog a disjunção. Se 'X' for satisfeito, a cláusula disjuntiva também o será; caso contrário, se o 'Y' ainda puder ser satisfeito, a disjunção ainda seria satisfeita. Esta só falharia se nem 'X' nem 'Y' pudessem ambos ser satisfeitos.

Recorrendo ao funtor ";", a definição de genitor, apresentada em 9 e 10, na página 26, em Prolog ficaria:

```
genitor(X, Y) :- pai(X, Y); mãe(X, Y).
```

---

[25] Como ele não está sendo diretamente empregado, não comentaremos o predicado "fail" aqui. Informações sobre ele podem ser encontradas em Clocksin & Mellish 1981: 85, em Covington, Nute & Vellino 1997: 34-6 e em Le 1993: 119-21.

[26] Mais informações sobre os cuidados no emprego da negação por falha podem ser encontradas em Covington, Nute & Vellino 1997: 20-2 e em Le 1993: 122-6.

Apesar de parecer mais natural do que a definição em duas linhas de 9 e 10, quase todos os autores de introduções ao Prolog recomendam que não se use o funtor ";" e, se usar, empregar parêntesis extras para evitar confusões com o operador ",". Como essa definição poderia ser expressa em Prolog como:

```
genitor(X, Y) :- pai(X, Y).
genitor(X, Y) :- mae(X, Y).
```

também em duas linhas como em 9 e 10, com o mesmo efeito, normalmente os programadores de Prolog dão preferência para esta última solução.

Um uso muito mais freqüente para ";" ocorre na operação do interpretador de Prolog. Quando solicitamos para o interpretador encontrar uma solução para um programa, quando ele a encontra e ainda restam outras soluções (geralmente indicado pelo sinal '->'), essas outras soluções podem ser obtidas, uma a uma, com o ";".[27] Assim, pela definição de "pai/2" acima, caso solicitássemos a um interpretador Prolog as soluções para ela, obteríamos como primeira resposta:

?- pai(X, Y).

X = luiz

Y = arthur ->

O sinal "->" no final da resposta significa que o interpretador ainda é capaz de tentar encontrar outras soluções. Dessa maneira, para solicitar a próxima delas, é preciso digitar o ";":

?- pai(X, Y).

X = luiz

Y = arthur -> ;

X = antonio

---

[27] No Sicstus, o sinal é "?" logo após a solução. No SWI, porém, não aparece nenhum sinal; sabe-se que existem outras soluções porque o interpretador não apresenta nenhum sinal de término (nem "*yes*" nem "*no*"). Mas as alternativas são obtidas com o mesmo ";", porém não é necessário aqui o "*enter*", como nos dois anteriores.

Y = luiz ->;

X = ventura

Y = nilda

no

Como ainda é possível notar, depois da segunda solução o Prolog ainda aponta outras possibilidades. Solicitado mais uma vez a encontrá-la o interpretador agora apresenta um último par de unificações para as duas variáveis, mas termina o procedimento indicando a anuência da solução. Isso significa que, naquele ponto, não existiam mais alternativas de solução.

3.4.5. Integer

O predicado "integer/1" serve para testar se um termo é um número inteiro. Assim, um interpretador Prolog responderia afirmativamente às seguintes solicitações: "?- integer(0).", "?- integer(1).", "?- integer(2).", e assim por diante. No entanto, esse predicado não tem capacidade gerativa; ou seja, caso solicitássemos a um interpretador "?- integer(X).", sua resposta seria negativa.

3.4.6. Is

Como o predicado "integer/1", o predicado "is/2" também serve para avaliar expressões aritméticas. Segundo Clocksin & Mellish (1981: 134), para "X is Y", "*Y* precisa ser instanciado com uma estrutura que possa ser interpretada como uma expressão aritmética [...] Primeiro, a estrutura instanciada por *Y* precisa ser um inteiro, chamado de *resultado*. Esse resultado é unificado com *X*, e o *is* tem sucesso ou falha dependendo dessa unificação."

As expressões aritméticas mencionadas acima são expressões do tipo "X + Y", "X – Y", "X * Y" e "X / Y", que correspondem respectivamente à soma, à subtração, à

multiplicação e à divisão.[28] O emprego desse operador está ligado a operações aritméticas como "X is 2 + 3", de forma que 'X = 5'. Uma aplicação típica pode ser vista no programa abaixo, que determina a posição de um elemento em uma lista.

```
position(Element, [Element | _], 1).
position(Element, [_ | Rest], Position) :-
      position(Element, Rest, PositionRest), Position is
      PositionRest + 1.
```

Com esse programa, pode-se descobrir a posição ocupada pelo "c" na lista "[a, b, c, d, e, f, c]" através de "position(c, [a, b, c, d, e, f, c], Position).", que vai resultar primeiro em 'Position = 3' e depois em 'Position = 7'. Alternativamente, pode-se descobrir qual é o quinto elemento dessa mesma lista, através de "position(Element, [a, b, c , d, e, f, c], 5).", que resultaria em 'Element = e". Esse mesmo programa ainda poderia ser usado para criar (ou unificar) uma lista com um determinado elemento em uma determinada posição da lista; assim, "position(c, List, 3)." resultaria numa lista 'L = [X, Y, c | Z]'.[29]

3.4.7. Setof

O predicado "setof/3" serve para reunir dados em uma única lista. Segundo Le (1993: 203), a definição declarativa de "setof(X, G, L)" seria "*L* é a lista ordenada dos diferentes termos *X* obtidos da solução do objetivo *G*". Em outros termos, em 'L' são colocados todos os termos que satisfazem o predicado 'G', mas os termos só são registrados nessa lista uma única vez, e eles são registrados em ordem crescente numérica e alfabeticamente (com os números vindo também antes dos caracteres).

Ainda empregando a mesma definição acima de "pai/2", poderíamos obter o conjunto de todos os pais através da cláusula "setof(X, pai(X), Pais)" (ainda seria preciso também definir "pai(X) :- pai(X, Y)"). A solução seria 'Pais = [antonio, luiz, ventura]'.

---

[28] Há uma quinta operação aritmética, "X mod Y" cujo resultado é o resto da divisão de *X* por *Y*

[29] Para ser empregado dessa última maneira, precisaríamos tornar esse programa um pouco mais determinístico: caso pedíssemos outras respostas, depois dessa primeira, apresentada no texto, o programa entraria em regressão infinita.

3.4.8. *Cut*

Segundo Pereira & Shieber (1987: 138), o *cut* é um "recurso metalógico que muda o regime de controle de um programa em Prolog e que pode ser usado com diversas finalidades, inclusive para o aumento da eficiência, a eliminação de redundâncias e a codificação de condicionais". Isso equivale a dizer que o *cut* não é um recurso propriamente declarativo, apresentando um conteúdo muito mais procedimental; assim, o *cut* introduz características não-declarativas e não-lógicas num programa.

Na concepção de Clocksin & Mellish (1981: 75), "o 'cut' permite dizer ao Prolog que escolhas anteriores não precisam ser consideradas novamente quando ele retrocede pela cadeia de objetivos satisfeitos". Com isso, apesar de aumentar o caráter procedimental de um programa, em detrimento do seu caráter declarativo, pode-se garantir mais rapidez na execução do programa e ocupação de menos espaço de memória.

Na definição de Le (1993: 110), "para impedir retrocessos indesejados, o Prolog oferece o predicado de controle *cut*, denotado pelo símbolo '!', cujo significado operacional é apresentado a serguir. O 'cut' (!) é uma cláusula que sempre é satisfeita, mas quando se encontra um *cut* durante um retrocesso, então a cláusula que contém o *cut* falha imediatamente, o que também acontece com a cláusula que a invocou."

Um exemplo do emprego do '!' é a diminuição do espaço de busca. Como vimos na seção 3.4.6 acima, o predicado "position/3" quando usado para criar uma lista com um determinado elemento em uma determinada posição indicava outra possibilidade de solução além da primeira, depois da qual o programa entrava em uma recursão infinita. Esse problema pode ser resolvido, tornando o programa mais determinístico pela inserção de um '!' no final da segunda regra da definição do predicado:

```
position(Element, [Element | _], 1).
position(Element, [_ | Rest], Position) :-
      position(Element, Rest, PositionRest), Position is
      PositionRest + 1, !.
```

Definido como acima, o programa apresenta uma única solução (a mesma apresentada na seção 3.4.6). No entanto, nessa nova versão, o programa só é capaz de

encontrar a primeira posição de um determinado elemento; assim, caso haja mais de uma ocorrência daquele elemento na lista, o programa só consegue encontrar a primeira (ou seja, para "position(c, [a, b, c, d, e, f, c], Position)." o interpretador só vai responder 'Position = 3'). Mas o que é mais grave, agora esse programa é incapaz de encontrar o elemento que ocupa uma determinada posição na lista; dessa forma, por esta versão determinística, não há solução para "position(Element, [a, b, c, d, e, f, c], 5)."

Como vimos, então, o uso do '!' pode afetar a bidirecionalidade das definições em Prolog. Por isso, todos os autores são unânimes em recomendar cautela no emprego desse recurso.

3.4.9. Gramática de cláusula definida

Qualquer interpretador Prolog possui embutido em si um mecanismo para facilitar a representação de gramáticas independentes de contexto (*context-free grammar*); esse mecanismo é conhecido como gramática de cláusula definida (*definte clause grammar*, ou DCG).

Uma gramática independente de contexto consiste numa série de regras que especifica a forma que cada constituinte de sua língua pode assumir. As regras de uma gramática desse tipo têm a estrutura "X → Y", que pode ser lida como 'X pode assumir a forma de Y' ou, alternativamente, como 'X se reescreve como Y', e ainda como 'X se constitui de Y'.

Uma gramática simples desse tipo seria:

S → SN SV

SN → Det N

SV → V SN

A primeira regra dessa gramática nos diz que um S (sentença) pode assumir a forma de um SN (sintagma nominal) seguido de um SV (sintagma verbal). Pela segunda regra, ficamos sabendo que um SN se reescreve como um Det (determinante) seguido por um N (nome). Finalmente, através da terceira regra, sabemos que um SV se constitui por

um V (verbo) seguido de um SN. E com isso somos capazes de construir a estrutura de constituintes de uma sentença.

Para que essa estrutura de constituintes possa se constituir efetivamente numa sentença, ainda é preciso ocorrer o que se costuma chamar de inserção lexical; ou seja, a introdução de símbolos terminais nessa estrutura de símbolos não-terminais. Essa inserção também pode ser feita através das mesmas regras de reescrita:

Det → o, a

N → homem, banana

V → comeu

Inserindo-se, então, esses itens lexicais na estrutura de constituintes permitida pela gramática acima, obteríamos a sentença "O homem comeu a banana".

Em Prolog, essa mesma gramática pode ser diretamente escrita através da DCG como:

```
s --> sn, sv.
sn --> det, n.
sv --> v, sn.
det --> [o].
det --> [a].
n --> [homem].
n --> [banana].
v --> [comeu].
```

Com essa DCG,[30] poderíamos facilmente constatar que "[o, homem, comeu, a, banana]" é uma cadeia aceitável, através de "s([o, homem, comeu, a, banana], [ ])." Pode parecer misterioso que tenhamos usado um predicado de dois argumentos ("s/2") para inquirir se uma lista de palavras é reconhecida através de uma gramática, principalmente quando na própria gramática o predicado com funtor "s" nem sequer possuía argumento nenhum; mas é que os interpretadores Prolog não operam diretamente com DCGs. Antes de introduzir uma DCG em sua memória de trabalho, o interpretador reconhece as regras

[30] Para ficar ainda mais parecida com a gramática de reescrita apresentada no texto, poderíamos ter escrito as regras do determinante e do nome respectivamente como "det → [a]; [o]." e "n → [homem]; [banana].";
no entanto, preferi usar o formato mais comum, com cada entrada lexical listada separadamente.

com formato de DCG e as traduz para um formato que permita lidar com uma técnica conhecida como diferença de lista (*list difference*).

Assim, na verdade, uma DCG como a acima é tratada em Prolog como:[31]

```
s(P0, P) :- sn(P0, P1), sv(P1, P).
sn(P0, P) :- det(P0, P1), n(P1, P).
sv(P0, P) :- v(P0, P1), sn(P1, P).
det([o | P], P).
det([a | P], P).
n([homem | P], P).
n([banana | P], P).
v([comeu | P], P).
```

Empregando essa técnica de diferença de lista, podemos conceber o predicado "s(P0, P)" da seguinte maneira: a lista "P0" é uma sentença com resto "P". Assim, a instrução "s([o, homem, comeu, a, banana], [])." pode ser entendida como 'a lista "[o, homem, comeu, a, banana]" é uma sentença sem nenhum resto'. Unificada com a cabeça da primeira regra da DCG acima, essa instrução resulta em "s([o, homem, comeu, a, banana], []) :- sn([o, homem, comeu, a, banana], X), sv(X, [])." , criando dois caminhos a serem satisfeitos: "sn([o, homem, comeu, a, banana], X)." e "sv(X, [])." , para o mesmo valor de 'X' em ambas as expressões.[32]

A primeira dessas instruções casa com a cabeça da segunda regra, resultando em "sn([o, homem, comeu, a, banana], X) :- det([o, homem, comeu, a, banana], Y), n(Y, X).", que cria mais duas sub-instruções: "det([o, homem, comeu, a, banana], Y)." e "n(Y, X)." A primeira destas sub-instruções casa com a primeira regra de inserção lexical para os determinantes "det([o | Y], Y).", unificando "Y" com o resto da lista e resultando na unificação 'Y = [homem, comeu, a, banana]'. Com isso, a segunda sub-instrução passa a "n([homem, comeu, a, banana], X).", que unifica com a primeira regra de inserção lexical para os nomes ("n([homem | X], X)."), resultando em 'X = [comeu, a, banana]'.

---

[31] A compilação das regras de uma DCG segue os seguintes formatos: 1) as inserções lexicais "cat --> [expr]." são traduzidas como "cat([expr | X], X).", onde "cat" e "expr" representam metavariáveis para as categorias e para as expressões, respectivamente, e 2) as outras regras, de tipo "cat_0 --> cat_1, ..., cat_i" são traduzidas como "$cat_0(P_0, P_i)$:- $cat_1(P_0, P_1)$, ..., $cat_i(P_{i-1}, P_i)$."

[32] Estaremos empregando consecutivamente o recurso de renomeação das variáveis para evitar problemas de confusão entre as unificações entre elas. Assim, o "P" da regra original foi aqui substituído por "X".

Conseqüentemente, a segunda daquelas duas instruções iniciais passa a "sv([comeu, a, banana], [])." , que, por sua vez, unifica com a cabeça da terceira regra da gramática: "sv([comeu, a, banana], []) :- v([comeu, a, banana], Z), sn(Z, [])." , resultando em outras duas sub-instruções: "v([comeu, a, banana], Z)." e "sn(Z, [])." A primeira delas casa com a regra de inserção lexical para o verbo ("v([comeu | Z], Z)."), resultando em 'Z = [a, banana]' e, por unificação, tornando a segunda sub-instrução em "sn([a, banana], [])." Esta, por sua vez, casa novamente com a cabeça da segunda regra: "sn([a, banana], []) :- det([a, banana], W), n(W, [])." , resultando em ainda outras duas sub-instruções: "det([a, banana], W)." e "n(W, [])." A primeira unifica com a regra de inserção lexical "det([a | W], W)." causando 'W = [banana] e transformando a segunda sub-instrução em "n([banana], [])." Ela, finalmente, casa com a regra de inserção lexical "n([banana | T], T)." unificando 'T = []'.

Nesse ponto, todas as sub-instruções geradas foram satisfatoriamente provadas, o que prova "s([o, homem, comeu, a, banana], [])."

A DCG tem sido largamente empregada no processamento de língua natural em Prolog. O exemplo mais difundido desse uso é o próprio livro de Pereira & Shieber (1987), no qual os autores apresentam minuciosamente a DCG e várias técnicas que ela permite desenvolver para se representar gramáticas para as línguas; nesse livro, o objetivo final dos autores é o de apresentar um sistema de interação homem-máquina construído através de uma DCG. Descrições mais simples da DCG podem ser encontradas em Clocksin e Mellish (1981: 210-32), Le (1993: 476-511) e Covington, Nute e Vellino (1997: 407-53); outras aplicações, incluindo uma pequena aplicação com estrutura de características (*feature structure*),[33] podem ser encontrada em Covington (1994).

[33] Algumas vezes, traduz-se *feature structure* por "estrutura de traços".

3.5. Biblioteca de alguns procedimentos comuns

3.5.1. Member

O predicado "member/2" é amplamente empregado, pois permite fazer testes entre uma lista e seus membros. Sua definição declarativa é bastante simples: um elemento é membro de uma lista se ele estiver na cabeça dessa lista; caso contrário, esse elemento é membro da lista se ele for membro do resto dessa lista. Em Prolog, isso é representado da seguinte maneira:

```
member(X, [X | _]).
member(X, [_ | Y]) :- member(X, Y).
```

Na sua aplicação mais evidente, esse predicado testa se um determinado elemento faz parte de uma determinada lista. Assim, "member(c, [a, b, c, d])." vai ser avaliado positivamente, pois "c" é o terceiro elemento da lista "[a, b, c, d]"; já "member(c, [a, b, c, d, e, f, c]" é avaliado positivamente duas vezes, pois "c" aparece duas vezes na lista: na terceira e na última posição.

Além disso, o predicado "member/2" pode ainda ser usado para apresentar, um a um, os membros de uma lista. Para "member(X, [a, b, c, d, e, f, c])", um interpretador Prolog apresentará seqüencialmente, e à medida que se solicita, cada um dos elementos da lista "[a, b, c, d, e, f, c]", da esquerda para a direita: primeiro "a", depois "b", em seguida "c", até finalmente o seu último elemento "c".

3.5.2. Append

A difundida definição da função de união de listas (*append*) é um exemplo clássico de procedimento:[34]

```
append([], L, L).
append([A | B], C, [A | D]) :- append(B, C, D).
```

[34] Em alguns interpretadores, essa definição faz parte do conjunto dos predicados previamente definidos.

Com este procedimento, a partir das listas "[a, b, c]" e "[d, e, f]", é possível obter a lista "[a, b, c, d, e, f]".[35]

Estruturalmente, o procedimento para 'append/3' apresenta uma diferença em relação ao procedimento para 'ancestral/2': a condição de término da recursividade do primeiro é um fato composto pelo próprio predicado cujo procedimento se está estabelecendo ("append([],L,L)."); no outro, o predicado da condição de término e o predicado relativo à regra do procedimento eram diferentes. Por isso, no procedimento para 'append/3', é preciso que a condição de término da recursividade venha antes da regra recursiva; isso já não é necessário no procedimento para 'ancestral/2'.

[35] Como este é um predicado muito conhecido, e é mencionado em qualquer manual de iniciação em Prolog, não explicarei o seu funcionamento; para maiores detalhes, ver Clocksin & Mellish 1981.

4. Analisadores para as línguas de DWP

4.1. Língua L_0

4.1.1. Sintaxe de L_0

A língua L_0 contém apenas oito expressões básicas distribuídas em três categorias sintáticas segundo a Tabela 1 (DWP: 14).

Categorias sintáticas	Expressões Básicas
Nomes	d, n, j, m
Predicados de um lugar	M, B
Predicados de dois lugares	K, L

Tabela 1 - Expressões básicas de L_0 e suas respectivas categorias

Através da listagem numa mesma célula da tabela, essa apresentação das expressões básicas de L_0 poderia sugerir que a definição das categorias em Prolog fosse feita por um predicado para cada uma delas, cujo único argumento seria a lista de suas expressões, segundo o Programa 1.

```
% L0_Cat1.ari
%     (Dowty, Wall & Peters 1981: 14)
% Basic categories for L0 defined as expressions list

name([d, n, j, m]).

one_place_predicate(['M','B']).

two_place_predicate(['K','L']).
```

Programa 1 - Categorias básicas definidas por lista de expressões

Com as categorias definidas em listas de expressões, no entanto, as regras de formação 1 e 2 precisariam executar algum teste de inclusão, como no Programa 2 (em que se emprega o predicado 'member/2'[36]), para determinar se uma certa expressão faz parte da lista daquela categoria.

---

[36] Sobre esse predicado, ver capítulo 3.5.1, na página 44

```
% L0_Syn1.ari
%     (Dowty, Wall & Peters 1981: 15)
% Rules 1 & 2 for L0 with list-defined categories

% Loading 'member' definition

:- reconsult('Member.ari').

% Loading basic expressions

:- reconsult('L0_Cat1.ari').

% Formation rules

/*1*/ sentence(F) :- F =.. [D,A], name(X),
       one_place_predicate(Y), member(D,Y), member(A,X).
/*2*/ sentence(F) :- F =.. [G,A,B], name(X),
       two_place_predicate(Y), member(G,Y), member(A,X),
       member(B,X).
```

Programa 2 - Regras de formação 1 e 2 para categorias definidas em listas

Dessa maneira, a formação das sentenças atômicas[37] exigiria primeiro o acesso à lista de expressões de cada categoria ('one_place_predicate/1', 'two_place_predicate/1' e 'name/1'[38]) e depois a seleção de um elemento dessas listas ('member/2'); esse procedimento ainda precisaria ser repetido para cada expressão da sentença, em cada uma dessas duas regras. Assim, ele precisa ser executado duas vezes na primeira regra, e três na segunda.

Mas se as categorias básicas forem individualmente definidas como predicado de cada expressão, conforme o Programa 3, a definição explícita desse tipo de teste torna-se desnecessária, pois ele é automaticamente executado pelo mecanismo de prova do próprio Prolog, já que os predicados que definem as categorias recuperam apenas uma expressão de cada vez, executando o mesmo teste em uma única operação.

```
% L0_Cat2.ari
%     (Dowty, Wall & Peters 1981: 14)
% Basic categories for L0 defined as expression predicate

name(d).
name(n).
name(j).
```

[37] Sentenças atômicas são sentenças que não são constituídas por outras sentenças, apenas por expressões básicas.

[38] Não confundir com o predicado pré-definido 'name/2', já explicado na seção 3.4.2.

```
name(m).

one_place_predicate('M').
one_place_predicate('B').

two_place_predicate('K').
two_place_predicate('L').
```

Programa 3 - Categorias básicas definidas como predicados de expressões

A implementação de todas as regras de formação feitas a partir dessa definição das categorias básicas será explicada a seguir, logo depois de sua apresentação.

Dentre as regras que regulamentam a formação das expressões complexas, apresentadas no Quadro 1 (DWP: 15),[39] pode-se distinguir fundamentalmente dois tipos: as regras que combinam apenas expressões básicas (como 1 e 2), formando sentenças atômicas; e as regras que compõem uma sentença a partir de outras sentenças (como 3 a 7).

1. Se δ é um predicado de um lugar e α é um nome, então $\delta(\alpha)$ é uma sentença.
2. Se γ é um predicado de dois lugares e α e β são nomes, então $\gamma(\alpha, \beta)$ é uma sentença.
3. Se ϕ é uma sentença, então $\neg\phi$ é uma sentença.
4. Se ϕ e ψ são sentenças, então $[\phi \wedge \psi]$ é uma sentença.
5. Se ϕ e ψ são sentenças, então $[\phi \vee \psi]$ é uma sentença.
6. Se ϕ e ψ são sentenças, então $[\phi \rightarrow \psi]$ é uma sentença.
7. Se ϕ e ψ são sentenças, então $[\phi \leftrightarrow \psi]$ é uma sentença.

Quadro 1 - Regras de formação para L_0

A aplicação recursiva dessas regras determina o conjunto infinito de todas as sentenças de L_0, e as expressões que não puderem ser formadas a partir dessas regras não fazem parte dessa língua.

---

[39] Nessas regras, as letras gregas representam variáveis cujo domínio são quaisquer expressões de L_0. Essas regras estabelecem ainda uma quarta categoria sintática em L_0: a das sentenças. Os sinais "$\wedge$", "$\vee$", "$\rightarrow$" e "$\leftrightarrow$" são normalmente empregados para representar os conectivos lógicos respectivamente associados aos seguintes termos da língua: "e", "ou", "se... então...", "se e apenas se... então..."; e seus significados serão apresentados mais adiante. A inserção desses e de alguns outros sinais — "(", ")", ",", "$\neg$", "[" e"]" — é feita sincategorialmente (ou seja, como expressões que não são listadas em nenhuma categoria básica), através apenas das regras de formação; qualquer um desses sinais, no entanto, poderia ser introduzido como expressão básica. Nesse caso, parece que a decisão foi arbitrária; nada impediria, porém, que essa distinção

Dentre essas infinitas sentenças de L_0, 40 são sentenças atômicas estabelecidas pelas regras 1 e 2. Dentre elas estão, por exemplo, "M(d)" e "B(j)", formadas através da regra 1; já as sentenças "K(d,j)" e "L(m,n)", por sua vez, são determinadas pela regra 2.

A partir dessas 40 sentenças atômicas, e através das regras 3 a 7, pode-se constatar ainda que "$\neg$M(d)", "[L(m,n) $\wedge$ B(j)]", "[$\neg$M(d) $\vee$ K(d,j)]", "[[L(m,n) $\wedge$ B(j)] $\rightarrow$ M(d)]" e "[K(d,j) $\leftrightarrow$ [L(m,n) $\wedge$ B(j)]]" também são sentenças de L_0.

Quaisquer outras combinações que não possam ser compostas a partir das definições recursivas do Quadro 1 e das expressões básicas da Tabela 1 não fazem parte de L_0. Assim, apesar de se parecerem de alguma forma com as sentenças atômicas, seqüências de sinais como "$\neg$[M(d)]", "B(j,d)" e "K(n)", por exemplo, não são sentenças de L_0. Em decorrência disso, qualquer expressão complexa formada a partir delas, mesmo que respeite as regras 3 a 7, como "[B(j,d) $\vee$ K(n)]", também não pode ser sentença de L_0. Finalmente, também não fazem parte de L_0 as combinações que não respeitem as regras 3 a 7, ainda que sejam constituídas a partir das sentenças atômicas de L_0 (como "L(m,n) $\wedge$ B(j)", "[$\neg$ $\vee$ K(d,j)]" e "[K(d,j) $\leftrightarrow$ L(m,n) $\rightarrow$ B(j)]", por exemplo).

Assim, pode-se 'provar' que uma determinada sentença faz ou não parte de uma certa língua usando-se o mesmo tipo de procedimento empregado nos manuais de cálculo de predicados. A 'prova' de que a sentença "[K(d,j) $\leftrightarrow$ [L(m,n) $\wedge$ B(j)]]", por exemplo, faz realmente parte de L_0 pode ser apresentada como na Tabela 2.

---

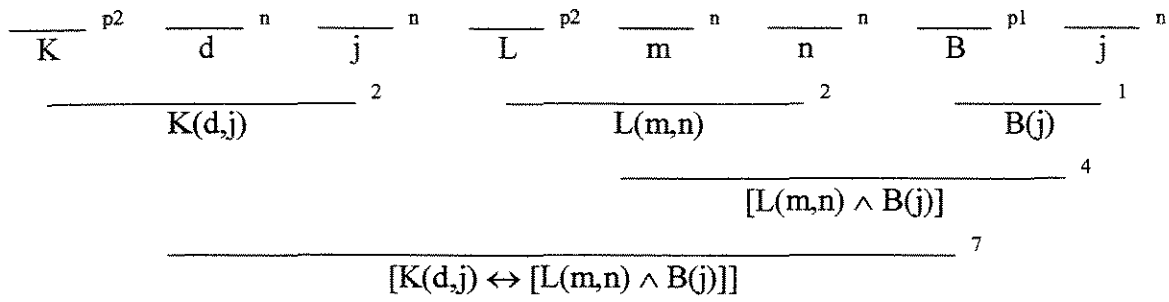
notacional fosse relacionada a alguma distinção funcional (por exemplo, relacionando as categorias lexicais às expressões categoremáticas e as categorias funcionais às expressões sincategoremáticas).

1. K é um predicado de dois lugares	Pela Tabela 1
2. d é um nome	Pela Tabela 1
3. j é um nome	Pela Tabela 1
4. K(d,j) é uma sentença	Por 1, 2, 3 e pela regra 2 do Quadro 1
5. L é um predicado de dois lugares	Pela Tabela 1
6. m é um nome	Pela Tabela 1
7. n é um nome	Pela Tabela 1
8. L(m,n) é uma sentença	Por 5, 6, 7 e pela regra 2 do Quadro 1
9. B é um predicado de um lugar	Pela Tabela 1
10. B(j) é uma sentença	Por 3, 9 e pela regra 1 do Quadro 1
11. [L(m,n) $\wedge$ B(j)] é uma sentença	Por 8, 10 e pela regra 4 do Quadro 1
12. [K(d,j) $\leftrightarrow$ [L(m,n) $\wedge$ B(j)]] é uma sentença	Por 4, 11 e pela regra 7 do Quadro 1

Tabela 2 - Prova de que "[K(d,j) $\leftrightarrow$ [L(m,n) $\wedge$ B(j)]]" é uma sentença de L_0

Outra notação para a apresentação de provas que também tem sido normalmente empregada, pela gramática categorial principalmente, consiste em indicar cada operação através de um traço horizontal sobre a expressão, indexado pela respectiva regra. Além de ser mais prático por evitar as remissões internas, esse método de apresentação ainda tem a vantagem de não sugerir uma solução seqüencial, como o anterior.[40] Esse método de demonstração de prova é conhecido como "estilo de Prawitz", devido ao autor que o empregou pela primeira vez (Prawitz 1965). A prova de que a mesma sentença "[K(d,j) $\leftrightarrow$ [L(m,n) $\wedge$ B(j)]]" faz parte de L_0, segundo esse outro método, pode ser vista no Quadro 2, onde a aplicação das regras é representada por traços indexados, superpostos às expressões que elas licenciam (aqui, as categorias das expressões básicas são indicadas por abreviações dos seus respectivos rótulos: "n" para os nomes, "p1" para os predicados de um lugar, e "p2" para os predicados de dois lugares; as regras empregadas são identificadas pela sua respectiva numeração no Quadro 1).

[40] Na verdade, todos os dois são apenas notações para a apresentação de uma prova, nenhum deles está necessariamente ligado a qualquer método procedimental de resolução. No entanto, pelo seu formato de apresentação linear, a primeira às vezes é confundida com uma solução seqüencial; enquanto a segunda tem inspirado soluções por processamento paralelo.



Quadro 2 - Prova de "[K(d,j) ↔ [L(m,n) ∧ B(j)]]" ao estilo de Prawitz

A definição das sentenças de L_0 em Prolog é feita pelo Programa 4.[41]

```
% L0_Syn2.ari
%    (Dowty, Wall & Peters 1981: 15)
% Formation rules for L0

% Loading basic expressions

:- reconsult('L0_Cat2.ari').

% Formation rules

/*1*/ sentence(F) :- one_place_predicate(D),
                     name(A), F =.. [D, A].
/*2*/ sentence(F) :- two_place_predicate(G),
                     name(A), name(B), F =.. [G, A, B].
/*3*/ sentence([not, F]) :- sentence(F).
/*4*/ sentence([F, and, P]) :- sentence(F), sentence(P).
/*5*/ sentence([F, or, P]) :- sentence(F), sentence(P).
/*6*/ sentence([F, if, P]) :- sentence(F), sentence(P).
/*7*/ sentence([F, iff, P]) :- sentence(F), sentence(P).
```

Programa 4 - Regras de formação para L_0 em Prolog

Como a maioria dos interpretadores de Prolog não dispõe de qualquer recurso gráfico, não se pode representar os conectivos com os símbolos tradicionais do cálculo de predicados. O que se faz normalmente é escrevê-los por extenso, e esta será a solução adotada aqui também.

[41] No Programa 2, primeiro se escolhiam as expressões básicas para só então montar as sentenças atômicas, o que o torna um pouco menos ineficiente para a geração. No Programa 4, como a implementação da Semântica de Montague feita aqui parece ser mais eficiente no reconhecimento do que na geração, as sentenças atômicas são primeiro decompostas em suas expressões básicas e só depois é que se testa se são de categorias adequadas. Essa questão de eficiência em relação à geração e ao reconhecimento tem conseqüências evidentes para discussões psicolingüísticas que não poderão ser feitas aqui.

A definição das sentenças atômicas também exige uma adaptação, pois o Prolog não permite o uso de variáveis na posição de funtor. Para resolver isso, o Prolog dispõe de um operador pré-definido ('=..', apresentado na página 33) que transforma uma lista não-vazia[42] em um predicado cujo funtor é o primeiro elemento dessa lista, e cujos argumentos são os seus outros elementos, na mesma ordem em que eles aparecem na lista; esse operador também atua na ordem inversa, transformando um predicado em uma lista cujo primeiro elemento é o funtor e os argumentos são o seu resto. Assim, o predicado 'a/0' pode ser transformado na lista "[a]"; já os predicados 'a/1' e 'a/2', seguidos respectivamente pelos argumentos 'b' ("a(b)") e 'c,d' ("a(c,d)"), podem ser transformados nas listas "[a,b]" e "[a,c,d]".

Com esse operador, as sentenças atômicas de L_0 podem ser primeiro separadas em suas expressões básicas, para que depois a adequação categorial de cada uma delas possa ser individualmente testada. Depois de definidas as sentenças atômicas, a parte recursiva das regras de formação já pode atuar.

A partir das categorias básicas, estabelecidas no Programa 3, e das regras de formação, definidas no Programa 4, pode-se demonstrar que a sentença "['K'(d, j), iff, ['L'(m, n), and, 'B'(j)]]" faz parte de L_0, através da máquina de inferência do Prolog, de uma forma similar à prova de $[K(d,j) \leftrightarrow [L(m,n) \wedge B(j)]]$.

Essa demonstração começa quando pedimos ao Prolog para satisfazer a cláusula "sentence(['K'(d, j), iff, ['L'(m, n), and, 'B'(j)]]).", e os passos para a sua execução estão esquematizados na Tabela 3.

[42] Uma lista não-vazia é uma lista com pelo menos um elemento; está é a imposição mínima para se construir o predicado mais simples possível, sem nenhum argumento.

Para provar	Casa com	Unifica com	Vai para
1. :- sentence(['K'(d, j), iff, ['L'(m, n), and 'B'(j)]]).	sentence([F, iff, P]) :- sentence(F), sentence(P).	F = 'K'(d, j)	a. 2
		P = ['L'(m, n), and, 'B'(j)]	b. 6
		yes	c. fim
2. :- sentence('K'(d, j)).	sentence(F) :- two_place_predicate(G), name(A), name(B), F =.. [G,A,B].	F = 'K'(d, j)	=..
		G = 'K'	a. 3
		A = d	b. 4
		B = j	c. 5
		yes	d. 1b
3. :- two_place_predicate('K').	two_place_predicate('K').	yes	2b
4. :- name(d).	name(d).	yes	2c
5. :- name(j).	name(j).	yes	2d
6. :- sentence(['L'(m, n), and, 'B'(j)]).	sentence([F, and, P]) :- sentence(F), sentence(P).	F = 'L'(m, n)	a. 7
		P = 'B'(j)	b. 11
		yes	c. 1c
7. :- sentence('L'(m, n)).	sentence(F) :- two_place_predicate(G), name(A), name(B), F =.. [G,A,B].	F = 'L'(m, n)	=..
		G = 'L'	a. 8
		A = m	b. 9
		B = n	c. 10
		yes	d. 6b
8. :- two_place_predicate('L').	two_place_predicate('L').	yes	7b
9. :- name(m).	name(m).	yes	7c
10. :- name(n).	name(n).	yes	7d
11. :- sentence('B'(j)).	sentence(F) :- one_place_predicate(D), name(A), F =.. [D,A].	F = 'B'(j)	=..
		D = 'B'	a. 12
		A = j	b. 13
		yes	c. 6c
12. :- one_place_predicate('B').	one_place_predicate('B').	yes	11b
13. :- name(j).	name(j).	yes	11c

Tabela 3 - Prova em Prolog de "sentence([k(d, j), iff, [l(m, n), and, b(j)]]).

Para provar que "['K'(d, j), iff, ['L'(m, n), and, 'B'(j)]]" é uma sentença (":- sentence(['K'(d, j), iff, ['L'(m, n), and, 'B'(j)]]).", na linha 1), o mecanismo de prova do Prolog identifica que ela casa com a início da regra 7 do programa, unificando as variáveis 'F = 'K'(d, j)' e 'P = ['L'(m, n), and, 'B'(j)]'; com essa unificação, a prova da cláusula inicial passa a depender da prova de duas novas cláusulas: ":- sentence('K'(d, j))." (na linha 2) e ":- sentence(['L'(m, n), and, 'B'(j)]])." (na linha 6).

A primeira destas cláusulas casa com o início da segunda regra de formação no programa, unificando a variável 'F = 'K'(d, j)'; depois, através do operador "=..", são

unificadas também as variáveis 'G = 'K'', 'A = d' e 'B = j'. E assim, são criadas mais três cláusulas que precisam ser satisfeitas: ":- two_place_predicate('K')." (na linha 3), ":- name(d)." (na linha 4) e ":- name(j)." (na linha 5), que são trivialmente provadas através do Programa 3.[43] Com isso, fica provado que "'K'(d, j)" é uma sentença.

Para continuar, o mecanismo de prova recorre à regra 4 do programa, cujo início casa com a segunda das novas cláusulas, através da unificação das variáveis 'F = 'L'(m, n)' e 'P = 'B'(j)'. Assim, outras duas novas cláusulas também precisam ser provadas: ":- sentence('L'(m, n))." (na linha 7) e ":- sentence('B'(j))." (na linha 11).

A prova de ":- sentence('L'(m, n))." é semelhante à de ":- sentence('K'(d, j)).": ela casa com o início da segunda regra de formação, causando a unificação das variáveis 'G = 'L'', 'A = m' e 'B = n' (na linha 7); aqui também os testes categoriais são triviais (nas linhas 8, 9 e 10), o que termina esta sub-prova.

Finalmente, a prova de ":- sentence('B'(j))." é feita com o casamento desta com o início da primeira regra de formação, o que unifica as variáveis 'D = 'B'' e 'A = j'. As duas últimas cláusulas a serem satisfeitas são: ":- one_place_predicate('B')." (na linha 12) e ":- name(j)." (na linha 13), que também são trivialmente provadas. Com todas as sub-cláusulas provadas, termina a prova de ":- sentence(['K'(d, j), iff, ['L'(m, n), and, 'B'(j)]])."

[43] No final de cada um desses passos, na última célula das respectivas linhas, indica-se o local por onde aquela sub-prova deve continuar.

4.1.2. Semântica de L_0

Para que os valores semânticos das expressões compostas de L_0 possam ser atribuídos de acordo com o princípio de composicionalidade,[44] segundo as regras sintáticas apropriadas, é preciso primeiro que a cada expressão básica corresponda algum valor semântico.

Os valores semânticos dos nomes de L_0 são indivíduos, atribuídos conforme a Tabela 4. Segundo DWP: 18, esses valores não são apenas "entidades lingüísticas, mas sim *entidades do 'mundo real'*"; assim, os valores semânticos dos nomes de L_0 são os próprios indivíduos, e não apenas os nomes que os designam.[45] Ainda é necessário que a cada expressão básica corresponda um único valor semântico (nos casos de ambigüidade lexical, os nomes com mais de um valor devem ser tratados como itens lexicais diferentes), o que vai permitir que toda expressão conexa tenha sempre o seu respectivo valor semântico.

Nome	**Valor semântico**
d	Richard Nixon
n	Noam Chomsky
j	John Mitchell
m	Muhammad Ali

Tabela 4 - Valores semânticos dos nomes de L_0

Aos predicados de um lugar serão atribuídos conjuntos de indivíduos como seus valores semânticos: "intuitivamente, o conjunto de indivíduos para o qual o predicado é verdadeiro" (DWP: 18). A título de ilustração, os autores sugerem que o predicado 'B'

---

[44] O princípio de composicionalidade, comumente atribuído a Frege, diz que o significado de uma expressão complexa é constituído pelo significado das partes que o compõem, levando-se em consideração a maneira com que essas partes são combinadas.

[45] Essa opção extensional é apenas didática. Segundo o professor José Borges Neto (comunicação pessoal), o importante na semântica de Montague é a formalização da relação entre as expressões lingüísticas e os seus respectivos valores semânticos sejam lá o que eles possam vir a ser.

corresponda ao conjunto de indivíduos carecas (*bald*) e o predicado 'M' ao conjunto de indivíduos que usam bigode (*mustache*).[46]

Com essas informações, já é possível perceber como o valor semântico de algumas sentenças de L_0 podem ser determinados a partir dos valores semânticos de suas expressões básicas e das regras sintáticas que a compõem. Uma sentença como 'M(j)' estabelece que o indivíduo John Mitchell (denotado pelo nome 'j') faz parte do conjunto dos indivíduos que usam bigode (denotado pelo predicado de um lugar 'M'). Empregando a convenção notacional usual, pode-se dizer que o valor semântico de qualquer sentença formada por um nome α e por um predicado de um lugar δ é $[\![\alpha]\!] \supseteq [\![\delta]\!]$; ou, para ser ainda mais fiel: a sentença $\delta(\alpha)$ só é verdadeira se $[\![\alpha]\!] \in [\![\delta]\!]$.[47]

Para os predicados de dois lugares, o valor semântico corresponde a conjuntos de pares de indivíduos; mais especificamente, o valor semântico desses predicados é o conjunto de pares ordenados para os quais o predicado é verdadeiro. Uma maneira ainda mais composicional de entender o valor semântico desses predicados consiste em apresentá-los como funções que tomam um argumento, resultando então num predicado de um lugar (que, por sua vez, ainda é uma função que toma outro argumento para ficar completamente saturada, sem mais argumentos a serem preenchidos). Ainda a título de ilustração, os autores sugerem que o predicado 'K' designe o conjunto dos pares tal que o primeiro conheça (*to know*) o segundo, e o predicado 'L' designe o conjunto dos pares tal que o primeiro ame (*to love*) o segundo.

---

[46] Uma das desvantagens da abordagem extensional é que esses conjuntos precisariam conter **todos** os indivíduos daquela determinada classe. Uma perspectiva mais natural, segundo a qual o predicado designa uma característica compartilhada pelos membros daquele conjunto, só fica disponível quando se introduz a abordagem intensional. Um conjunto é designado intensionalmente através de alguma propriedade (como em {x | x é um 'numero inteiro ímpar}); na abordagem extensional, os conjuntos são designados pela listagem exaustiva de seus membros (assim, seria impossível designar o conjunto dos números inteiros ímpares, a não ser abreviadamente como {3, 5, 7, 9...}. Como não chegaremos à abordagem intensional nessa tese, vamos nos ater à abordagem extensional.

[47] A primeira das notações é lida como "o valor semântico de α está contido no valor semântico de δ", enquanto a segunda como "o valor semântico de α pertence ao valor semântico de δ". Como é possível perceber também, nem sempre o editor de equações usado consegue lidar com a notação (o que era para ser um colchete com barras duplas é mostrado como dois colchetes depois do δ, que é como ele é feito efetivamente; mas isso não acontece depois do α).

Assim, finalmente, pode-se determinar os valores semânticos das sentenças atômicas restantes. Uma sentença como "K(d, j)", por exemplo, pode ter o valor semântico expresso como 'os indivíduos Richard Nixon e John Mitchell, exatamente nessa ordem, fazem parte dos pares de indivíduos nos quais o primeiro conhece o segundo'. Generalizando, o valor semântico de uma sentença $\gamma(\alpha, \beta)$, composta pelos nomes α e β, e pelo predicado de dois lugares γ, pode ser representado pela seguinte notação: $\langle [\![\alpha]\!], [\![\beta]\!] \rangle \subseteq [\![\gamma]\!]$ (ou ainda $\langle [\![\alpha]\!], [\![\beta]\!] \rangle \in [\![\gamma]\!]$).[48]

Para as sentenças não-atômicas de L_0, a atribuição de seus valores de verdade é determinada pelo valor das sentenças que a compõem e pela colaboração das operações semânticas dos seus conectivos. Esse valor semântico dos conectivos é normalmente apresentado em manuais de introdução ao cálculo de predicados através de tabelas de verdade, como as que estão esquematizadas na Tabela 5.[49]

ϕ	ψ	$[\phi \leftrightarrow \psi]$	$[\phi \rightarrow \psi]$	$[\phi \vee \psi]$	$[\phi \wedge \psi]$	$\neg\phi$
F	F	V	V	F	F	V
F	V	F	V	V	F	
V	F	F	F	V	F	F
V	V	V	V	V	V	

Tabela 5 - Valores de verdade para os conectivos de L_0

Todas essas regras para atribuição de valores semânticos às expressões compostas podem ser declarativamente apresentadas como no Quadro 3 (DWP: 21).

[48] Ou seja, os valores semânticos de α e de β constituem um par ordenado que *está contido no* (ou *pertence ao*) valor semântico de γ.

[49] Como de costume, "F" representa o falso e "V" o verdadeiro.

1. Se δ é um predicado de um lugar e α é um nome, então $\delta(\alpha)$ é verdadeira se e apenas se $[\![\alpha]\!] \in [\![\delta]\!]$.
2. Se γ é um predicado de dois lugares e α e β são nomes, então $\gamma(\alpha, \beta)$ é verdadeira se e apenas se $\langle [\![\alpha]\!], [\![\beta]\!] \rangle \in [\![\gamma]\!]$.
3. Se ϕ é uma sentença, então $\neg\phi$ é verdadeira se e apenas se ϕ for falsa.
4. Se ϕ e ψ são sentenças, então $[\phi \wedge \psi]$ é verdadeira se e apenas se tanto ϕ quanto ψ forem ambas verdadeiras.
5. Se ϕ e ψ são sentenças, então $[\phi \vee \psi]$ é verdadeira se e apenas se ϕ ou ψ forem uma das duas verdadeiras.
6. Se ϕ e ψ são sentenças, então $[\phi \rightarrow \psi]$ é verdadeira se e apenas se ϕ for falsa ou então ψ verdadeira.
7. Se ϕ e ψ são sentenças, então $[\phi \leftrightarrow \psi]$ é verdadeira se e apenas se tanto ϕ quanto ψ sejam ambas verdadeiras ou ambas falsas.

Quadro 3 - Regras semânticas para L_0

Ainda a título de ilustração, os autores supõem que os seguintes predicados sejam verdadeiros: "M(j)" (John Mitchell usa bigode), "K(d, j)" (Richard Nixon conhece John Mitchell), e "B(j)" (John Mitchell é careca). Como esse é um mundo habitado exclusivamente por homens, para evitarmos sugestões embaraçosas para qualquer um desses indivíduos, estamos supondo que o valor semântico do predicado 'L' seja o conjunto vazio (ou seja, este é um mundo onde ninguém ama ninguém). Assim, usando a mesma convenção notacional introduzida acima, $[\![M]\!]$ = {John Mitchell}, $[\![B]\!]$ = {John Mitchell}, $[\![K]\!]$ = {⟨Richard Nixon, John Mitchell⟩} e $[\![L]\!] = \varnothing$.[50]

Através dos mesmos métodos para provar que uma sentença fazia parte de L_0, podemos igualmente comprovar o seu valor semântico. A sentença "[K(d, j) ∧ B(j)]", por exemplo, pode ser demonstrada verdadeira segundo a Tabela 6.

[50] Seguindo a notação usada na teoria de conjuntos, as chaves indicam conjuntos. Assim, o valor semântico tanto para 'M' quanto para 'B' é o conjunto unitário formado apenas pelo indivíduo John Mitchell — o que não significa que eles sejam sinônimos, essa é só uma coincidência extensional. O valor semântico de 'K' é um conjunto de um único par ordenado formado pelos indivíduos John Mitchell e Richard Nixon, exatamente nessa ordem. Ainda da notação da teoria de conjuntos, empregamos o símbolo ∅ para designar o conjunto vazio.

a. K é um predicado de dois lugares	Pela lista das expressões básicas
b. $[\![K]\!]$ = {⟨Richard Nixon, John Mitchell⟩}	Pelas atribuições supostas
c. d é um nome	Pela lista das expressões básicas
d. $[\![d]\!]$ = Richard Nixon	Pelas atribuições supostas
e. j é um nome	Pela lista das expressões básicas
f. $[\![j]\!]$ = John Mitchell	Pelas atribuições supostas
g. $[\![K(d,j)]\!] = 1$	Por a, b, c, d , e, f e pela regra 2 da Tabela 5
h. B é um predicado de um lugar	Pela lista das expressões básicas
i. $[\![B]\!]$ = {John Mitchell}	Pelas atribuições supostas
j. $[\![B(j)]\!] = 1$	Por h, i, e, f e pela regra 1da Tabela 5
k. K(d, j) é uma sentença	Pelas regras de formação
l. B(j) é uma sentença	Pelas regras de formação
m. $[\![K(d,j) \wedge B(j)]\!] = 1$	Por k, l, g, j e pela regra 4 da Tabela 5

Tabela 6 - Prova da verdade de "K(d, j) ∧ B(j)"

Para reproduzir automaticamente essa prova em Prolog, é necessário declarar o valor semântico de cada expressão básica, como se pode ver no Programa 5. Aqui a atribuição é feita através do predicado 'semantic_value/2', que toma uma expressão básica como seu primeiro argumento e seu respectivo valor semântico como segundo argumento.

```
% L0_Att.ari
%    (Dowty, Wall & Peters 1981: 17-20)
% Semantic values for basic expressions of L0

% Names

semantic_value(d, [richard, nixon]).
semantic_value(n, [noam, chomsky]).
semantic_value(j, [john, mitchell]).
semantic_value(m, [muhammad, ali]).

% One place predicates

semantic_value('M', [john, mitchell]).
semantic_value('B', [john, mitchell]).

% Two place predicates

semantic_value('K', [[richard, nixon], [john, mitchell]]).
```

Programa 5 - Valores semânticos das expressões básicas de L_0 em Prolog

Além da atribuição dos valores semânticos às expressões básicas, precisamos ainda traduzir em Prolog as regras para a atribuição dos valores semânticos às sentenças atômicas e para as não-atômicas, apresentadas declarativamente no Quadro 3 e esquematicamente na Tabela 5. O Programa 6, abaixo, executa em Prolog todas essas regras.

```
% L0_Sem1.ari
%     (Dowty, Wall & Peters 1981: 20-21)
% Semantics of L0 (Reference to lexicon and syntax)

% Loading syntactic rules

:- reconsult('L0_Syn2.ari').

% Loading basic attributions

:- reconsult('L0_Att.ari').

% Rules

/*1*/ semantic_value(F) :-
       one_place_predicate(D), name(A), F =.. [D, A],
       semantic_value(A, X), semantic_value(D, X).
/*2*/ semantic_value(F) :-
       two_place_predicate(G), F =.. [G, A, B],
       name(A), name(B), semantic_value(A, X),
       semantic_value(B, Y), semantic_value(G, [X, Y]).
/*3*/ semantic_value([not, F]) :-
       sentence(F), not semantic_value(F).
/*4*/ semantic_value([F, and, P]) :-
       sentence(F), sentence(P),
       semantic_value(F), semantic_value(P).
/*5*/ semantic_value([F, or, P]) :-
       sentence(F), sentence(P),
       (semantic_value(F); semantic_value(P)).
/*6*/ semantic_value([F, if, P]) :-
       sentence(F), sentence(P),
       (not semantic_value(F); semantic_value(P)).
/*7*/ semantic_value([F, iff, P]) :-
       sentence(F), sentence(P),
       ((semantic_value(F), semantic_value(P));
        (not semantic_value(F), not semantic_value(P))).
```

Programa 6 - Semântica para L_0 sem uniformização categorial

Nesse Programa 6, é possível perceber alguma assimetria no fato de que as duas primeiras regras invocam predicados e nomes, enquanto as outras se referem sempre a sentenças. Escritas da forma como estão, as regras 1 e 2 executam novamente operações que já foram realizadas pelas respectivas regras sintáticas: as categorias de todas as

expressões básicas nas sentenças atômicas são testadas tanto pela regra sintática quanto pela regra semântica correspondente (por exemplo: "se δ é um predicado de um lugar", quando declarado, e "one_place_predicate(D)", em Prolog, que aparecem tanto nas regras sintáticas quanto nas semânticas). Uma maneira de reduzir esse problema é regularizar a referência categorial das regras semânticas apenas às sentenças, como no Programa 7.

```
% L0_Sem2.ari
%      (Dowty, Wall & Peters 1981: 21)
% Semantics of L0 (Optimized exclusive reference to syntax)

% Loading syntactic rules

:- reconsult('L0_Syn2.ari').

% Loading basic attributions

:- reconsult('L0_Att.ari').

% Rules

/*1*/ semantic_value(F) :- sentence(F), F =.. [D, A],
       semantic_value(A, X), semantic_value(D, X).
/*2*/ semantic_value(F) :- sentence(F), F =.. [G, A, B],
       semantic_value(A, X), semantic_value(B, Y),
       semantic_value(G, [X, Y]).
/*3*/ semantic_value([not, F]) :- not semantic_value(F).
/*4*/ semantic_value([F, and, P]) :- semantic_value(F),
       semantic_value(P).
/*5*/ semantic_value([F, or, P]) :-
       (semantic_value(F); semantic_value(P)).
/*6*/ semantic_value([F, if, P]) :-
       (not semantic_value(F); semantic_value(P)).
/*7*/ semantic_value([F, iff, P]) :-
       ((semantic_value(F), semantic_value(P));
        (not semantic_value(F), not semantic_value(P))).
```

Programa 7 - Semânticas para L_0 com uniformização categorial

Além da homogeneização da referência categorial, no Programa 7 também foram evitadas as desnecessárias menções às categorias das subpartes das sentenças compostas, porque estas são sempre testadas pela regra que avalia o valor semântico das sentenças atômicas. Na Tabela 7, abaixo, onde estão indicados os passos percorridos por um interpretador Prolog para provar "semantic_value([not, 'M'(n)]).", através do Programa

6, pode-se ver o mesmo teste para a sentença "'M'(n)" sendo invocado duas vezes: na linha 1 e depois na 2.[51]

Para provar	Casa com	Unificando	Vai para
1. semantic_value([not, 'M'(n)]).	semantic_value([not, F]) :- sentence (F), not semantic_value(F).	F = 'M'(n)	a. 2
		yes	b. fim
2. semantic_value('M'(n)).	semantic_value(F) :- sentence (F), F =.. [D, A], semantic_value(A, X), semantic_value(D, X).	F = 'M'(n)	
		A = n	a. 3
		D = m	b. 4
		no	c. 1b
3. semantic_value(n, X).	semantic_value(n, [noam, chomsky]).	X = [noam, chomsky]	2b
4. semantic_value('M', [noam, chomsky]).		no	2c

Tabela 7 - Exemplo de prova em Prolog pelo Programa 6

Excluídas essas pequenas diferenças da referência categorial homogeneizada e de sua otimização, ambos os programas são idênticos. Para as sentenças simples, o primeiro passo é desmembrá-las em suas expressões básicas ("F =.. [D, A]", para as com predicado de um lugar, e "F =.. [G, A, B]", para as com predicado de dois lugares), das quais se recupera os seus respectivos valores semânticos (através do predicado "semantic_value/2" aplicado a cada uma delas), incluindo nessas operações as restrições de unificação das variáveis de modo a satisfazer as exigências de compartilhamento entre os seus respectivos valores. Assim, na regra 1, "semantic_value(A, X), semantic_value(D, X)" garante que o valor semântico de 'A' (que, devido às regras sintáticas, só pode ser um nome) precisa estar na lista de valores semânticos de 'D' (que, ainda pelas regras sintáticas, só pode ser um predicado de um lugar); enquanto na regra 2, "semantic_value(A, X), semantic_value(B, Y), semantic_value(G, [X, Y])" garante que os valores semânticos de 'A' e de 'B' (também nomes) precisam constar, nessa mesma ordem, do conjunto de pares ordenados que corresponde ao valor semântico de 'G' (um predicado de dois lugares).

As sentenças compostas, por sua vez, podem ser diretamente traduzidas em Prolog, usando o operador 'not' e a disjunção ';'. Assim, na regra 3, o valor semântico de

[51] Como as provas sintáticas já foram apresentadas na seção anterior, elas não serão incluídas aqui.

"[not, F]" vai depender de que o valor semântico de 'F' não seja verdadeiro ("not semantic_value(F)"); na regra 4, o valor de "[F, and, P]" depende tanto que o valor de 'F' quanto o de 'P' sejam ambos verdadeiros ("semantic_value(F), semantic_value(P)"); na regra 5, o valor de "[F, or, P]" só é verdadeiro se algum dos valores de 'F' e de 'P' (ou ambos) forem verdadeiros ("(semantic_value(F); semantic_value(P))"); na regra 6, o valor de "[F, if, P]" depende de que ou o valor de 'F' seja falso ou que o de 'P' seja verdadeiro ("(not semantic_value(F); semantic_value(P))"); e na regra 7, finalmente, os valores de 'F' e 'P' devem ser ambos verdadeiros ou ambos falsos para que "[F, iff, P]" seja verdadeiro ("((semantic_value(F), semantic_value(P)); (not semantic_value(F), not semantic_value(P)))").

A partir do Programa 7, seguido os passos apresentados na Tabela 8, um interpretador Prolog pode provar automaticamente "semantic_value(['K'(d, j), and, 'B'(j)]).", ou seja, que a sentença "['K'(d, j), and, 'B'(j)]" é verdadeira quando usamos L_0 para descrever os fatos supostos no Programa 5.[52]

[52] Aqui também omitiremos as provas sintáticas.

Para provar	Casa com	Unificando	Vai para
1. semantic_value(['K'(d, j), and 'B'(j)]).	semantic_value([F, and, P]) :- semantic_value(F), semantic_value(P).	F = 'K'(d, j)	a. 2
		P = 'B'(j)	b. 6
		yes	c. fim
2. semantic_value('K'(d, j)).	semantic_value(F) :- sentence(F), F =.. [G, A, B], semantic_value(A, X), semantic_value(B, Y), semantic_value(G, [X, Y]).	F = 'K'(d, j)	
		A = d	a. 3
		B = j	b. 4
		G = 'K'	c. 5
		yes	d. 1b
3. semantic_value(d, X).	semantic_value(d, [richard, nixon]).	X = [richard, nixon]	2b
4. semantic_value(j, Y).	semantic_value([j, [john, mitchell]).	Y = [john, mitchell]	2c
5. semantic_value(['K', [[richard, nixon], [john, mitchell]]]).	semantic_value(['K', [[richard, nixon], [john, mitchell]]]).	yes	2d
6. semantic_value('B'(j)).	semantic_value (F) :- sentence(F), F = [D, A], semantic_value(A, X), semantic_value(D, X).	F = 'B'(j)	
		A = j	a. 7
		D = 'B'	b. 8
		yes	c. 1c
7. semantic_value(j, X).	semantic_value(j,[john, mitchell]).	X = [john, mitchell]	6b
8. semantic_value('B', [john, mitchell]).	semantic_value('B', [john, mitchell]).	yes	6c

Tabela 8 - Exemplo de prova em Prolog pelo Programa 7

Para provar que "['K'(d, j), and, 'B'(j)]" é verdadeira, o Prolog casa a chamada "semantic_value(['K'(d, j), and, 'B'(j)])." com a quarta regra do Programa 7 ("semantic_value([F, and, P]) :- semantic_value(F), semantic_value(P)."), na linha 1, unificando as variáveis 'F = 'K'(d, j)' e 'P = 'B'(j)'. Assim, para continuar a prova, o Prolog precisa satisfazer então "semantic_value('K'(d, j).", na linha 2, que casa com a segunda regra do Programa 7 ("semantic_value(F) :- sentence(F), F =.. [G, A, B], semantic_value(A, X), semantic_value(B, Y), semantic_value(G, [X, Y])."), unificando 'F = 'K'(d, j)', 'G = 'K'', 'A = d' e 'B = j'; a partir das variáveis 'A' e 'B', o Prolog instancia as variáveis 'X = [richard, nixon]' e 'Y = [john, mitchell]', através das atribuições básicas do Programa 5. Com isso, resta apenas satisfazer "semantic_value(['K', [[richard, nixon], [john, mitchell]]).", que também se encontra na lista de atribuições do Programa 5.

Para terminar a prova, o Prolog precisa satisfazer ainda a segunda parte da primeira regra invocada: "semantic_value('B'(j)).", na linha 6, que casa com a primeira regra do Programa 7 ("semantic_value (F) :- sentence(F), F = [D, A], semantic_value(A, X), semantic_value(D, X)."), unificando 'F = 'B'(j)', 'D = 'B'' e 'A = j'. Novamente através do Programa 5, e a partir da variável 'A', a variável 'X' é instanciada com o valor "[john, mitchell]"; e, finalmente, "semantic_value('B', [john, mitchell])." também pode ser satisfeito, porque consta da lista das atribuições básicas.

Assim, tendo obtido sucesso em todas as sub-provas, o interpretador Prolog responde afirmativamente a pergunta inicial "?- semantic_value(['K'(d, j), and, 'B'(j)]).", o que significa que a sentença "['K'(d, j), and, 'B'(j)]" é verdadeira para L_0 segundo as atribuições básicas supostas aqui.

4.2. Língua L_{0E}

4.2.1. Sintaxe de L_{0E}

Para a língua L_{0E}, diferentemente da opção para L_0, os autores preferiram uma gramática de estrutura sintagmática (*phrase structure grammar*),[53] conforme o Quadro 4.[54]

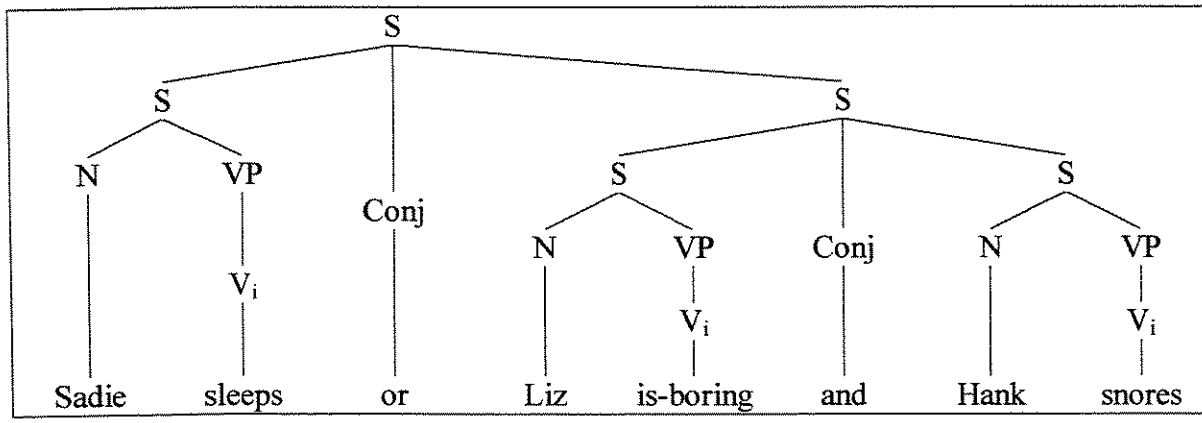
$S \rightarrow S\ Conj\ S$	$Conj \rightarrow$ and, or
$S \rightarrow Neg\ S$	$N \rightarrow$ Sadie, Liz, Hank
$S \rightarrow N\ VP$	$V_i \rightarrow$ snores, sleeps, is-boring
$VP \rightarrow V_i$	$V_t \rightarrow$ loves, hates, is-taller-than
$VP \rightarrow V_t\ N$	$Neg \rightarrow$ it-is-not-the-case-that

Quadro 4 - Gramática de estrutura sintagmática para L_{0E}

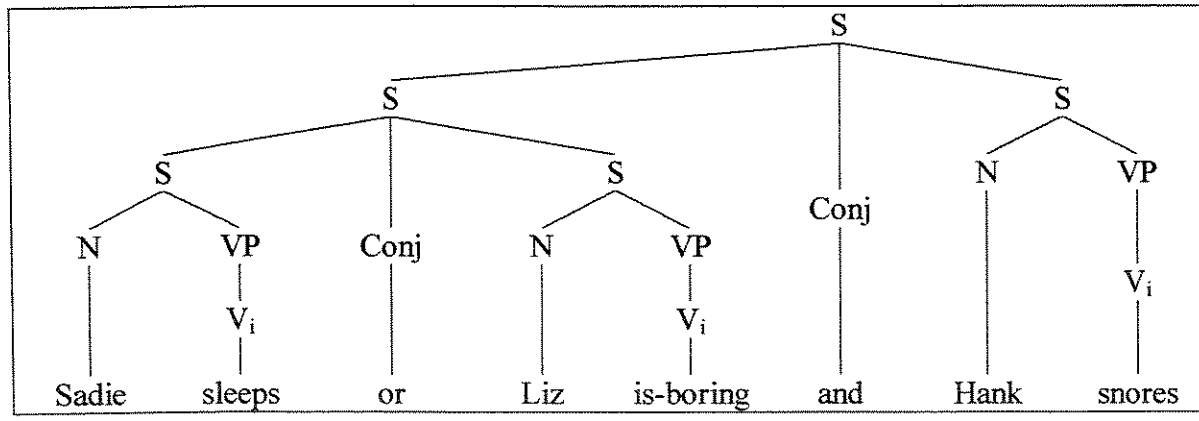
[53] Como a semântica de Montague não pressupõe um tipo específico de sintaxe, contanto que ela também seja completamente formalizável, essa mudança da abordagem definicional em L_0 para a de gramática de reescrita em L_{0E} geralmente serve para reforçar essa flexibilidade em relação à representação sintática.

[54] Como é possível perceber através dessa flexibilidade de escolha, a semântica de Montague não está comprometida com um tipo específico de sintaxe: basta apenas que a gramática permita uma análise estrutural formal.

Com essa gramática de reescrita independente de contexto (*context-free*), é possível representar a derivação de qualquer sentença de L_{0E} através de um diagrama em árvore, mesmo que uma única sentença apresente mais de uma derivação, como no Quadro 5 e no Quadro 6.



Quadro 5 - Ambigüidade estrutural com ramificação à direita



Quadro 6 - Ambigüidade estrutural com ramificação à esquerda

Como a língua L_{0E} permite a formação de uma mesma sentença através do uso de regras diferentes (como "Sadie sleeps or Liz is-boring and Hank snores"), com o diagrama em árvore pode-se representar essa ambigüidade estrutural através das diferentes ramificações nos distintos diagramas que uma mesma sentença permitir. Assim, no Quadro 5, a análise estrutural é obtida aplicando-se em sentido descendente (*top-down*) [55] primeiro a regra 'S → S Conj S', para depois aplicar a regra 'S → N VP' no

[55] Uma análise descendente á a que começa pelas regras de reescrita (mais especificamente, pelo lado esquerdo delas) para depois chegar à sentença.

primeiro *S* (obtendo-se "[N VP] Conj S") e a seguir novamente a regra de coordenação no segundo *S* (resultando em "[N VP] Conj [S Conj S]"); só então é que a regra para sentenças atômicas é aplicada mais duas vezes (chegando-se finalmente a "[N VP] Conj [[N VP] Conj [N VP]]"). Já no Quadro 6, depois da primeira aplicação da regra 'S → S Conj S', ela já é reaplicada no primeiro *S* ("[S Conj S] Conj S"); daí então a regra 'S → N VP' é aplicada a cada um dos três *Ss* ("[[N VP] Conj [N VP]] Conj [N VP]"). Dessa forma, obtém-se uma estrutura ramificada à direita no Quadro 5 e outra ramificada à esquerda no Quadro 6. Com esse procedimento, a análise semântica não precisa lidar com ambigüidades criadas estruturalmente, porque estas já foram explicitadas sintaticamente.

A forma mais evidente de se implementar em Prolog esse tipo de gramática de estrutura sintagmática é escrevê-la, como no Programa 8, usando o formato de DCG.[56]

```
% L0e_Syn1.ari
%     (Dowty, Wall & Peters 1981: 23)
% Syntax of L0e (DCG parser)

% Formation rules

s --> n, vp.
s --> neg, s.
s --> s, conj, s.

vp --> vi.
vp --> vt, n.

% Basic expressions

conj --> [and].
conj --> [or].

n --> [sadie].
n --> [liz].
n --> [hank].

vi --> [snores].
vi --> [sleeps].
vi --> [is, boring].

vt --> [loves].
vt --> [hates].
vt --> [is, taller, than].
```

[56] Sobre a DCG, ver seção 3.4.9, na página 40

```
neg --> [it, is, not, the, case, that].
```

Programa 8 - Sintaxe de L_{0E} em Prolog usando DCG

À primeira vista, essa implementação parece ser satisfatória principalmente porque permite representar as locuções que os autores preferiram representar como unidades lexicais ("is-boring", "is-taller-than" e "it-is-not-the-case-that") mais naturalmente, como listas de unidades menores — "[is, boring]", "[is, taller, than]" e "[it, is, not, the, case, that]". No entanto, como o interpretador Prolog funciona descendentemente e da esquerda para a direita, gramáticas com recursividade à esquerda, como a proposta por DWP: 23 para L_{0E}, criam problemas para o processamento computacional (nesse caso, devido especificamente à regra "s --> s, conj, s."), já exaustivamente relatados na literatura sobre Prolog (como, por exemplo, em Covington 1994: 48). Com o Programa 8, conseguimos obter apenas a análise com ramificação à direita, apresentada no Quadro 5. Mas o interpretador Prolog ainda indica que existem outras possibilidades de análise disponíveis; caso ele seja solicitado a apresentá-las, a resposta será a de insuficiência de memória, porque o interpretador é induzido a uma recursão infinita. (E só obtemos alguma resposta porque a regra de recursão à esquerda foi estrategicamente posicionada na última cláusula da definição de sentença; se ela fosse a primeira, como aparece no Quadro 4, a recursão infinita ocorreria imediatamente e não se chegaria a nenhuma análise.)

Uma das maneiras de resolver esse problema é empregar uma técnica de análise de tipo ascendente (*bottom-up*),[57] como em Covington 1994: 155-158,[58] representada aqui na Tabela 9.[59] Essa técnica, conhecida como análise gramatical por deslocamento-e-redução (*shift-reduce parsing*), consiste basicamente em duas operações: 1) o

[57] Ao contrário da análise descendente, a análise ascendente começa identificando as unidades sentenciais para só depois começar a aplicar as regras de reescrita, usando o lado direito delas.

[58] Uma explicação que também inclui alguma reflexão psicolingüística pode ser encontrada em Pereira 1985.

[59] Outra alternativa seria usar "técnicas para remover a recursão à esquerda das gramáticas independentes de contexto" (Pereira & Shieber 1987: 179); no entanto, essa alternativa é considerada menos adequada tanto descritiva quanto explicativamente.

deslocamento das expressões básicas, uma a uma, para uma pilha (*stack*)[60] e 2) a aplicação de redução dessa pilha segundo as regras estabelecidas pela gramática. Na Tabela 9, essa técnica foi empregada para analisar a sentença "Liz is-taller-than Sadie", considerando a mesma gramática apresentada no Quadro 4.

	Expressão	Pilha	Operação
1.	Liz is-taller-than Sadie		
2.	is-taller-than Sadie	Liz	deslocamento
3.	is-taller-than Sadie	N	redução (N → Liz)
4.	Sadie	N is-taller-than	deslocamento
5.	Sadie	N V_t	redução (V_t → is-taller-than)
6.		N V_t Sadie	deslocamento
7.		N V_t N	redução (N → Sadie)
8.		N VP	redução (VP → V_t N)
9.		S	redução (S → N VP)

Tabela 9 - Análise de "Liz is-taller-than Sadie" por deslocamento-e-redução

A análise começa na linha 1, apenas com a expressão de entrada. Na linha 2, como não havia antes nenhum elemento na pilha que permitisse qualquer redução,[61] o primeiro elemento da expressão de entrada ("Liz") é deslocado para a pilha (de forma que a expressão de entrada passa a ser "is-taller-than Sadie"). Agora com um elemento na pilha, pode-se tentar a primeira redução: como a pilha contém um elemento que faz parte do corpo de uma regra de inserção lexical ("N → Liz" da coluna à direita no Quadro 4), esse elemento pode ser reduzido de 'Liz' para 'N' (linha 3); como nenhuma outra redução pode ser aplicada, outro deslocamento pode ocorrer (linha 4). Agora a pilha contém "N is-taller-than", à qual pode-se aplicar uma redução, devido à inserção lexical "V_t → is-taller-than" (também da coluna à esquerda no Quadro 4), como na linha 5; mas como não há outra regra que permita mais reduções, só nos resta aplicar novamente o deslocamento (linha 6), consumindo todos os elementos da expressão de entrada, o que

[60] Uma pilha é uma estrutura de armazenamento de informação onde o único elemento acessível é o último armazenado (como se estivessem 'empilhados'); os outros elementos anteriores só se tornam acessíveis na medida em que os posteriores vão sendo removidos. Ou seja, na pilha "a b c d", o elemento 'b' só pode ser acessado depois que os elementos 'd' e 'c' forem removidos (supondo que o empilhamento ocorre pela direita).

[61] Na análise gramatical por deslocamento-e-redução, a redução tem prioridade sobre o deslocamento; ou seja, só se aplica o deslocamento quando nenhuma redução puder mais ser aplicada.

significa que a operação de deslocamento não está mais disponível para essa análise. No entanto, a regra "N → Sadie" permite a redução da pilha para "N V_t N" (linha 7), depois a regra "VP → V_t N" permite uma segunda redução para "N VP" (linha 8), e finalmente a regra "S → N VP" possibilita a última redução (linha 9), fazendo com que a análise atinja sua condição de término — que se caracteriza pelo esvaziamento da expressão de entrada e pela redução da pilha ao termo 'S'.[62]

Para um analisador gramatical por deslocamento-e-redução em Prolog, é preciso começar a conceber a pilha como uma lista no sentido inverso ao apresentado, já que uma lista é sempre definida em Prolog como 'um elemento atômico seguido de outra lista' (assim, a lista "[a, b, c]" é, na verdade, composta pelo átomo 'a' seguido da lista "[b, c], que por sua vez é composta do átomo 'b' seguido da lista "[c]", que ainda é o átomo "c" seguido pela lista vazia; ou seja, "[a | [b | [c | []]]]"). Essa concepção de lista replica exatamente a estrutura de uma pilha, com o empilhamento ocorrendo à esquerda, através do primeiro elemento da lista.

Outra simplificação que também pode ser feita, enquanto pudermos pressupor que todas as análises vão começar com a pilha vazia, é inverter a prioridade da redução, tornando o deslocamento a primeira operação, já que nunca no início vai haver nada mesmo na pilha para ser reduzido.

As regras também precisam ser escritas em sentido inverso, para facilitar a sua aplicação durante as operações de redução da pilha. Assim, uma regra como "S → S Conj S" precisa ser representada em Prolog como "rule([s, conj, s | X], [s | X]).", o que significa que sempre que tivermos uma pilha que comece pela seqüência "s, conj, s", podemos reduzi-la a um único "s". Ainda para permitir a conversão das expressões básicas em suas categorias, optou-se pela inclusão de uma última regra que troca na pilha a expressão básica pela sua respectiva categoria — outra opção seria incluir uma cláusula especial para isso no início da definição de redução ("reduce([Word | X], ReducedStack)

---

[62] Caso a pilha não tivesse chegado a um único 'S' ou ainda restasse algum elemento na coluna da expressão, o analisador precisaria acusar a impossibilidade da análise.

:- word(Category, Word), reduce([Category | X], ReducedStack).”), com a conseqüente eliminação da cláusula responsável por isso na definição das regras.[63]

Finalmente, as expressões básicas podem ser representadas junto com suas respectivas categorias através da definição das expressões básicas. Todas essas características estão implementadas no Programa 9. Como é possível perceber, além das expressões básicas se assemelharem mais às de DWP: 23 (em ambas, algumas expressões que seriam naturalmente tratadas como compostas são tratadas como se fossem um único item), as regras sintáticas também podem ser escritas exatamente na mesma ordem, pois agora não há mais problemas com a recursão à esquerda.

```
% L0e_Syn2.ari
%      (Dowty, Wall & Peters 1981: 23)
% Syntax of L0e (Shift-reduce parser)
%      (Covington 1994: 159)

% Shift-reduce parser

parse(Sentence, Category) :-
      shift_reduce(Sentence, [], Category).

shift_reduce(Expression, Stack, Category) :-
      shift(Stack, Expression, NewStack, NewExpression),
      reduce(NewStack, ReducedStack),
      shift_reduce(NewExpression, ReducedStack, Category).
shift_reduce([], Category, Category).

shift(X, [H | Y], [H | X], Y).

reduce(Stack, ReducedStack) :-
      rule(Stack, NewStack),
      reduce(NewStack, ReducedStack).
reduce(Stack, Stack).

% Formation rules

rule([s, conj, s | X], [s | X]).     % s --> s, conj, s.
rule([s, neg | X], [s | X]).         % s --> neg, s.
rule([vp, n | X], [s | X]).          % s --> n, vp.
rule([vi | X], [vp | X]).            % vp --> vi.
rule([n, vt | X], [vp | X]).         % vp --> vt, n.
rule([Word | X], [Category | X]) :-
```

[63] Ainda uma segunda opção seria executar a conversão categorial durante o deslocamento; ou seja, ao invés de deslocar a expressão básica para a pilha, apenas eliminá-la da expressão de entrada, incluindo na pilha sua categoria correspondente. Assim, a cláusula que define o deslocamento passaria a ser escrita como “shift(Stack, [Word | NewExpression], [Category | Stack], NewExpression) :- word(Category, Word)”.

```
      word(Category, Word).

% Basic expressions

word(conj, and).
word(conj, or).
word(n, sadie).
word(n, liz).
word(n, hank).
word(vi, snores).
word(vi, sleeps).
word(vi, is_boring).
word(vt, loves).
word(vt, hates).
word(vt, is_taller_than).
word(neg, it_is_not_the_case_that).
```

Programa 9 - Analisador gramatical por deslocamento-e-redução em Prolog

Para ilustrar o funcionamento desse analisador gramatical, vamos acompanhar a dedução[64] da sentença "[liz, is_taller_than, sadie]", apresentada na Tabela 10, abaixo, de acordo com o Programa 9.

[64] A idéia de análise como cadeia de dedução sempre esteve presente nos outros programas; para maiores detalhes sobre isso, ver Pereira & Warren 1983 e Stabler 1993.

Para provar	Casa com	Unificando	Vai para
1. parse([liz, is_taller_than, sadie], [s]).	parse(Sentence, Category) :- shift_reduce(Sentence, [], Category).	Sentence = [liz, is_taller_than, sadie]	a. 2
		Category = [s]	
		yes	b. fim
2. shift_reduce([liz, is_taller_than, sadie], [], [s]).	shift_reduce(Expression, Stack, Category) :- shift(Stack, Expression, NewStack, NewExpression), reduce(NewStack, ReducedStack), shift_reduce(NewExpression, ReducedStack, Category).	Expression = [liz, is_taller_than, sadie]	a. 3
		Stack = []	
		Category = [s]	
		NewStack = [liz]	b. 4
		NewExpression = [is_taller_than, sadie]	c. 8
		ReducedStack = [n]	d. 1b
3. shift([], [liz, is_taller_than, sadie], NewStack, NewExpression).	shift(X, [H | Y], [H | X], Y).	X = []	2b
		H = liz	
		Y = [is_taller_than, sadie] = NewExpression	
4. reduce([liz], ReducedStack).	reduce(Stack, ReducedStack) :- rule(Stack, NewStack), reduce(NewStack, ReducedStack).	Stack = [liz]	a. 5
		NewStack = [n]	b. 7
		ReducedStack = [n]	c. 2c
5. rule([liz], NewStack).	rule([Word | X], [Category | X]) :- word(Category, Word).	Word = liz	a. 6
		X = []	b. 4b
		Category = n	
6. word(Category, liz).	word(n, liz).	n = Category	5b
7. reduce([n], ReducedStack).	reduce(Stack, Stack).	Stack = [n] = ReducedStack	4c
8. shift_reduce([is_taller_than, sadie], [n], [s]).	shift_reduce(Expression, Stack,Category) :- shift(Stack, Expression, NewStack, NewExpression), reduce(NewStack, ReducedStack), shift_reduce(NewExpression, ReducedStack, Category).	Expression = [is_taller_than, sadie]	a. 9
		Stack = [n]	
		Category = [s]	
		NewStack = [is_taller_than, n]	b. 10
		NewExpression = [sadie]	c. 14
		ReducedStack = [vt, n]	d. 2d

Para provar	Casa com	Unificando	Vai para
9. shift([n], [is_taller_than, sadie], NewStack, NewExpression).	shift(X, [H | Y], [H | X], Y).	X = [n]	8b
		H = is_taller_than	
		Y = [sadie] = NewSentence	
10. reduce([is_taller_than, n], ReducedStack).	reduce(Stack, ReducedStack) :- rule(Stack, NewStack), reduce(NewStack, ReducedStack).	Stack = [is_taller_than, n]	a. 11
		NewStack = [vt, n]	b. 13
		ReducedStack = [vt, n]	c. 8c
11. rule([is_taller_than, n], NewStack).	rule([Word | X], [Category | X]) :- word(Category, Word).	Word = is_taller_than	a. 12
		X = [n]	b. 10b
		Category = vt	
12. word(Category, is_taller_than).	word(vt, is_taller_than).	vt = Category	11b
13. reduce([vt, n], ReducedStack).	reduce(Stack,Stack).	Stack = [vt, n] = ReducedStack	10c
14. shift_reduce([sadie], [vt, n], [s]).	shift_reduce(Expression, Stack,Category) :- shift(Stack, Expression, NewStack, NewExpression), reduce(NewStack, ReducedStack), shift_reduce(NewExpression, ReducedStack, Category).	Expression = [sadie]	a. 15
		Stack = [vt, n]	
		Category = [s]	
		NewStack = [sadie, vt, n]	b. 16
		NewExpression = []	c. 24
		ReducedStack = [s]	d. 8d
15. shift([vt, n], [sadie], NewStack, NewExpression).	shift(X, [H | Y], [H | X], Y).	X = [vt, n]	14b
		H = [sadie]	
		Y = [] = NewSentence	
16. reduce([sadie, vt, n], ReducedStack).	reduce(Stack, ReducedStack) :- rule(Stack, NewStack), reduce(NewStack, ReducedStack).	Stack = [sadie, vt, n]	a. 17
		NewStack = [n, vt, n]	b. 19
		ReducedStack = [s]	c. 14c
17. rule([sadie, vt, n], NewStack).	rule([Word | X], [Category | X]) :- word(Category, Word).	Word = sadie	a. 18
		X = [vt, n]	
		Category = n	b. 16b
18. word(Category, sadie).	word(n, sadie).	n = Category	17b
19. reduce([n, vt, n], ReducedStack)	reduce(Stack, ReducedStack) :- rule(Stack, NewStack)	Stack = [n, vt, n]	a. 20

Para provar	Casa com	Unificando	Vai para
ReducedStack).	:- rule(Stack, NewStack), reduce(NewStack, ReducedStack).	NewStack = [vp, n]	b. 21
		ReducedStack = [s]	c. 16c
20. rule[n, vt, n], NewStack).	rule([n, vt | X], [vp | X]).	X = n	19b
21. reduce([vp, n], ReducedStack).	reduce(Stack, ReducedStack) :- rule(Stack, NewStack), reduce(NewStack, ReducedStack).	Stack = [vp, n]	a. 22
		NewStack = [s]	b. 23
		ReducedStack = [s]	c. 19c
22. rule([vp, n], NewStack).	rule([vp, n | X], [s | X]).	X = []	21b
23. reduce([s], ReducedStack).	reduce(Stack, Stack).	Stack = [s] = ReducedStack	21c
24. shift_reduce([], [s], [s]).	shift_reduce([], Category, Category).	Category = [s]	14d

Tabela 10 - Análise gramatical de "[liz, is_taller_than, sadie]" pelo Programa 9

A análise começa na linha 1, com "parse([liz, is_taller_than, sadie], [s]).", que casa com a primeira cláusula do Programa 9, cuja única função é garantir o mencionado pressuposto de que a análise começa com a pilha vazia. Assim, na linha 2, devido à unificação demonstrada na linha anterior, a nova cláusula a ser satisfeita é "shift_reduce([liz, is_taller_than, sadie], [], [s]).''; esta, por sua vez, casa com a primeira cláusula da definição de "shift-reduce/3", criando três novos objetivos a serem satisfeitos:

1) "shift([],[liz, is_taller_than, sadie], NewStack, NewExpression).", na linha 3 (que tem por função apenas tirar o primeiro elemento da expressão e colocá-lo na pilha, o que é feito pela ligação das variáveis: 'NewStack' é composto pelo primeiro elemento da segunda lista anteposto à primeira lista, enquanto 'NewExpression' é o que sobra da segunda lista sem esse primeiro elemento),

2) "reduce([liz], ReducedStack).", na linha 4, e

3) "shift_reduce([is_taller_than, sadie], [n], [s]).", na linha 8.

A aplicação do predicado "reduce/2" à pilha, agora instanciada com "[liz]", serve para substituir nessa mesma pilha a expressão básica ("Liz") pela sua respectiva categoria ("n"). Isso é feito primeiro através do casamento com a primeira cláusula da definição de "reduce/2", resultando no objetivo "reduce([liz], ReducedStack) :- rule([liz], NewStack),

reduce(NewStack, ReducedStack).". Com "rule([liz], NewStack).", que casa apenas com a última cláusula da definição de "rule/2" ("rule([Word | X], [Category | X]) :- word(Category, Word)."), faz-se a substituição da expressão pela categoria, enquanto o predicado "reduce(NewStack, ReducedStack)" (depois que a variável 'NewStack' foi instanciada com "[n]") garante que mais nenhuma redução ainda possa ser feita.

O terceiro desses objetivos ("shift_reduce([is_taller_than, sadie], [n], [s]).") introduz a recursividade da definição, recomeçando o ciclo de deslocamento da expressão para a pilha e de todas as reduções que esse deslocamento permitir. Assim, a expressão "is_taller_than" é deslocada para a pilha (pelo predicado "shift([n], [is_taller_than, sadie], NewStack, NewExpression)", na linha 9), e depois substituída pela sua categoria (devido ao predicado "reduce([is_taller_than, n], ReducedStack)", entre as linhas 10 e 13);[65] como a pilha então constituída ('[vt, n]') não permite mais reduções, o ciclo é reiniciado para o resto da expressão e para essa nova pilha (linha 14).

Com esse novo objetivo ("shift_reduce([sadie], [vt, n], [s])."), o analisador inicia o seu último ciclo de deslocamento, já que não restará mais nenhuma expressão básica a ser deslocada para a pilha depois disso. Assim, a expressão "sadie" é deslocada (linha 15) e substituída pela sua respectiva categoria (linhas 16 a 19); agora com a pilha "[n, vt, n]", tem início uma série de duas reduções: primeiro através de "rule([n, vt | X], [vp | X])." (regra de formação de predicados: "VP $\rightarrow$ V_t N"), que transforma a pilha em "[vp, n]" (na linha 22), e depois por "rule([vp, n | X], [s | X])." (regra de formação das sentenças: "S $\rightarrow$ N VP"), que simplifica a pilha para '[s]' (na linha 24).

Como o interpretador consegue chegar ao final das reduções com a pilha instanciada com o mesmo valor com que iniciamos a prova ("[s]"), a resposta que o interpretador dá para a pergunta "?- parse([liz, is_taller_than, sadie], [s])." é afirmativa.

---

[65] Como esse procedimento é exatamente igual ao que já foi explicado anteriormente, entre as linhas 4 e 7, não repetiremos aqui sua explicação.

4.2.2. Semântica de L_{0E}

Como foi mencionado anteriormente, nas notas 53 e 54, na p.66, a escolha de uma gramática de estrutura sintagmática ainda é completamente compatível com o princípio de composicionalidade; como antes, basta apenas que a cada expressão básica seja atribuído um valor semântico e para cada regra sintagmática corresponda uma regra semântica que determine o seu valor a partir dos valores de seus constituintes.

Ainda como em L_0, os nomes denotam indivíduos. A título de exemplificação, DWP: 26 sugerem os seguintes valores para os nomes de L_{0E}:

- $[\![\text{Sadie}]\!]$ = Anwar Sadat,
- $[\![\text{Liz}]\!]$ = Queen Elizabeth II e
- $[\![\text{Hank}]\!]$ = Henry Kissinger.

Para os valores semânticos das sentenças de L_{0E} (ou seja, para os valores de verdade), os autores introduzem uma pequena variante notacional: ao invés de classificar uma sentença como "verdadeira" ou "falsa", eles sugerem que esses valores sejam respectivamente representados pelos números "1" e "0". Dessa forma, é possível representar a atribuição do valor semântico para uma sentença *S* exatamente da mesma forma como foi feito para os nomes: $[\![S]\!] = 1$, se a sentença for verdadeira, e $[\![S]\!] = 0$, se a sentença for falsa.

Com essa mudança, podemos representar um conjunto de indivíduos como uma função de indivíduos para valores de verdade, da seguinte maneira: $f_S(x) = 1$ se $x_{x \in C} \in S$ e $f_S(x) = 0$ se $x_{x \in C} \notin S$. Essa função é chamada de *função característica* e divide um domínio *C* em dois subconjuntos: o dos indivíduos para os quais o valor da função é 1 (o conjunto *S*) e o dos indivíduos para os quais esse valor é 0 (o complemento de *S*).

Conseqüentemente, os valores semânticos dos V_i's (que correspondem aos predicados de um lugar em L_0, que denotavam conjuntos de indivíduos) podem agora ser concebidos como funções características: funções que, para cada indivíduo, determinam

um valor de verdade correspondente. Para [[snores]], DWP: 26 sugerem a seguinte função característica:

$$[[\text{snores}]] = \begin{bmatrix} \text{Anwar Sadat} & \rightarrow & 1 \\ \text{Queen Elizabeth II} & \rightarrow & 1 \\ \text{Henry Kissinger} & \rightarrow & 0 \end{bmatrix}$$

Como uma função é um conjunto de pares ordenados, o que se acabou de representar acima, na verdade, é o conjunto {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 1⟩, ⟨Henry Kissinger, 0⟩}.

Para os outros dois V_i's, DWP: 27 sugerem as seguintes funções características:

$$[[\text{sleeps}]] = \begin{bmatrix} \text{Anwar Sadat} & \rightarrow & 1 \\ \text{Queen Elizabeth II} & \rightarrow & 0 \\ \text{Henry Kissinger} & \rightarrow & 0 \end{bmatrix}$$

$$[[\text{is - boring}]] = \begin{bmatrix} \text{Anwar Sadat} & \rightarrow & 1 \\ \text{Queen Elizabeth II} & \rightarrow & 1 \\ \text{Henry Kissinger} & \rightarrow & 1 \end{bmatrix}$$

A partir dos valores semânticos dos nomes e dos verbos intransitivos, já é possível calcular o valor semântico das sentenças compostas pelas regras "S → N VP" e "VP → V_i". Para a sentença "Sadie snores", por exemplo, o resultado é obtido instanciando-se o argumento da função característica correspondente ao valor semântico de "snores" com o valor semântico de "Sadie"; segundo os valores sugeridos acima, essa sentença ocorre ser verdadeira: [[Sadie snores]] = [[snores]]([[Sadie]]) = 1, pois o valor semântico de "Sadie" ([[Sadie]] = Anwar Sadat) aplicado à função característica que é valor semântico de "snores" ([[snores]] = {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 1⟩, ⟨Henry Kissinger, 0⟩}) resulta em "1".

Já que os valores semânticos só podem ser atribuídos a expressões sintaticamente analisadas, conforme foi visto na seção anterior, as regras de atribuição precisam recorrer

às configurações sintáticas. Assim, para começar, precisaríamos de duas regras, uma para cada categoria lexical:

a. Se α é N e N → β, então $[\![\alpha]\!] = [\![\beta]\!]$, e

b. Se α é V_i e V_i → β, então $[\![\alpha]\!] = [\![\beta]\!]$.

Na verdade, precisaríamos de tantas regras desse tipo quantas fossem as categorias lexicais da língua; no entanto, introduzindo na regra os conceitos de categoria e item lexical, todas essas regras podem ser abreviadas como:[66]

1. Se α é γ e γ → β (onde γ representa uma categoria lexical e β um item lexical), então $[\![\alpha]\!] = [\![\beta]\!]$.

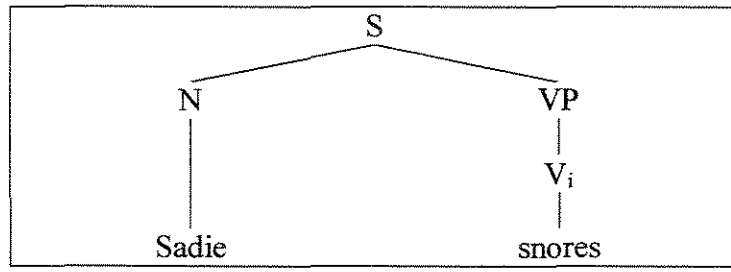
A regra semântica correspondente à regra sintática "VP → V_i" pode ser definida da seguinte maneira:

2. Se α é VP e VP → V_i, então $[\![\alpha]\!] = [\![V_i]\!]$.

Finalmente, para poder calcular o valor da sentença, precisamos ainda da regra semântica corresponde à regra sintática "S → N VP":

3. Se α é S e S → N VP, então $[\![\alpha]\!] = [\![VP]\!]([\![N]\!])$.

De volta à sentença "Sadie snores", pelas regras sintáticas apresentadas na seção anterior, conseguimos montar a árvore sintática abaixo:



[66] Mais precisamente, este é antes um esquema do que uma regra propriamente, e ele abrevia 12 regras em L_{0E}, correspondente a cada um dos seus itens lexicais.

Com as três regras semânticas mais acima, podemos também calcular o valor semântico de cada ponto da árvore, de forma que o valor semântico de toda a sentença é o valor semântico atribuído ao nó "S" no topo da árvore. Pela regra 1, pode-se calcular o valor tanto do nó "N" quanto do nó "V_i": $[\![N]\!]$ = [[Sadie]] = Anwar Sadat e $[\![V_i]\!]$ = [[snores]] = {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 1⟩, ⟨Henry Kissinger, 0⟩}; pela regra 2, calcula-se o valor do nó "VP": [[VP]] = $[\![V_i]\!]$ = {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 1⟩, ⟨Henry Kissinger, 0⟩}; e finalmente pela regra 3, é possível calcular o valor do nó "S": [[S]] = $[\![VP]\!]([\![N]\!])$ = {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 1⟩, ⟨Henry Kissinger, 0⟩}(Anwar Sadat) = 1.

Na seqüência, para calcular o valor semântico das sentenças construídas com V_t's, é preciso definir os seus respectivos valores semânticos. Segundo a gramática de L_{0E}, um V_t se combina com um N resultando em um VP; dessa forma, era de se esperar que o valor semântico correspondente a um V_t fosse uma função de valores semânticos de N (indivíduos) para valores semânticos de VP (uma função característica). Assim, para "loves", por exemplo, DWP: 31 sugerem a seguinte função, que mapeia indivíduos em funções características:[67]

$$[\![\text{loves}]\!] = \left[\begin{array}{lll} \text{Anwar Sadat} & \rightarrow & \left[\begin{array}{ll} \text{Anwar Sadat} & \rightarrow 1 \\ \text{Queen Elizabeth II} & \rightarrow 1 \\ \text{Henry Kissinger} & \rightarrow 1 \end{array}\right] \\ \text{Queen Elizabeth II} & \rightarrow & \left[\begin{array}{ll} \text{Anwar Sadat} & \rightarrow 0 \\ \text{Queen Elizabeth II} & \rightarrow 1 \\ \text{Henry Kissinger} & \rightarrow 0 \end{array}\right] \\ \text{Henry Kissinger} & \rightarrow & \left[\begin{array}{ll} \text{Anwar Sadat} & \rightarrow 1 \\ \text{Queen Elizabeth II} & \rightarrow 0 \\ \text{Henry Kissinger} & \rightarrow 1 \end{array}\right] \end{array}\right]$$

Para serem exaustivos, DWP: 32-33 sugerem ainda os valores dos outros V_t's:

---

[67] Na verdade, para "loves", os autores empregam uma notação mais extensiva; mas como para os outros V_t's eles preferem essa notação mais simplificada, apresentamos aqui apenas essa versão mais simples. Para mais detalhes, ver DWP: 31.

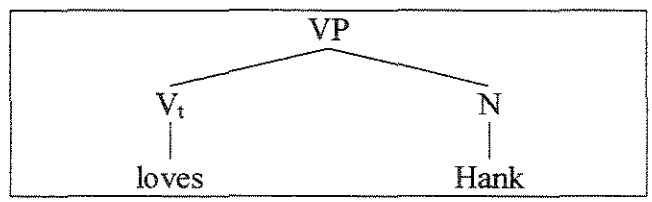
$$[\![\text{hates}]\!] = \begin{bmatrix} \text{Anwar Sadat} \rightarrow \begin{bmatrix} \text{Anwar Sadat} \rightarrow 1 \\ \text{Queen Elizabeth II} \rightarrow 1 \\ \text{Henry Kissinger} \rightarrow 1 \end{bmatrix} \\ \text{Queen Elizabeth II} \rightarrow \begin{bmatrix} \text{Anwar Sadat} \rightarrow 0 \\ \text{Queen Elizabeth II} \rightarrow 1 \\ \text{Henry Kissinger} \rightarrow 0 \end{bmatrix} \\ \text{Henry Kissinger} \rightarrow \begin{bmatrix} \text{Anwar Sadat} \rightarrow 1 \\ \text{Queen Elizabeth II} \rightarrow 0 \\ \text{Henry Kissinger} \rightarrow 1 \end{bmatrix} \end{bmatrix}$$

$$[\![\text{is - taller - than}]\!] = \begin{bmatrix} \text{Anwar Sadat} \rightarrow \begin{bmatrix} \text{Anwar Sadat} \rightarrow 1 \\ \text{Queen Elizabeth II} \rightarrow 1 \\ \text{Henry Kissinger} \rightarrow 1 \end{bmatrix} \\ \text{Queen Elizabeth II} \rightarrow \begin{bmatrix} \text{Anwar Sadat} \rightarrow 0 \\ \text{Queen Elizabeth II} \rightarrow 1 \\ \text{Henry Kissinger} \rightarrow 0 \end{bmatrix} \\ \text{Henry Kissinger} \rightarrow \begin{bmatrix} \text{Anwar Sadat} \rightarrow 1 \\ \text{Queen Elizabeth II} \rightarrow 0 \\ \text{Henry Kissinger} \rightarrow 1 \end{bmatrix} \end{bmatrix}$$

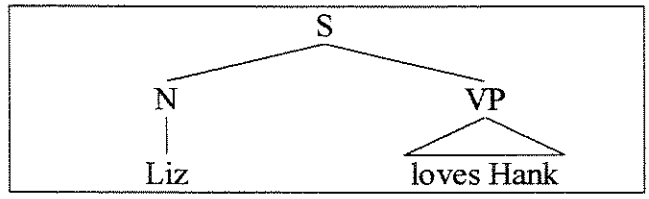
Agora, para calcular o valor semântico de um VP formado a partir de um V_t, a seguinte regra semântica ainda é necessária:

4. Se α é um VP e VP $\rightarrow V_t$ N, então $[\![\alpha]\!] = [\![V_t]\!]([\![N]\!])$.

Para ilustrar, a gramática de L_{0E} associa à expressão "loves Hank" a seguinte árvore:



De forma agora elementar, o valor dos nós "N" e "V_t" são determinado pela mesma regra 1, correspondendo respectivamente a [[Hank]] e [[loves]]. E a partir da regra 4, então, pode-se saber que [[*VP*]] = [[loves]]([[Hank]]) = {⟨Anwar Sadat, {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 1⟩, ⟨Henry Kissinger, 1⟩}, ⟨Queen Elizabeth II, {⟨Anwar Sadat, 0⟩, ⟨Queen Elizabeth II, 1⟩, ⟨Henry Kissinger, 0⟩}⟩, ⟨Henry Kissinger, {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 0⟩, ⟨Henry Kissinger, 1⟩}⟩}(Henry Kissinger) = {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 0⟩, ⟨Henry Kissinger, 1⟩}. Assim como estão determinadas as regras de L_{0E}, nada mais é necessário para se calcular o valor semântico de sentenças compostas com V_t's; o valor de "Liz loves Hank", por exemplo, cuja análise sintática é representada pela seguinte árvore:[68]



também pode ser determinado pela regra 3, de tal forma que [[S]] = [[VP]]([[N]]) = {⟨Anwar Sadat, 1⟩, ⟨Queen Elizabeth II, 0⟩, ⟨Henry Kissinger, 1⟩}(Queen Elizabeth II) = 0. Ou seja, a sentença "Liz loves Hank" é falsa em L_{0E} para as atribuições sugeridas.

O valor dos conectivos, finalmente, também pode ser representado por funções características que apresentam o mesmo efeito das tabelas de verdade apresentadas na Tabela 5 da seção 1.1.1, na p. 58. O valor semântico da negação, que é uma expressão que constitui um S a partir de outro, é uma função de valores de verdade a valores de verdade, de tal forma que:

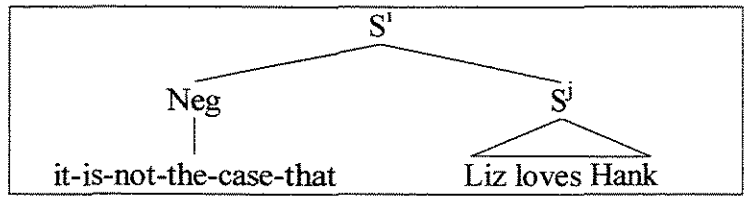
$$[\![\text{it - is - not - the - case - that}]\!] = \begin{bmatrix} 1 \rightarrow 0 \\ 0 \rightarrow 1 \end{bmatrix}$$

[68] Como de costume, o triângulo representa uma abreviação estrutural. Assim, na árvore acima, a estrutura interna de "loves Hank" é omitida, já que ela foi analisada um pouco antes.

A regra semântica correspondente é:[69]

5. Se α é S^i e $S^i \rightarrow$ Neg S^j, então $[\![\alpha]\!] = [\![\text{Neg}]\!]([\![S^j]\!])$.

Assim, o valor de uma sentença como "it-is-not-the-case-that Liz loves Hank", que é composta pela expressão básica "it-is-not-the-case-that" e pela sentença "Liz loves Hank" (cujo valor acabamos de calcular), pode ser deduzido da seguinte maneira. A análise sintática dessa sentença é representada pela árvore abaixo:



assim, o valor do S^i é determinado pela regra 5 e corresponde a: $[\![\text{it - is - not - the - case - that}]\!]([\![\text{Liz loves Hank}]\!]) = \{\langle 1, 0\rangle, \langle 0, 1\rangle\}(0) = 1$. Ou seja, "it-is-not-the-case-that Liz loves Hank" é verdadeira em L_{0E} para as atribuições supostas.

Aos conectivos binários "and" e "or" correspondem as seguintes funções características:

$$[\![\text{and}]\!] = \begin{bmatrix} \langle 1, 1\rangle \rightarrow 1 \\ \langle 1, 0\rangle \rightarrow 0 \\ \langle 0, 1\rangle \rightarrow 0 \\ \langle 0, 0\rangle \rightarrow 0 \end{bmatrix}$$

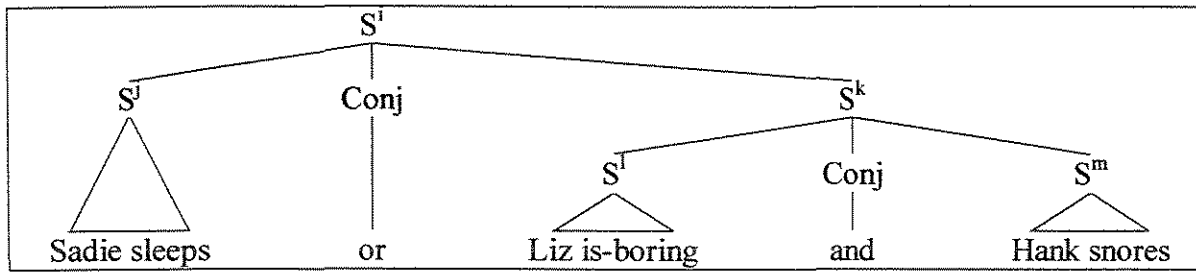
$$[\![\text{or}]\!] = \begin{bmatrix} \langle 1, 1\rangle \rightarrow 1 \\ \langle 1, 0\rangle \rightarrow 1 \\ \langle 0, 1\rangle \rightarrow 1 \\ \langle 0, 0\rangle \rightarrow 0 \end{bmatrix}$$

[69] A função dos expoentes aqui é apenas a de distinguidor. O objetivo é evitar a confusão entre "S" à direita e à esquerda da seta; assim, poderíamos também ter escolhido [S, S] para designar o "S" depois da seta e apenas S para o de antes dela (assim como se usa [NP, S] e [NP, VP] na tradição gerativista para distinguir sujeito e objeto direto, respectivamente). Os expoentes foram adotados antes por comodidade tipográfica; além disso, a outra notação podia também ser confundida com a notação de lista do Prolog.

Sintaticamente, como essas conjunções são expressões que combinadas com duas sentenças formam uma terceira sentença, o seu valor semântico é uma função que mapeia dois valores de verdade (mais especificamente, um par ordenado de valores de verdade)[70] em um terceiro valor de verdade. E para que as sentenças com essas expressões possam ser semanticamente calculadas, ainda é preciso uma última regra:

6. Se α é S^i e $S^i \rightarrow S^j$ Conj S^k, então $[\![\alpha]\!] = [\![Conj]\!](\langle [\![S^j]\!], [\![S^k]\!]\rangle)$.

Para exemplificar o funcionamento dessa regra 6, vamos usar a sentença "Sadie sleeps or Liz is-boring and Hank snores" que, como vimos anteriormente (Quadro 5 e Quadro 6, nas pp. 67-67), apresentava uma ambigüidade estrutural. Na análise com a árvore ramificada à direita, a representação sintática equivale a:



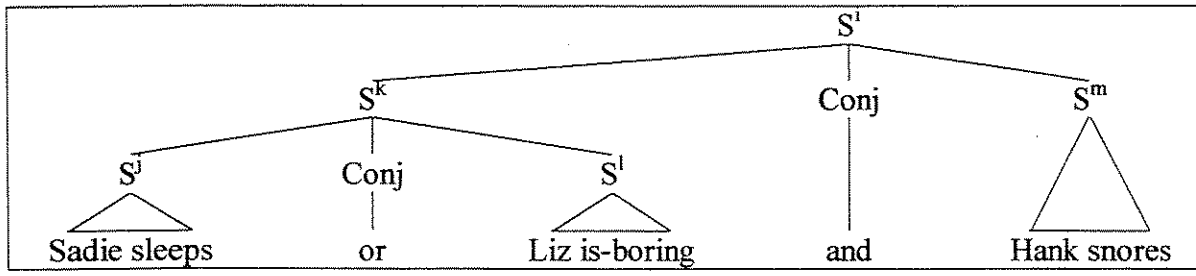
Com esta análise sintática, $[\![S^k]\!] = [\![and]\!](\langle [\![S^l]\!], [\![S^m]\!]\rangle)$ e $[\![S^i]\!] = [\![or]\!](\langle [\![S^j]\!], [\![S^k]\!]\rangle)$. Como $[\![S^l]\!] = 1$ e $[\![S^m]\!] = 0$,[71] então $[\![S^k]\!] = \{\langle\langle 1, 1\rangle, 1\rangle, \langle\langle 1, 0\rangle, 0\rangle, \langle\langle 0, 1\rangle, 0\rangle, \langle\langle 0, 0\rangle, 0\rangle\}(\langle 1, 0\rangle) = 0$. E ainda como $[\![S^j]\!] = 1$, então $[\![S^k]\!] = \{\langle\langle 1, 1\rangle, 1\rangle, \langle\langle 1, 0\rangle, 1\rangle, \langle\langle 0, 1\rangle, 1\rangle, \langle\langle 0, 0\rangle, 0\rangle\}(\langle 1, 0\rangle) = 1$. Portanto, para a análise sintática ramificada à direita, a sentença "Sadie sleeps or Liz is-boring and Hank snores" é verdadeira em L_{0E} para as atribuições sugeridas.

---

[70] Para "and" e "or" a necessidade de que seja um par ordenado não é evidente, já que a ordem não interfere em nenhum desses dois conectivos; no entanto, como a ordem é fundamental para a definição de "$\rightarrow$", e do que corresponderia a "se" em L_{0E} (se os autores tivessem sido mais exaustivos), isso pode ter justificado a opção pelo par ordenado. Essa opção ainda evita uma decisão sobre a ordem de aplicação dos conectivos: se os seus valores fossem definidos como funções de um valor de verdade para funções de outro valor de verdade (segundo o esquema: $[T_1 \rightarrow [\ T_2 \rightarrow T_3]]$), precisaríamos definir, para cada conectivo, se ele se associa à direita ou à esquerda (ou, esquematicamente, se "S (Conj S)" ou se "(S Conj) S").

[71] Deixamos os cálculos de $[\![S^l]\!]$ e de $[\![S^m]\!]$ como exercício paro o leitor.

Já quando usamos a segunda análise sintática, com a ramificação à esquerda, representada pela seguinte árvore:



Tem-se: $[\![S^i]\!] = [\![and]\!](\langle [\![S^k]\!], [\![S^m]\!] \rangle)$ e $[\![S^k]\!] = [\![or]\!](\langle [\![S^j]\!], [\![S^l]\!] \rangle)$. Assim, com os mesmos valores $[\![S^j]\!] = 1$, $[\![S^l]\!] = 1$ e $[\![S^m]\!] = 0$, descobre-se que $[\![S^k]\!] = \{\langle\langle 1, 1\rangle, 1\rangle, \langle\langle 1, 0\rangle, 1\rangle, \langle\langle 0, 1\rangle, 1\rangle, \langle\langle 0, 0\rangle, 0\rangle\}(\langle 1, 1\rangle) = 1$, e depois que $[\![S^i]\!] = \{\langle\langle 1, 1\rangle, 1\rangle, \langle\langle 1, 0\rangle, 0\rangle, \langle\langle 0, 1\rangle, 0\rangle, \langle\langle 0, 0\rangle, 0\rangle\}(\langle 1, 0\rangle) = 0$. Ou seja, para a análise sintática ramificada à esquerda, a sentença "Sadie sleeps or Liz is-boring and Hank snores" é falsa em L_{0E} para as atribuições supostas.

Com o funcionamento da semântica de L_{0E} completamente explicado, podemos passar agora ao funcionamento do programa que executa em Prolog essas especificações. Antes de comentar o programa de análise semântica em si, é preciso estabelecer as atribuições dos valores semânticos para as expressões básicas de L_{0E}. Seguindo as sugestões oferecidas pelos autores, podemos manter exatamente o mesmo esquema que foi usado para a semântica de L_0, através do predicado "semantic_value/2", onde o primeiro argumento corresponde às expressões e o segundo aos seus respectivos valores semânticos. Essas atribuições são reproduzidas pelo Programa 10, abaixo.

```
% L0e_Att.ari
%     (Dowty, Wall & Peters 1981: 26-33)
% Semantic values for basic expressions of L0e

% n

semantic_value(sadie, anwar_sadat).
semantic_value(liz, queen_elizabeth_ii).
semantic_value(hank, henry_kissinger).

% vi

semantic_value(snores, [anwar_sadat, 1]).
```

```
semantic_value(snores, [queen_elizabeth_ii, 1]).
semantic_value(snores, [henry_kissinger, 0]).
semantic_value(sleeps, [anwar_sadat, 1]).
semantic_value(sleeps, [queen_elizabeth_ii, 0]).
semantic_value(sleeps, [henry_kissinger, 0]).
semantic_value(is_boring, [anwar_sadat, 1]).
semantic_value(is_boring, [queen_elizabeth_ii, 1]).
semantic_value(is_boring, [henry_kissinger, 1]).

% vt

semantic_value(loves, [anwar_sadat, anwar_sadat, 1]).
semantic_value(loves, [anwar_sadat, queen_elizabeth_ii, 1]).
semantic_value(loves, [anwar_sadat, henry_kissinger, 1]).
semantic_value(loves, [queen_elizabeth_ii, anwar_sadat, 0]).
semantic_value(loves, [queen_elizabeth_ii,
      queen_elizabeth_ii, 1]).
semantic_value(loves, [queen_elizabeth_ii, henry_kissinger,
      0]).
semantic_value(loves, [henry_kissinger, anwar_sadat, 1]).
semantic_value(loves, [henry_kissinger, queen_elizabeth_ii,
      0]).
semantic_value(loves, [henry_kissinger, henry_kissinger,
      1]).
semantic_value(hates, [anwar_sadat, anwar_sadat, 0]).
semantic_value(hates, [anwar_sadat, queen_elizabeth_ii, 0]).
semantic_value(hates, [anwar_sadat, henry_kissinger, 0]).
semantic_value(hates, [queen_elizabeth_ii, anwar_sadat, 1]).
semantic_value(hates, [queen_elizabeth_ii,
      queen_elizabeth_ii, 0]).
semantic_value(hates, [queen_elizabeth_ii, henry_kissinger,
      1]).
semantic_value(hates, [henry_kissinger, anwar_sadat, 0]).
semantic_value(hates, [henry_kissinger, queen_elizabeth_ii,
      1]).
semantic_value(hates, [henry_kissinger, henry_kissinger,
      0]).
semantic_value(is_taller_than, [anwar_sadat, anwar_sadat,
      0]).
semantic_value(is_taller_than, [anwar_sadat,
      queen_elizabeth_ii, 1]).
semantic_value(is_taller_than, [anwar_sadat,
      henry_kissinger, 0]).
semantic_value(is_taller_than, [queen_elizabeth_ii,
      anwar_sadat, 0]).
semantic_value(is_taller_than, [queen_elizabeth_ii,
      queen_elizabeth_ii, 0]).
semantic_value(is_taller_than, [queen_elizabeth_ii,
      henry_kissinger, 0]).
semantic_value(is_taller_than, [henry_kissinger,
      anwar_sadat, 1]).
semantic_value(is_taller_than, [henry_kissinger,
      queen_elizabeth_ii, 1]).
semantic_value(is_taller_than, [henry_kissinger,
      henry_kissinger, 0]).

% neg
```

```
semantic_value(it_is_not_the_case_that, [0, 1]).
semantic_value(it_is_not_the_case_that, [1, 0]).

% conj

semantic_value(and, [1, 1, 1]).
semantic_value(and, [1, 0, 0]).
semantic_value(and, [0, 1, 0]).
semantic_value(and, [0, 0, 0]).
semantic_value(or, [1, 1, 1]).
semantic_value(or, [1, 0, 1]).
semantic_value(or, [0, 1, 1]).
semantic_value(or, [0, 0, 0]).
```

Programa 10 - Atribuições básicas supostas para L_{0E}

O fato do nome "sadie" corresponder ao valor semântico "anwar_sadat", por exemplo, é expresso como "semantic_value(sadie,anwar_sadat)."; a principal diferença para L_0, agora, é que os valores semânticos dos nomes são átomos e não listas (ou seja, o que era representado antes como "[anwar, sadat]" aqui é representado como "anwar_sadat").

Para os V_i's, os seus valores semânticos podem ser expressos por uma lista cujo primeiro elemento é um indivíduo enquanto o segundo elemento é o seu respectivo valor de verdade para aquele indivíduo; assim, para representarmos a função característica correspondente a "snores", por exemplo, é preciso listar o seu valor semântico para cada indivíduo:

1. semantic_value(snores, [anwar_sadat, 1]).
2. semantic_value(snores, [queen_elizabeth_ii, 1]).
3. semantic_value(snores, [henry_kissinger, 0]).

A diferença em relação à semântica de L_0 é que não se declara apenas os indivíduos para os quais o predicado seja verdadeiro, mas sim seu valor para cada indivíduo do domínio; dessa forma, podemos construir uma representação semântica para cada expressão bem formada de L_{0E} e não apenas constatar o valor de verdade das sentenças, como em L_0; ou seja, onde antes tínhamos apenas um autômato para confirmar se uma determinada estrutura era bem formada, agora temos um transdutor que transforma uma estrutura seqüencial (uma lista correspondente a uma expressão de L_{0E}) em outra hierarquizada (os valores semânticos dessas expressões para as atribuições sugeridas).

Como os V_t's são funções de indivíduos para funções características, a lista dos valores semânticos dessas expressões precisa conter sempre três elementos: os dois últimos determinam a função característica para o indivíduo que encabeça essa lista. Assim, para "loves", por exemplo, os valores pertinentes são:

1. semantic_value(loves, [anwar_sadat, anwar_sadat, 1]).
2. semantic_value(loves, [anwar_sadat, queen_elizabeth_ii, 1]).
3. semantic_value(loves, [anwar_sadat, henry_kissinger, 1]).

...

Ou seja, segundo essas atribuições, todo mundo ama Anwar Sadat. Conforme essa disposição do programa, a função característica correspondente ao VP "loves X" vai ser obtida removendo-se o primeiro elemento da lista de valores semânticos, quando o primeiro argumento do predicado "semantic_value/2" for "loves" e o mencionado elemento for o valor semântico de 'X'.

Para os conectivos, finalmente, seus valores semânticos serão listas apenas de valores de verdade. Como a negação é um operador unário que inverte o valor da sentença ao qual ele se prende, o seu valor semântico pode ser expresso em Prolog por dois fatos:

1. semantic_value(it_is_not_the_case_that, [0, 1]).
2. semantic_value(it_is_not_the_case_that, [1, 0]).

Dessa maneira, para os conectivos, o resultado de sua aplicação sempre estará expresso no último elemento do segundo argumento do predicado "semantic_value"; em sua totalidade, os fatos que determinam os valores semânticos dos conectivos reproduzem exatamente a sua respectiva tabela de verdade. Como os outros dois conectivos são operadores binários, os valores de entrada ocupam as duas primeiras posições de cada lista; assim, os fatos pertinentes para "and", por exemplo, são:

1. semantic_value(and, [1, 1, 1]).
2. semantic_value(and, [1, 0, 0]).
3. semantic_value(and, [0, 1, 0]).

4. semantic_value(and, [0, 0, 0]).

A definição do valor semântico de "or" pode ser feita exatamente da mesma maneira, apenas com os valores adequados na última posição da lista.

Com os valores semânticos atribuídos às expressões básicas de L_{0E}, podemos facilmente adaptar o analisador gramatical por deslocamento-e-redução desenvolvido no Programa 9, na p. 73 da seção anterior, de forma que ele não apenas informe a categoria da expressão analisada, mas também apresente o seu respectivo valor semântico. Isso é feito pelo Programa 11, abaixo.[72]

```
% L0e_Sem2.ari
%      (Dowty, Wall & Peters 1981: 23)
% Semantics of L0e (Shift-reduce parser)
%      (Covington 1994: 159 - modified)

% Load basic attributions

:- reconsult('L0e_Att.ari').

% Drive predicate

parse(Expr) :-
      parse(Expr, [], [SynCat, SemVal]), nl,
      write('Expression: '), write(Expr), nl,
      write('Syntactic Category: '), write(SynCat), nl,
      write('Semantic Value: '), write(SemVal), nl, nl.

% Shift-reduce parser

parse(Expr, Stack, Result) :-
      shift(Expr, NewExpr, Stack, NewStack),
      reduce(NewStack, ReducedStack),
      parse(NewExpr, ReducedStack, Result).
parse([], [Result], Result).

shift([Word|NewExp], NewExp, Stack, [Word | Stack]).

reduce(Stack, ReducedStack) :-
      rule(Stack, NewStack),
      reduce(NewStack, ReducedStack).
reduce(Stack, Stack).

% Formation rules
```

[72] Apesar do analisador sintático por DCG não funcionar direito, conforme discussão apresentada na p. 68, a título de exercício, o interpretador semântico correspondente foi implementado e pode ser visto na seção 8.1 do apêndice.

```
rule([[s, T1], [conj, [T1, T2, T]], [s, T2] | Stack], [[s,
      T] | Stack]).
rule([[s, T1], [neg, [T1, T]] | Stack], [[s, T] | Stack]).
rule([[vp, [E, T]], [n, E] | Stack], [[s, T] | Stack]).
rule([[vi, SemVal] | Stack], [[vp, SemVal] | Stack]).
rule([[n, E], [vt, [E | SemVal]] | Stack], [[vp, SemVal] |
      Stack]).
rule([Word | Stack], [[SynCat, SemVal] | Stack]) :-
      word(Word, SynCat), semantic_value(Word, SemVal).

% Basic expressions

word(and, conj).
word(or, conj).
word(sadie, n).
word(liz, n).
word(hank, n).
word(snores, vi).
word(sleeps, vi).
word(is_boring, vi).
word(loves, vt).
word(hates, vt).
word(is_taller_than, vt).
word(it_is_not_the_case_that, neg).
```

Programa 11 - Semântica de L_{0E} por deslocamento-e-redução

A única distinção entre o Programa 11 e sua antiga versão exclusivamente sintática (Programa 9) está no formato da pilha: se antes ela continha apenas uma categoria sintática, agora ela contém também o seu respectivo valor semântico. Assim, esta nova pilha é composta por um par ordenado constituído pela categoria sintática ('SynCat') e pelo valor semântico ('SemVal') da expressão básica, que no programa é representado por uma lista de dois elementos: "[SynCat, SemVal]".

Por causa dessa pequena mudança, as principais revisões se concentram na definição das regras da gramática (predicado "rule/2"). Antes, esses predicados operavam apenas sobre uma lista de categorias, resultado em outra lista simplificada em relação à primeira, no seguinte esquema: "rule([Cat_1, ..., Cat_i | Rest], [Cat_j | Rest])"; como a nova pilha é uma lista de pares ordenados, o predicado "rule/2" ainda simplifica uma lista, mas agora segundo o seguinte esquema: "rule([[$SynCat_1$, $SemVal_1$], ..., [$SynCat_i$, $SemVal_i$] | Rest], [[$SynCat_j$, $SemVal_j$] | Rest])"; apenas o último fato dessa definição, responsável por trocar na pilha uma palavra pelo seu respectivo par ordenado de categoria e valor, é um pouco diferente desse esquema: "rule([Word | Stack], [[SynCat, SemVal] | Stack])",

segundo as unificações apropriadas ("word(Word, SynCat)" e semantic_value(Word, SemVal)").

Os predicados "shift/4" e "reduce/2", por outro lado, não precisam de nenhuma modificação; e na definição de "parse/3", a única mudança foi na sua última cláusula, apenas para eliminar uma parentização desnecessária, introduzida pela forma como a lista de pares ordenados de categoria sintática e valor semântico é operada. Apenas por comodidade e para facilitar a operação do analisador, foi incluído um predicado ("parse/1") para inicializar o analisador com as condições apropriadas e para informar separadamente os resultados; assim, por exemplo, à solicitação "?- parse([sadie, sleeps, or, liz, is_boring, and, hank, snores]).", esse analisador pode apresentar duas soluções:

1. Expression: [sadie, sleeps, or, liz, is_boring, and, hank, snores]
   Syntactic Category: s
   Semantic Value: 0

2. Expression: [sadie, sleeps, or, liz, is_boring, and, hank, snores]
   Syntactic Category: s
   Semantic Value: 1

Como essa versão semântica é muito semelhante à versão sintática, não apresentaremos aqui a mesma tabela simulando o processamento de um exemplo, como fizemos anteriormente; até porque nenhum procedimento novo foi inserido, de forma que eles executam exatamente os mesmos passos. Todo o cálculo semântico ocorre pela unificação das variáveis no segundo elemento do par ordenado de categoria e valor, da seguinte forma:

1. a última regra, correspondente à inserção lexical ("rule([Word | Stack], [[SynCat, SemVal] | Stack]) :- word(Word, SynCat), semantic_value(Word, SemVal)."), detecta a ocorrência de uma palavra na pilha ('Word') e a substitui pelo par ordenado composto pela sua categoria sintática ('SynCat') e pelo seu valor semântico ('SemVal');

2. as duas regras anteriores a esta são responsáveis pelo cálculo do sintagma verbal: a correspondente ao verbo intransitivo apenas transfere o valor semântico do verbo para o sintagma e muda a categoria sintática ("rule([[vi, SemVal] | Stack], [[vp, SemVal] | Stack])."), enquanto a do verbo transitivo executa a aplicação funcional do seu valor semântico ao valor semântico do seu objeto ("rule([[n, E], [vt, [E | SemVal]] | Stack], [[vp, SemVal] | Stack])."");
3. a regra de formação das sentenças atômicas ("rule([[vp, [E, T]], [n, E] | Stack], [[s, T] | Stack])."") faz apenas a aplicação funcional do valor semântico do sintagma verbal ao valor semântico do sujeito;
4. as duas primeiras regras, finalmente, fazem a aplicação funcional do valor semântico dos conectivos aos seus respectivos argumentos: a primeira, dos conectivos binários, faz a aplicação a dois argumentos ("rule([[s, T1], [conj, [T1, T2, T]], [s, T2] | Stack], [[s, T] | Stack])."), enquanto a segunda, da negação, faz a aplicação a um único argumento ("rule([[s, T1], [neg, [T1, T]] | Stack], [[s, T] | Stack])."").

4.3. Língua L_1

4.3.1. Sintaxe de L_1

A língua L_1 é apenas uma extensão de L_0, no sentido de que esta última está propriamente contida na primeira; ou seja, nem toda sentença de L_1 é sentença de L_0, mas todas as sentenças de L_0 fazem parte de L_1 e serão interpretadas nessa última da mesma maneira como eram interpretadas na primeira.

A diferença entre ambas é que em L_1 foram introduzidas variáveis individuais e quantificadores para essas variáveis. Ainda que, do ponto de vista sintático, essa inserção

pareça quase insignificante, ela representa uma mudança bastante drástica para a sua interpretação semântica. Mas vejamos primeiro a sua sintaxe.

A língua L_1 contém exatamente as mesmas expressões básicas da língua L_0, às quais se acrescenta uma quantidade enumerável, mas infinita, de variáveis individuais. Essas variáveis serão representadas por v_i, para qualquer número natural i (para evitar uma indexação excessiva, DWP: 56 sugerem abreviar v_1 como x, v_2 como y e v_3 como z, e a atenção é concentrada exclusivamente nessas três variáveis). Isso está representado na Tabela 11.

Categorias sintáticas	Expressões Básicas
Nomes	d, n, j, m
Variáveis individuais	v_1 (x), v_2 (y), v_3 (z), v_4...
Predicados de um lugar	M, B
Predicados de dois lugares	K, L

Tabela 11 - Expressões básicas de L_1 e suas respectivas categorias

Na parte sintática, além das mesmas regras de formação de L_0 já apresentadas Quadro 1, na p. 49, L_1 apresenta ainda mais duas regras para a formação das fórmulas, onde figuram as variáveis e os quantificadores são inseridos sincategorematicamente (regras 8 e 9, no Quadro 7, abaixo).

1. Se δ é um predicado de um lugar e α é um termo, então $\delta(\alpha)$ é uma fórmula.
2. Se γ é um predicado de dois lugares e α e β são termos, então $\gamma(\alpha, \beta)$ é uma fórmula.
3. Se ϕ é uma fórmula, então $\neg\phi$ é uma fórmula.
4. Se ϕ e ψ são fórmulas, então $[\phi \wedge \psi]$ é uma fórmula.
5. Se ϕ e ψ são fórmulas, então $[\phi \vee \psi]$ é uma fórmula.
6. Se ϕ e ψ são fórmulas, então $[\phi \rightarrow \psi]$ é uma fórmula.
7. Se ϕ e ψ são fórmulas, então $[\phi \leftrightarrow \psi]$ é uma fórmula.
8. Se ϕ é uma fórmula e u é uma variável, então $\forall u\phi$ é uma fórmula.
9. Se ϕ é uma fórmula e u é uma variável, então $\exists u\phi$ é uma fórmula.

Quadro 7 - Regras de formação para L_1

Nessa nova formulação, as regras sintáticas apresentam ainda mais duas pequenas variações: as regras 1 e 2 falam de termos e não mais de nomes, e todas elas referem-se a fórmulas ao invés de sentenças.

A mudança de "nomes" para "termos" se justifica porque as variáveis pertencem à mesma categoria sintática dos nomes. Dessa forma, o conjunto dos termos de uma língua é a união dos seus conjuntos de nomes e de variáveis, e é a esse conjunto maior que as regras 1 e 2 fazem menção. Já a modificação terminológica de "sentença" para "fórmula" não é explicitamente justificada pelos autores.

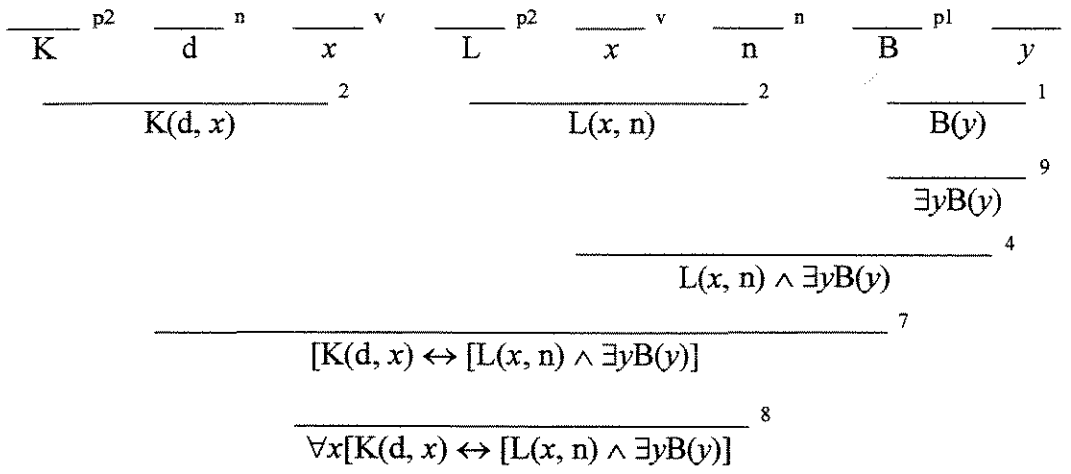
Com a inclusão das variáveis e da possibilidade de sua quantificação, surge uma distinção entre variáveis livres e ligadas, que pode ser definida da seguinte maneira: "a ocorrência de uma variável u em uma fórmula ϕ é *ligada em* ϕ se ela ocorre em ϕ dentro de uma sub-fórmula como $\forall u\psi$ ou $\exists u\psi$; caso contrário, essa ocorrência é *livre* em ϕ" (DWP: 57). Por essa definição, uma variável u é ligada sempre que aparecer nos contextos $\forall u$[... u...] ou $\exists u$[... u...], e livre quando ela não estiver quantificada ([... u...]) ou quando estiver apenas no escopo da quantificação de outra variável ($\forall z$[... u...] ou $\exists z$[... u...]).[73]

Através dessas regras de formação, a sintaxe de L_1 permite o que se chama de quantificação vazia, que é a quantificação de uma variável que não está no escopo do quantificador, como em "$\forall x$K(j, m)" ou "$\exists y$B(x)", por exemplo. Ainda que lingüisticamente essa solução não pareça muito natural, excluir a formação desse tipo de sentença complicaria excessivamente a sintaxe; e como as fórmulas com essa quantificação vazia serão interpretadas exatamente como as respectivas fórmulas sem ela (nos nossos exemplos: "K(j, m)" e "B(x)"), os autores preferem ignorar essa questão.

Assim, como já foi dito, todas as antigas sentenças de L_0 também são sentenças de L_1 e podem ser analisadas com os mesmos recursos usados para analisar as sentenças de L_0. E, a título de ilustração, podemos ver que a fórmula "$\forall x$[K(d, x) $\leftrightarrow$ [L(x, n) $\wedge$ $\exists y$B(y)]] pertence a L_1, usando o mesmo método de Prawitz exemplificado para L_0.

---

[73] O termo "fórmula" serve para indicar qualquer expressão construída através das regras do Quadro 7, quando não se exige a ligação das variáveis quantificadas; quando uma fórmula não apresenta variáveis livres, ela é chamada de "sentença".



Quadro 8 - Prova de $\forall x[K(d,x) \leftrightarrow [L(x,n) \wedge \exists y B(y)]$ ao estilo de Prawitz

Para representar as categorias básicas de L_1 em Prolog, com apenas *x*, *y* e *z* como variáveis, basta acrescentar ao programa de categorias básicas de L_0 (Programa 5, na p. 60) a listagem de variáveis, como no Programa 12, abaixo.

```
% L1_Cat1.ari
%     (Dowty, Wall & Peters 1981: 56)
% Basic categories for L1 defined as expression predicate

name(d).
name(n).
name(j).
name(m).

variable(x).
variable(y).
variable(z).

term(X) :- name(X).
term(X) :- variable(X).

one_place_predicate('M').
one_place_predicate('B').

two_place_predicate('K').
two_place_predicate('L').
```

Programa 12 - Categorias básicas de L_1 apenas com variáveis *x*, *y* e *z*

Aqui para L_1, nos restringiremos apenas a esta versão com as três primeiras variáveis abreviadas para *x*, *y* e *z*. No entanto, caso seja preciso manter o mecanismo de indexação das variáveis, isso não seria difícil, ainda que o programa apresentado sirva

exclusivamente para o reconhecimento, como veremos no próximo parágrafo. Essa versão com as variáveis nomeadas como v_i é apresentada no Programa 13, abaixo.

```
% L1_Cat2.ari
%     (Dowty, Wall & Peters 1981: 56)
% Basic categories for L1 defined as expression predicate

name(d).
name(n).
name(j).
name(m).

variable(X) :- name(X,[118|Y]), name(Z,Y), integer(Z).

term(X) :- name(X).
term(X) :- variable(X).

one_place_predicate('M').
one_place_predicate('B').

two_place_predicate('K').
two_place_predicate('L').
```

Programa 13 - Categorias básicas de L_1 com variáveis de tipo v_i

A capacidade de apenas reconhecimento, mencionada no parágrafo anterior, se deve ao emprego do predicado pré-definido 'integer/1', que exige a instanciação do seu argumento: caso o seu argumento seja uma variável não instanciada, ele simplesmente falha.[74] Mas como essa não é uma questão prioritária para a Semântica de Montague (nem tampouco para a elaboração de um analisador gramatical, mas sim para um gerador de sentenças), vamos deixá-la de lado.

Como está implementado no Programa 13, o predicado "variable/1" testa se a primeira letra do termo instanciado como seu argumento é "v" (118 em ASCII) e depois testa se o resto é mesmo um número natural (através do predicado pré-definido "integer/1", ver capítulo 3.4.5, na página 37).[75] Para isso, o termo é convertido em uma lista em que

---

[74] Para fins de geração, uma forma de implementar o predicado 'variable/1' seria através da definição de um predicado capaz de também gerar os números naturais, como:

```
natural(0).
natural(X) :- natural(Y), X is Y + 1.
```

[75] Na verdade, no manual, os autores empregam apenas os números naturais; mas o predicado "integer/1" aceita qualquer número inteiro, o que inclui os negativos, que não fazem parte dos números natuarais. Para ser

cada caractere é convertido em seu respectivo valor ASCII, pelo predicado pré-definido 'name/1' (de onde se 'extrai' o primeiro caractere, que precisa ser necessariamente um "v"; sobre esse predicado, ver capítulo 3.4.2, na página 34), e depois a lista restante é novamente convertida a um termo, pelo mesmo predicado "name/1"; só então é que o novo termo é testado para ver se é mesmo um número natural. Assim, as variáveis nessa interpretação terão o formato: "v0", "v1", "v2", "v3"... Por simplicidade, aqui vamos empregar apenas a versão das variáveis abreviadas.

Com as expressões básicas determinadas, ainda é preciso incluir as duas novas regras para os quantificadores; isso é feito no Programa 14, abaixo.

```
% L1_Syn1.ari
%     (Dowty, Wall & Peters 1981: 56)
% Formation rules for L1

% Loading basic expressions

:- reconsult('L1_Cat1.ari').

% Formation rules

/*1*/ formula(F) :- F =.. [D,A], one_place_predicate(D),
       term(A).
/*2*/ formula(F) :- F =.. [G,A,B], two_place_predicate(G),
       term(A), term(B).
/*3*/ formula([not,F]) :- formula(F).
/*4*/ formula([F,and,P]) :- formula(F), formula(P).
/*5*/ formula([F,or,P]) :- formula(F), formula(P).
/*6*/ formula([F,if,P]) :- formula(F), formula(P).
/*7*/ formula([F,iff,P]) :- formula(F), formula(P).
/*8*/ formula([all,U,F]) :- variable(U), formula(F).
/*9*/ formula([some,U,F]) :- variable(U), formula(F).
```

Programa 14 - Regras de formação para L_1

---

mais fiel ao manual, ainda que apenas para reconhecimento, uma alternativa seria empregar a seguinte definição de "natural/1":

```
natural(X) :- integer(X), X >= 0.
```

Como a questão da indexação também não é de muita importância, sendo suficiente apenas que as variáveis estejam consistentemente indexadas de forma que as variáveis diferentes sejam efetivamente distintas umas das outras, e como isso é satisfeito pela implementação proposta no texto, também desconsideraremos essa variação.

As novas regras 8 e 9 determinam um novo tipo de fórmula em que os quantificadores ocupam a primeira posição de uma lista, na qual a segunda posição é ocupada por uma variável, e a terceira por uma outra fórmula.

Como a inclusão dessas duas novas regras não modifica em nada o analisador definido para a sintaxe de L_0, torna-se desnecessário e repetitivo a apresentação de um exemplo de análise.

4.3.2. Semântica de L_1

Se, na parte sintática, a língua L_1 é uma mera extensão de L_0, do ponto de vista semântico, a língua L_1 apresenta uma complexidade um pouco maior, imposta pela interpretação das variáveis. Mas apesar dessa complicação, todo o resto de L_1 pode ser interpretado como em L_0 e, na verdade, a complexa interpretação das variáveis de L_1 é composicionalmente acrescentada à interpretação dos outros elementos de L_1.

Sendo assim, nessa apresentação da semântica de L_1, vamos nos deter mais na interpretação das variáveis, já que a interpretação dos outros elementos comuns a L_0 já foi discutida anteriormente.

Intuitivamente, uma variável marca uma posição argumental que não está ocupada por um argumento específico e definido; nesse sentido, ela pode ser entendida como uma posição vazia onde cada um dos elementos do domínio vai ser testado segundo certos objetivos determinados pelos quantificadores.

Assim, uma sentença como "Todos os homens morrem" pode ser parafraseada como 'para cada um dos elementos do domínio, se esse elemento é homem, então ele morre', ou em notação do cálculo de predicados: "$\forall x[H(x) \rightarrow M(x)]$". Ou seja, para conhecermos o valor semântico da sentença "Todos os homens morrem" precisamos fazer todos os elementos do domínio ocupar, um de cada vez, a posição indicada pela variável de forma que a implicação resulte verdadeira para todos os indivíduos que satisfaçam a condição de serem homens.

Da mesma forma, a sentença "Algum homem morre" é parafraseada por 'para pelo menos um elemento do domínio, é verdadeiro que ele, ao mesmo tempo, é homem e morre': "$\exists x[H(x) \& M(x)]$", na mesma notação do cálculo de predicados. E também para "Algum homem morre", precisamos fazer todos os elementos do domínio ocupar, ainda um de cada vez, a posição da variável de forma a podermos constatar se, para algum deles, a conjunção ocorre ser verdadeira.[76]

Além das variáveis, outra mudança significativa na interpretação de L_1 é a menção explícita aos modelos nos quais a linguagem pode ser interpretada.[77] Assim, agora os valores semânticos não são mais funções absolutas, mas sim funções relativas a modelos e atribuições, como no Quadro 9 (DWP: 60).

1. Se u é uma variável, então $[\![u]\!]^{M,g} = g(u)$.
2. Se α é uma constante não-lógica, então $[\![\alpha]\!]^{M,g} = F(\alpha)$.

Quadro 9 – Valores semânticos para expressões básicas de L_1

Para a interpretação das constantes não-lógicas,[78] a atribuição de valores é feita através de uma função (F). Como essa função é relativa ao modelo no qual se está interpretando L_1, a parte simbólica da regra 2, acima, pode ser lida como 'o valor semântico para a constante não-lógica α, em relação ao modelo M e à atribuição g, é igual à aplicação da função F a α'.[79]

---

[76] As sentenças com quantificação existencial e universal sempre foram tradicionalmente traduzidas, respectivamente, pela conjunção e pela implicação; como a justificativa para essa escolha pode ser encontrada em qualquer manual de introdução à lógica, evitaremos essa discussão aqui.

[77] Como a noção de modelo só foi incluída no livro (DWP: 44-47)junto com a língua L_{0E} depois da discussão das regras de interpretação, ela não tinha sido incluída antes aqui.

[78] Como as constantes lógicas são inseridas sincategorematicamente, elas recebem interpretação diretamente nas respectivas regras semânticas.

[79] Parece que aqui os autores não foram tão precisos como seria possível. Como a interpretação varia segundo o modelo, a regra não poderia mencionar uma função F fixa, mas sim uma função relativa ao modelo; assim, ao invés de F, a função poderia ser F_M, por exemplo.

A função *F* sugerida para o modelo *M*, no qual os autores vão interpretar L_1, é apresentada no Quadro 10 (DWP: 61):[80]

F(j) = a
F(d) = b
F(n) = c
F(m) = a
F(M) = {a, b, c}
F(B) = {b, c}
F(K) = {⟨a, a⟩, ⟨a, b⟩, ⟨b, c⟩}
F(L) = {⟨a, c⟩, ⟨b, a⟩, ⟨c, a⟩, ⟨c, c⟩}

Quadro 10 – Função de atribuição para o modelo *M*

Através da regra 2 e da função *F*, acima, é possível interpretar qualquer constante não-lógica de L_1. Assim, por exemplo, $[\![d]\!]^{M,g}$ = F(d), $[\![M]\!]^{M,g}$ = F(M) e $[\![L]\!]^{M,g}$ = F(L), segundo a regra 2; e, segundo o modelo do Quadro 10, F(d) = b, F(M) = {a, b, c} e F(L) = {⟨a, c⟩, ⟨b, a⟩, ⟨c, a⟩, ⟨c, c⟩}. Assim, portanto, $[\![d]\!]^{M,g}$ = b, $[\![M]\!]^{M,g}$ = {a, b, c} e $[\![L]\!]^{M,g}$ = {⟨a, c⟩, ⟨b, a⟩, ⟨c, a⟩, ⟨c, c⟩}.

Já a função *g*, de atribuição de valores às variáveis, é constituída por uma atribuição arbitrária inicial (AAI). Para L_1, os autores sugerem 'g(x) = c', 'g(y) = b' e 'g(z) = a'. A arbitrariedade dessa atribuição inicial não afetará a interpretação de L_1, pois ela só garante que as variáveis tenham sempre alguma atribuição, para que as expressões nas quais elas ocorram possam sempre ser interpretadas;[81] como se verá nos exemplos, a interpretação das variáveis segundo as regras de quantificação vai permitir que essa AAI varie para se

---

[80] Apesar das regras semânticas para L_1 serem apresentadas no mesmo estilo da de L_{0E}, envolvendo aplicação funcional, a função *F* é ainda apresentada como em L_0, através da listagem exaustiva de elementos dos conjuntos. Na implementação em Prolog, manteremos esse último padrão.

[81] Sem essa AAI, seria impossível interpretar composicionalmente qualquer expressão na qual aparecesse alguma variável.

encontrar alguma atribuição *g'* que faça a fórmula verdadeira (no caso da quantificação existencial) ou para verificar se todas as atribuições *g'* fazem com que a fórmula seja verdadeira (no caso da quantificação universal).

Para formalizar, então, a interpretação de L_1, além das modificações relativas às referências a modelos (*M*) e atribuições (*g*), basta incluir na semântica de L_0 as duas regras correspondentes às quantificações existencial e universal (regras 10 e 11). Tal conjunto de regras é apresentado no Quadro 3, abaixo (DWP: 60).

3. Se δ é um predicado de um lugar e α é um termo, então $[\![\delta(\alpha)]\!]^{M,g} = [\![\delta]\!]^{M,g}([\![\alpha]\!]^{M,g})$.
4. Se γ é um predicado de dois lugares e α e β são termos, então $[\![\gamma(\alpha,\beta)]\!]^{M,g} = [[\![\gamma]\!]^{M,g}([\![\beta]\!]^{M,g})]([\![\alpha]\!]^{M,g})$.
5. Se ϕ é uma fórmula, então $[\![\neg\phi]\!]^{M,g} = 1$ sse $[\![\phi]\!]^{M,g} = 0$; caso contrário, $[\![\neg\phi]\!]^{M,g} = 0$.
6. Se ϕ e ψ são fórmulas, então $[\![\phi \wedge \psi]\!]^{M,g} = 1$ sse tanto $[\![\phi]\!]^{M,g} = 1$ quanto $[\![\psi]\!]^{M,g} = 1$; caso contrário, $[\![\phi \wedge \psi]\!]^{M,g} = 0$.
7. Se ϕ e ψ são fórmulas, então $[\![\phi \vee \psi]\!]^{M,g} = 1$ sse ou $[\![\phi]\!]^{M,g} = 1$ ou $[\![\psi]\!]^{M,g} = 1$; caso contrário, $[\![\phi \vee \psi]\!]^{M,g} = 0$.
8. Se ϕ e ψ são fórmulas, então $[\![\phi \rightarrow \psi]\!]^{M,g} = 1$ sse $[\![\phi]\!]^{M,g} = 0$ ou então $[\![\psi]\!]^{M,g} = 1$; caso contrário, $[\![\phi \rightarrow \psi]\!]^{M,g} = 0$.
9. Se ϕ e ψ são fórmulas, então $[\![\phi \leftrightarrow \psi]\!]^{M,g} = 1$ sse tanto $[\![\phi]\!]^{M,g} = 1$ quanto $[\![\psi]\!]^{M,g} = 1$, ou tanto $[\![\phi]\!]^{M,g} = 0$ quanto $[\![\psi]\!]^{M,g} = 0$; caso contrário, $[\![\phi \leftrightarrow \psi]\!]^{M,g} = 0$.
10. Se ϕ é uma fórmula e *u* é uma variável, então $[\![\forall u\phi]\!]^{M,g} = 1$ sse para toda atribuição de valores *g'*, tal que *g'* seja exatamente igual a *g* excetuando-se possivelmente a atribuição individual de *u* em *g'*, $[\![\phi]\!]^{M,g'} = 1$.
11. Se ϕ é uma fórmula e *u* é uma variável, então $[\![\exists u\phi]\!]^{M,g} = 1$ sse para alguma atribuição de valores *g'*, tal que *g'* seja exatamente igual a *g* excetuando-se possivelmente a atribuição individual de *u* em *g'*, $[\![\phi]\!]^{M,g'} = 1$.

Quadro 11 - Regras semânticas para L_1

A título de ilustração, vamos acompanhar a interpretação de algumas expressões de L_1 que envolvam a quantificação de variáveis. No livro (DWP: 63-65), os autores apresentam a sua interpretação ilustrativa partindo da parte mais interna da expressão para

depois acrescentar as quantificações; aqui, apresentaremos as interpretações em sentido inverso, desmembrando as expressões complexas em expressões cada vez mais simples.)

A expressão "∃zB(z)", por exemplo, é construída aplicando-se primeiro a regra sintática 9, do Quadro 7, à variável *z* e à fórmula *B(z)*; esta última, por sua vez, é construída através da aplicação da regra sintática 3, do mesmo Quadro 7, ao predicado de um lugar *B* e à variável *z*. A regra semântica 11, do Quadro 11, relativa à construção da quantificação existencial, vai exigir que haja pelo menos uma atribuição *g'* para a variável *z* que faça a expressão "B(z)" verdadeira; aplicando-se a regra semântica 3, para a interpretação das fórmulas compostas por um predicado de um lugar e um termo, vamos descobrir que o valor semântico de *B* é {b,c} ($[\![B]\!]^{M,g}$ = F(B) = {b, c}), que aplicado ao valor semântico de *z*, que é *a* ($[\![z]\!]^{M,g}$ = g(z) = a), resulta numa fórmula falsa, já que *a* não faz parte de *{b, c}*. No entanto, uma outra atribuição *g'*, exatamente igual a *g* só que se atribuindo agora o valor *b* à variável *z* (representa-se isso por $g' = g^{[z/b]}$) faz a expressão "B(z)" verdadeira; pois $[\![z]\!]^{M,g'}$ = g'(z) = $g^{[z/b]}$(z) = b, que faz parte de $[\![B]\!]^{M,g'}$. Isso já bastaria para fazer a expressão "∃zB(z)" verdadeira; porém, além de $g^{[z/b]}$, uma outra atribuição *g"*, tal que $g'' = g^{[z/c]}$, também faz a expressão verdadeira, pois $[\![z]\!]^{M,g''}$ = g"(z) = $g^{[z/c]}$(z) = c, que também faz parte de $[\![B]\!]^{M,g''}$. Resumindo, há duas atribuições ($g^{[z/b]}$ e $g^{[z/c]}$) que fazem a expressão "∃zB(z)" verdadeira, apesar dela não ser verdadeira para a AAI; e apenas uma delas seria suficiente para a verdade de expressão. Todas essas etapas podem ser esquematizadas como na Tabela 12, abaixo, onde as duas soluções aparecem nas linhas 10 e 13 (destacadas com traços duplos):

1.	$[\![\exists z B(z)]\!]^{M,g} = [\![B(z)]\!]^{M,g'}$ para algum g'	pela regra 11
2.	$[\![B(z)]\!]^{M,g'} = [\![B]\!]^{M,g'}([\![z]\!]^{M,g'})$	de 1, pela regra 3
3.	$[\![B]\!]^{M,g'}$ = F(B)	de 2, pela regra 2
4.	F(B) = {b, c}	de 3, pelo Quadro 10
5.	$[\![z]\!]^{M,g'}$ = g'(z)	de 2, pela regra 1

6.	g'(z) = a	de 5, por AAI (g' = g)
7.	$[\![B(z)]\!]^{M,g[z,a]} = 0$	de 2, 3, 4, 5 e 6
8.	g'(z) = b	de 5, para g' = $g^{z/b}$
9.	$[\![B(z)]\!]^{M,g[z,b]} = 1$	de 2, 3, 4, 5 e 8
10.	$[\![\exists zB(z)]\!]^{M,g[z,b]} = 1$	de 1 e 9
11.	g'(z) = c	de 5, para g' = $g^{z/c}$
12.	$[\![B(z)]\!]^{M,g[z,c]} = 1$	de 2, 3, 4, 5 e 11
13.	$[\![\exists zB(z)]\!]^{M,g[z,c]} = 1$	de 1 e 12

Tabela 12 – Etapas para o cálculo do valor da expressão "∃zB(z)"

Outra expressão, esta agora envolvendo a quantificação universal, que ocorre ser verdadeira no modelo M é "∀yM(y)". Sua construção se dá pela regra sintática 8, novamente do Quadro 7, cuja contraparte semântica é a regra 10, do Quadro 11, que exige que toda atribuição *g'* seja verdadeira quando aplicada à fórmula que faz parte da expressão mais complexa que contém a quantificação universal. Em outras palavras, para "∀yM(y)" ser verdadeira, todas as entidades do modelo precisam ser atribuídas à variável *y* e a expressão "M(y)" resultar verdadeira para todas elas. (Convém lembrar aqui que um modelo, além de ser composto pela função *F* (de atribuição de valores às constantes não-lógicas), é constituído também por uma ontologia (o conjunto *A*, de entidades reconhecidas pelo modelo). Em termos mais formais, o modelo é definido por um par ordenado composto por sua ontologia *A* e por uma função de atribuição *F*; ou seja, M = ⟨A,F⟩.) Como no exemplo anterior, a expressão "M(y)", que também é formada por um predicado de um lugar *M* e uma variável *y*, é construída através da mesma regra sintática 3, do Quadro 7, e interpretada pela regra semântica 3, do Quadro 11. Para o modelo *M*, ocorre que F(M) é exatamente igual à sua ontologia (ou seja, {a, b, c}), o que configura a verdade da expressão "∀yM(y)". Um esquema mais detalhado de todo esse cálculo é apresentado na Tabela 13, abaixo.

1.	$[\![\forall y M(y)]\!]^{M,g} = [\![M(y)]\!]^{M,g'}$ para todo g'	pela regra 10
2.	$[\![M(y)]\!]^{M,g'} = [\![M]\!]^{M,g'}([\![y]\!]^{M,g'})$	de 1, pela regra 3
3.	$[\![M]\!]^{M,g'}$ = F(M)	de 2, pela regra 2
4.	F(M) = {a, b, c}	de 3, pelo Quadro 10
5.	$[\![y]\!]^{M,g'}$ = g'(y)	de 2, pela regra 1
6.	g'(y) = b	de 5, por AAI (g' = g)
7.	$[\![M(y)]\!]^{M,g[y/b]} = 1$	de 2, 3, 4, 5 e 6
8.	g'(y) = a	de 5, para g' = $g^{y/a}$
9.	$[\![M(y)]\!]^{M,g[y/a]} = 1$	de 2, 3, 4, 5 e 8
10.	g'(y) = c	de 5, para g' = $g^{y/c}$
11.	$[\![M(y)]\!]^{M,g[y/c]} = 1$	de 2, 3, 4, 5 e 10
12.	$[\![\forall y M(y)]\!]^{M,g} = 1$	de 1, 7, 9 e 11

Tabela 13 - Etapas para o cálculo do valor da expressão "∀yM(y)"

Já a expressão "∀xB(x)" não é verdadeira no modelo *M*, pois uma das entidades de sua ontologia não pertence ao conjunto dos valores do predicado de um lugar *B*, ainda que as outras duas entidades restantes pertençam. Essa prova se encontra na Tabela 14, abaixo.

1.	$[\![\forall x B(x)]\!]^{M,g} = [\![B(x)]\!]^{M,g'}$ para todo g'	pela regra 10
2.	$[\![B(x)]\!]^{M,g'} = [\![B]\!]^{M,g'}([\![x]\!]^{M,g'})$	de 1, pela regra 3
3.	$[\![B]\!]^{M,g'}$ = F(B)	de 2, pela regra 2
4.	F(B) = {b, c}	de 3, pelo Quadro 10
5.	$[\![x]\!]^{M,g'}$ = g'(x)	de 2, pela regra 1

6.	g'(x) = a	de 5, por AAI (g'= g)
7.	$[\![B(x)]\!]^{M,g[x/a]} = 0$	de 2, 3, 4, 5 e 6
8.	$[\![\forall x B(x)]\!]^{M,g} = 0$	de 1 e 7

Tabela 14 – Prova da falsidade de "∀xB(x)"

Finalmente, apresenta-se na Tabela 15, abaixo, a interpretação para a expressão "∀x∃yL(x, y)". A solução se encontra na última linha da tabela, traçada em negrito, enquanto que as três sub-soluções necessárias para essa solução final aparecem nas linhas marcadas com traços duplos (há uma quarta sub-solução: $[\![\exists y L(x,y)]\!]^{M,g[xc,y/c]} = 1$, que não chega a afetar a interpretação, já que as três apresentadas são suficientes).

1.	$[\![\forall x \exists y L(x,y)]\!]^{M,g} = [\![\exists y L(x,y)]\!]^{M,g'}$ para todo g'	pela regra 10
2.	$[\![\exists y L(x,y)]\!]^{M,g'} = [\![L(x,y)]\!]^{M,g''}$ para algum g''	de 1, pela regra 11
3.	$[\![L(x,y)]\!]^{M,g''} = [[\![L]\!]^{M,g''}([\![y]\!]^{M,g''})]([\![x]\!]^{M,g''})$	de 2, pela regra 4
4.	$[\![L]\!]^{M,g''} = F(L)$	de 3, pela regra 2
5.	F(L) = {⟨a, c⟩, ⟨b, a⟩, ⟨c, a⟩, ⟨c, c⟩}	de 4, pelo Quadro 10
6.	$[\![y]\!]^{M,g''} = g''(y)$	de 3, pela regra 1
7.	g"(y) = a	de 6, para $g'' = g^{y/a}$
8.	$[\![x]\!]^{M,g''} = g''(x)$	de 3, pela regra 1
9.	g"(x) = c	de 8, para $g'' = g^{x/c}$
10.	$[\![\exists y L(x,y)]\!]^{M,g[x/c,y/a]} = 1$	de 2 a 9
11.	g"(x) = b	de 8, para $g'' = g^{x/b}$
12.	$[\![\exists y L(x,y)]\!]^{M,g[x/b,y/a]} = 1$	de 2 a 8 e 11
13.	g"(y) = c	de 6, para $g'' = g^{y/c}$

14.	g"(x) = a	de 8, para g" = $g^{x/a}$
15.	$[\![\exists yL(x,y)]\!]^{M,g[x/a,y/c]} = 1$	de 2 a 6, 8, 13 e 14
16.	$[\![\forall x\exists yL(x,y)]\!]^{M,g} = 1$	de 1, 10, 12 e 15

Tabela 15 – Procedimento de interpretação para a expressão "∀x∃yL(x,y)"

(Para os mais curiosos, que ainda quiserem conferir o valor de algumas outras expressões, segundo o modelo *M*, são falsas: "∃x∀yL(x, y)", "∃x∀yK(x, y)" e "∀x∃yK(x, y)"; no entanto, a penúltima dessas expressões seria verdadeira num modelo *M'* que fosse exatamente igual a *M*, exceto por 'F(K) = {⟨a, a⟩, ⟨a, b⟩, ⟨a, c⟩}'.)

O Programa 15, abaixo, executa em Prolog a interpretação da linguagem L_1.

```
% L1_Sem6.ari
%      (Dowty, Wall & Peters 1980: 60)
% Semantics of L1, with model and assignment

% Loading syntactic rules

:- reconsult('L1_Syn1.ari').

% Basic expressions
% "semval" abrevia SEMantic VALue

semval(Variable,Model,Assignment,Value) :-
      variable(Variable),
      g(Variable,Model,Assignment,Value).
semval(Constant,Model,Assignment,Value) :-
      f(Model,[Constant,Value]).

% Rules

/*9*/ semval([some,Variable,Formula],Model,Assignment) :-
      variable(Variable),
      semval(Formula,Model,Assignment),
      some(Variable,Model,Assignment).
/*8*/ semval([all,Variable,Formula],Model,Assignment) :-
      setof(A,a(Model,A),Domain),
      variable(Variable),
      all(Model,Variable,Domain,Formula,Assignment).
/*7*/ semval([Formula1,iff,Formula2],Model,Assignment) :-
      ((semval(Formula1,Model,Assignment),
        semval(Formula2,Model,Assignment));
       (not semval(Formula1,Model,Assignment),
        not semval(Formula2,Model,Assignment))).
/*6*/ semval([Formula1,if,Formula2],Model,Assignment) :-
      (not semval(Formula1,Model,Assignment);
       semval(Formula2,Model,Assignment)).
```

```
/*5*/ semval([Formula1,or,Formula2],Model,Assignment) :-
      (semval(Formula1,Model,Assignment);
       semval(Formula2,Model,Assignment)).
/*4*/ semval([Formula1,and,Formula2],Model,Assignment) :-
      semval(Formula1,Model,Assignment),
      semval(Formula2,Model,Assignment).
/*3*/ semval([not,Formula],Model,Assignment):-
      not semval(Formula,Model,Assignment).
/*2*/ semval(Formula,Model,[Assignment]) :-
      formula(Formula), Formula =.. [Pred,Arg1,Arg2],
      semval(Pred,Model,Assignment,[SemVal1,SemVal2]),
      semval(Arg1,Model,Assignment,SemVal1),
      semval(Arg2,Model,Assignment,SemVal2).
/*1*/ semval(Formula,Model,[Assignment]) :-
      formula(Formula), Formula =.. [Pred,Arg],
      semval(Pred,Model,Assignment,SemVal),
      semval(Arg,Model,Assignment,SemVal).

% Assignment function

g(x,Model,[[x,Entity],Y,Z],Entity) :-
      a(Model,Entity).
g(y,Model,[X,[y,Entity],Z],Entity) :-
      a(Model,Entity).
g(z,Model,[X,Y,[z,Entity]],Entity) :-
      a(Model,Entity).

apply(x,[[x,Entity],Y,Z],Entity).
apply(y,[X,[y,Entity],Z],Entity).
apply(z,[X,Y,[z,Entity]],Entity).

some(Variable,Model,[Assignment]) :-
      !, apply(Variable,Assignment,Entity),
      a(Model,Entity).
some(Variable,Model,Assignment) :-
      a(Model,Entity),
      some_aux(Variable,Entity,Assignment).

some_aux(Variable,Entity,[Assignment]) :-
      apply(Variable,Assignment,Entity).
some_aux(Variable,Entity,[Assignment|Assignments]) :-
      apply(Variable,Assignment,Entity),
      some_aux(Variable,Entity,Assignments).

all(Model,Variable,[Entity],Formula,[Assignment]) :-
      !, apply(Variable,Assignment,Entity),
      semval(Formula,Model,[Assignment]).
all(Model,Variable,[Entity|DomainRest],Formula,[G|Gs]) :-
      apply(Variable,G,Entity),
      semval(Formula,Model,[G]),
      all(Model,Variable,DomainRest,Formula,Gs).

% Models 'm' and 'm1' for L1 interpretation

a(m,a).
a(m,b).
a(m,c).
```

```
f(m,[j,a]).
f(m,[d,b]).
f(m,[n,c]).
f(m,[m,a]).
f(m,['M',a]).
f(m,['M',b]).
f(m,['M',c]).
f(m,['B',b]).
f(m,['B',c]).
f(m,['K',[a,a]]).
f(m,['K',[a,b]]).
f(m,['K',[b,c]]).
f(m,['L',[a,c]]).
f(m,['L',[b,a]]).
f(m,['L',[c,a]]).
f(m,['L',[c,c]]).

a(m1,a).
a(m1,b).
a(m1,c).

f(m1,[j,a]).
f(m1,[d,b]).
f(m1,[n,c]).
f(m1,[m,a]).
f(m1,['M',a]).
f(m1,['M',b]).
f(m1,['M',c]).
f(m1,['B',b]).
f(m1,['B',c]).
f(m1,['K',[a,a]]).
f(m1,['K',[a,b]]).
f(m1,['K',[a,c]]).
f(m1,['L',[a,c]]).
f(m1,['L',[b,a]]).
f(m1,['L',[c,a]]).
f(m1,['L',[c,c]]).
```

Programa 15 – Programa para a interpretação de L_1

Os valores semânticos para as expressões básicas são atribuídos pelo predicado "semval/4"; ao contrário da maioria das outras regras semânticas, que utiliza o predicado "semval/3", esse predicado retorna o valor semântico (*semantic value*) como último argumento a partir da expressão básica (primeiro argumento), do modelo (segundo argumento) e da atribuição *g* (terceiro argumento). Como nas outras regras o que se está computando é o valor semântico de fórmulas, que é sempre um valor de verdade, preferiu-se optar por não registrá-lo como um argumento a mais.

Para as variáveis, esse valor é computado pela regra "semval(Variable, Model, Assignment, Value) :- variable(Variable), g(Variable, Model, Assignment, Value)." O que

essa regra faz é atribuir à variável 'Variable', depois de confirmar que ela é mesmo uma variável, um valor 'Value' através da função de atribuição "g/4". Esse predicado, por sua vez, é definido por três regras: 1) "g(x, Model, [[x, Entity], Y, Z], Entity) :- a(Model, Entity).", 2) "g(y, Model, [X, [y, Entity], Z], Entity) :- a(Model, Entity)." e 3) "g(z, Model, [X, Y, [z, Entity]], Entity) :- a(Model, Entity).", que atribuem a cada uma das três variáveis um dos valores da ontologia do modelo em que se está interpretando a linguagem.[82] A atribuição de valores às variáveis é representada então por uma lista de três pares ordenados, em que cada um deles representa a atribuição a uma das variáveis, sempre na mesma ordem (*x*, *y* e *z*); cada uma das regras especifica apenas o valor de uma das variáveis, ao contrário da AAI, mas isso não constitui obstáculo, já que a principal função de *g* é permitir as operações de substituição de valores para uma determinada variável de cada vez, que é exatamente o que o predicado "g/4" faz. Finalmente, os valores semânticos atribuídos às variáveis são obtidos através do predicado "a/2" que designa o conjunto das entidades que fazem parte do modelo especificado.

Já para as constantes não-lógicas, o valor semântico é estabelecido através da regra semântica 2 "semval(Constant, Model, Assignment, Value) :- f(Model, [Constant, Value]).". Nessa regra, o valor semântico das constantes é atribuído pela função *F*, definida aqui pelo predicado "f/2". Exatamente como na versão redigida da função *F*, aqui também o predicado "f/2" é independente da atribuição de valores às variáveis, deixando sempre o respectivo argumento ('Assignment') não instanciado; a conseqüência disso é que a determinação dos valores das constantes será sempre compatível com qualquer atribuição de valor para as variáveis.

(Convém nesse momento, antes de começar a apresentação das regras de interpretação das fórmulas, uma explicação da representação dos modelos no Programa 15, acima. Cada modelo nesse programa é especificado por um conjunto de dois tipos de cláusulas: o das cláusulas com o predicado "a/2", que designam a ontologia do modelo, e o

---

[82] Como apenas essas três variáveis serão usadas nos exemplos, optou-se por definir por listagem a atribuição dos valores a cada variável. Para ser mais fiel ao fato de que as variáveis são em número ilimitado, esse predicado precisaria ser modificado para isso, o que alteraria completamente o estabelecimento das atribuições *g* como uma lista de três pares ordenados, um para cada atribuição de valor às variáveis *x*, *y* e *z*, exatamente nessa mesma ordem.

das cláusulas com o predicado "f/2", que determinam as relações entre as constantes não-lógicas e os respectivos valores semânticos, como acabamos de explicar. Isso satisfaz a definição dos modelos como "M = ⟨A, F⟩"; para se recuperar em formato de lista cada um desses conjuntos no programa, basta usar o predicado pré-definido "setof/3" (ver capítulo 3.4.7, na página 38), que constrói um conjunto com todos os valores que satisfazem uma determinada variável de um predicado. Assim, o conjunto *A* pode ser reconstruído a partir de "setof(X, a(Model, X), A).", com seu resultado apresentado na variável 'A'; e, da mesma forma, o conjunto *F* pode ser obtido de "setof(X, f(Model, X), F)." No entanto, como nenhum desses conjuntos é necessário para a implementação proposta aqui, essas definições não foram incluídas no Programa 15. Finalmente, pode-se observar que foram estipulados dois modelos no programa (*m* e *m1*, que serão usados para exemplificar as implementações dos quantificadores; eles replicam, respectivamente *M* e *M'*); apesar deles terem sido diretamente inscritos no programa, isso não era indispensável: eles poderiam ter sido definidos em programas separados e consultados pelo Programa 15.[83])

Explicadas as regras semânticas para as expressões básicas, pode-se passar para as regras de expressões complexas. As regras para as expressões compostas por predicados de um ou dois lugares, mais os seus respectivos argumentos, continuam exatamente iguais às de L_0 (as únicas diferenças são o acréscimo dos argumentos relativos ao modelo e à atribuição, e a ordem dos predicados no corpo das regras, para diminuir um pouco o percurso executado pelo interpretador Prolog nas provas). Assim, além de processar o valor de verdade dessas expressões, como em L_0, essas regras agora também informam para que modelos e atribuições de valores às variáveis isso ocorre. Uma expressão como "B(x)" pode ser avaliada pelo programa através de "semval('B'(x), Model, G).", apresentando as seguintes respostas: 1) 'Model = m', 'G = [[[x, b], Y, Z]]',[84] 2) 'Model = m', 'G = [[[x, c],

---

[83] Outra opção para a definição teria sido registrar o argumento relativo ao modelo como variável para as cláusulas que fossem comuns a todos os modelos, como "a(M,c)." (que poderia ser parafraseado por 'a entidade *c* faz parte da ontologia de qualquer modelo'). Com isso, a diferença entre *m* e *m1* nos obrigaria a escrever cláusulas específicas apenas para a estipulação dos valores do predicado de dois lugares *K* ("f(m,['K',...])." e "f(m1,['K',...])."., onde as reticências precisam ser completadas por todos os valores apropriados); como a construção do modelo não é parte essencial da semântica apresentada aqui, essa questão não será aprofundada.

[84] Na verdade, o interpretador Prolog vai apresentar algumas variáveis numeradas (como "_0C1F", por exemplo), e não essas letras maiúsculas, mas isso é irrelevante.

Y, Z]]', 3) 'Model = m1', 'G = [[[x, b], Y, Z]]' e 4) 'Model = m1', 'G = [[[x, c], Y, Z]]', pois, em ambos os modelos *m* e *m1*, a fórmula "'B'(x)" só é verdadeira quando os valores *b* e *c* são atribuídos à variável "x"; já uma fórmula como "M(z)", avaliada por "semval('M'(z), Model, G).", que é verdadeira em ambos os modelos (pois "M(z)" é verdadeira para a atribuição de todos os elementos de *A* à variável "z"), resulta em: 1) 'Model = m', 'G = [[X, Y, [z, a]]]', 2) 'Model = m', 'G = [[X, Y, [z, b]]]', 3) 'Model = m', 'G = [[X, Y, [z, c]]]', 4) 'Model = m1', 'G = [[X, Y, [z, a]]]', 5) 'Model = m1', 'G = [[X, Y, [z, b]]]' e 6) 'Model = m1', 'G = [[X, Y, [z, c]]]'.

Uma expressão com um predicado de dois lugares como "K(x, y)" seria avaliada através de "semval('K'(x, y), Model, G).", e resultaria verdadeira nas seguintes atribuições e modelos: 1) 'Model = m', 'G = [[[x, a], [y,a], Z]]', 2) 'Model = m', 'G = [[[x, a], [y, b], Z]]', 3) 'Model = m', 'G = [[[x, b], [y, c], Z]]', 4) 'Model = m1', 'G = [[[x, a], [y,a], Z]]', 5) 'Model = m1', 'G = [[[x, a], [y, b], Z]]', 6) 'Model = m1', 'G = [[[x, a], [y, c], Z]]'. O outro predicado de dois lugares, numa expressão como "L(z, y)", seria avaliado por "semval('L'(z, y), Model, G)." como sendo verdadeiro apenas nos seguintes casos: 1) 'Model = m', 'G = [[X, [y, c], [z, a]]]', 2) 'Model = m', 'G = [[X, [y, a], [z, b]]]', 3) 'Model = m', 'G = [[X, [y, a], [z, c]]]', 4) 'Model = m', 'G = [[X, [y, c], [z, c]]]', 5) 'Model = m1', 'G = [[X, [y, c], [z, a]]]', 6) 'Model = m1', 'G = [[X, [y, a], [z, b]]]', 7) 'Model = m1', 'G = [[X, [y, a], [z, c]]]', 8) 'Model = m1', 'G = [[X, [y, c], [z, c]]]'.

(Como as regras 3 a 7, para os conectivos lógicos, não funcionam adequadamente nessa implementação, já que as variáveis não são instanciadas com o uso do predicado *not*, elas não serão explicadas aqui. Mas como esse problema não se colocava na implementação de L_{0E}, e como ele poderia ser facilmente adaptado à gramática de L_1, essa questão será ignorada.)

Para implementar a regra da quantificação universal, empregou-se o predicado 'all/5' na regra "semval([all, Variable, Formula], Model, Assignment) :- setof(A, a(Model, A), Domain), variable(Variable), all(Model, Variable, Domain, Formula, Assignment).", que avalia se todos os elementos do domínio (conjunto definido pelo predicado pré-definido 'setof/3', ver capítulo 3.4.7, na página 38) foram atribuídos à variável determinada em cada atribuição *g* para a fórmula no escopo do quantificador. Em outros termos, esse

predicado avalia se existe algum conjunto de atribuições de valores às variáveis que satisfaça a fórmula quando todos os valores do domínio são atribuídos à variável especificada. Assim, para o modelo sugerido, o predicado 'all/5' vai conferir se, para a variável "x", por exemplo, a fórmula satisfaz o conjunto de atribuições "[[[x, a], Y1, Z1], [[x, b], Y2, Z2], [[x, c], Y3, Z3]]". Isso é feito principalmente pela parte recursiva da definição de 'all/5', "all(Model, Variable, [Entity | DomainRest], Formula, [G | Gs]) :- apply(Variable, G, Entity), semval(Formula, Model, [G]), all(Model, Variable, DomainRest, Formula, Gs).", que cria uma atribuição aplicando o primeiro valor do domínio (pela variável 'Entity'), através de 'apply/3' (que é muito semelhante ao predicado 'g/4', mas que é independente do modelo e por isso não atribui um valor específico; ele apenas aplica um valor dado à variável), depois testa se a fórmula é verdadeira para essa atribuição e torna a repetir essas operações para os valores restantes do domínio ('DomainRest'), pela reaplicação do mesmo predicado 'all/5'. A condição de término "all(Model, Variable, [Domain], Formula, [Assignment]) :- apply(Variable, Assignment, Domain), semval(Formula, Model, [Assignment])." só é invocada quando resta apenas um único elemento no domínio ('[Domain]'), encerrando o ciclo recursivo.

A fórmula "$\forall$xM(x)", que é verdadeira para o modelo *M*, seria avaliada pela cláusula "semval([all, x, 'M'(x)], Model, G)." da seguinte maneira: essa cláusula casa com a regra de quantificação universal mencionada no início do parágrafo anterior. Para satisfazer o primeiro predicado do corpo da regra ("setof(A, a(Model, A), Domain)"), o interpretador Prolog vai criar uma lista 'Domain' com todos os valores de 'A' que satisfaçam o predicado "a(Model, A)"; para 'Model = m', esse conjunto é "[a, b, c]". O próximo predicado a ser satisfeito é "variable(x)", que apenas garante que "x" seja mesmo um variável; como ele é satisfeito, o interpretador pode passar ao predicado "all(m, x, [a, b, c], 'M'(x), Assignment)". Como o domínio ainda não é uma lista de um único elemento, o interpretador casa esse predicado com a segunda cláusula da definição de 'all/5'; assim, o valor "a" é aplicado à variável "x", resultando na atribuição "[[x,a], Y, Z]", e testa-se o valor semântico de "M(x)" para essa atribuição ("semval('M'(x), m, [[[x, a], Y, Z]])"), que resulta ser verdadeiro. Com isso os outros valores do domínio podem ser testados ("all(m, x, [b, c], 'M'(x), Gs)"); da mesma forma, aplica-se agora o valor "b" à variável "x", resultando numa atribuição que também faz "M(x)" verdadeira. Finalmente, restando

apenas um elemento no domínio, a condição de término é atingida ("all(m, x, [c], 'M'(x), [Assignment])"), agora é o valor "c" que é atribuído à variável "x", o que também resulta numa atribuição que satisfaz "M(x)". Assim, o predicado 'all/5' invocado por "semval([all, x, 'M'(x)], Model, G)." é completamente satisfeito para 'Model = m' e 'G = [[[x, a], Y1, Z1], [[x, b], Y2, Z2], [[x, c], Y3, Z3]]', pois as atribuições testadas são reunidas num único conjunto ('[G|Gs]'). (Para ser exaustivo, é preciso ainda dizer que essa mesma fórmula também seria verdadeira no modelo *M'*; mas como o predicado de um lugar "M" em ambos os modelos recebe exatamente o mesmo valor, é desnecessário repetir o procedimento para *M'*, que seria idêntico ao de *M*.)

Já a implementação da quantificação existencial exigiu um predicado um pouco mais complexo. Devido à possibilidade de uma fórmula quantificada universalmente estar no escopo de um quantificador existencial, o cálculo do seu valor semântico precisa considerar dois tipos de atribuições: além das atribuições simples, resultado da avaliação do valor semântico das fórmulas não quantificadas ou quantificadas apenas existencialmente, é preciso também considerar conjuntos de atribuições, resultado da avaliação de fórmulas com quantificação universal. Assim, para avaliar a quantificação existencial, a regra será "semval([some, Variable, Formula], Model, Assignment) :- variabel(Variable), semval(Formula, Model, Assignment), some(Variable, Model, Assignment)."; nessa regra, depois de garantido que 'Variable' é mesmo uma variável, procura-se uma atribuição ou um conjunto delas ('Assignment'), que satisfaça o predicado "some/3". No caso de uma única atribuição, esse predicado garante que alguma entidade do modelo ("a(Model, Entity)") esteja atribuída à variável especificada nessa atribuição ("apply(Variable, Assignment, Entity)").

(Antes de explicarmos como o predicado "some/3" trata os conjuntos de atribuições, vamos apresentar os exemplos de avaliação da quantificação existencial para fórmulas compostas por predicados de um lugar e quando ela está no escopo da quantificação universal.)

Avaliada pela cláusula "semval([some, x, 'B'(x)], Model, G).", a fórmula "$\exists$xB(x)", que é verdadeira no modelo *M* para as atribuições $g^{x/b}$ e $g^{x/c}$, será calculada do seguinte modo: casando com a regra de quantificação existencial "semval([some, Variable,

Formula], Model, Assignment) :- variable(Variable), semval(Formula, Model, Assignment), some(Variable, Model, Assignment).", e depois de confirmar que *x* é uma variável ("variable(x)"), encontra-se uma atribuição de valores que satisfaça "B(x)" ("semval('B'(x), Model, G). A primeira atribuição encontrada pelo interpretador, "[[[x, a], Y, Z]]" (correspondente a $g^{x/a}$), não satisfaz "B(x)" no modelo *M*, por isso o Prolog busca a segunda atribuição, "[[[x, b], Y, Z]]" (correspondente a $g^{x/b}$); como, nessa atribuição, a expressão "B(x)" é verdadeira, o interpretador passa ao último predicado da definição com os seguintes valores: "some(x, m, [[[x, b], Y, Z]])". Sendo a atribuição uma lista de um único elemento, esse predicado casa com a primeira cláusula da definição de "some/3", conferindo se nessa atribuição o valor *b*, aplicado à variável "x", é uma entidade do modelo *M*; como é, a regra é satisfeita, resultando em 'Model = m' e 'G = [[[x, b], Y, Z]]'. Mas isso não esgota todas as possibilidades de resposta, caso solicitássemos a próxima resposta, o interpretador Prolog tentaria agora a atribuição "[[[x, c], Y, Z]]" (correspondente a $g^{x/c}$).[85] Essa atribuição também satisfaz a expressão "B(x)" e, mais uma vez, o predicado "some/3" confere que o valor *c* aplicado à variável "x" é uma entidade do modelo *M*; assim, o segundo resultado é: 'Model = m' e 'G = [[[x, c], Y, Z]]'. (Ainda haveria mais duas respostas, para os mesmos valores atribuídos à mesma variável, só que para o modelo *M'*; como ainda aqui os procedimentos seriam exatamente os mesmos apresentados imediatamente acima, eles não serão repetidos para 'Model = m1'.)

Além de se avaliar a quantificação para predicados de um lugar, pode-se também avaliar o valor semântico dos predicados de dois lugares para a quantificação de cada uma das duas variáveis desse predicado. Para a expressão "$\forall x \exists y L(x,y)$", a única verdadeira no modelo *M* para essa combinação de quantificadores, essa avaliação será feita por "semval([all, x, [some, y, 'L'(x, y)]], Model, G)." Conforme já foi visto, a regra de quantificação universal procura por um conjunto de atribuições de forma que para a atribuição de cada valor a "x" haja um valor atribuído a "y" de tal forma que essa atribuição conjunta satisfaça o predicado "L(x, y)". Para o modelo *M*, dois conjuntos de atribuições

---

[85] O interpretador Prolog não tenta uma outra solução para o predicado "some/4", apesar de ainda haver uma segunda cláusula nessa definição, por causa do *cut* (!) no início do corpo da primeira regra, que interrompe a busca de alternativas depois de atingido. Sem esse artifício, o interpretador apresentaria uma segunda solução:

satisfazem esses requisitos: $\{g^{x/a,\ y/c}, g^{x/b,\ y/a}, g^{x/c,\ y/a}\}$ e $\{g^{x/a,\ y/c}, g^{x/b,\ y/a}, g^{x/c,\ y/c}\}$. Esses resultados, segundo o Programa 15, seriam: 1) 'Model = m' e 'G = [[[x, a], [y, c], Z1], [[x, b], [y, a], Z2], [[x, c], [y, a], Z3]]' e 2) 'Model = m' e 'G = [[[x, a], [y, c], Z1], [[x, b], [y, a], Z2], [[x, c], [y, c], Z3]]'. (Mais uma vez, o modelo *M'* ainda permite outras duas respostas exatamente iguais que novamente não serão apresentadas pelo mesmo motivo já explicado.)

Como não há no modelo *M* nenhum predicado que satisfaça uma expressão com a forma "$\exists x \forall y$...(x,y)", onde as reticências ocupam o lugar do suposto predicado, chega-se finalmente ao motivo da estipulação do modelo *M'*. Para exemplificar a avaliação das expressões com quantificação universal no escopo da quantificação existencial, que exigiu a inclusão do predicado "some_aux/3" na definição de "some/3", foi preciso criar um segundo modelo no qual algum predicado tivesse um valor para o seu primeiro argumento para o qual todos os valores do domínio pudessem ser valor do seu segundo argumento. No modelo *M*, o predicado de dois lugares "K" só não satisfaz essa exigência porque o seu último par ordenado é ⟨b, c⟩; como os dois primeiros são ⟨a, a⟩ e ⟨a, b⟩, caso esse último fosse ⟨a, c⟩, ao invés de ⟨b, c⟩, então esse predicado poderia ser usado como exemplo. Assim, no modelo *M'*, o valor semântico de "K" ("F(K)") é "{⟨a, a⟩, ⟨a, b⟩, ⟨a, c⟩}; e essa é a única diferença entre os modelos *M* e *M'*, conforme já tinha sido observado. Agora então é possível explicar o predicado auxiliar "some_aux/3" e exemplificar sua utilização na avaliação da quantificação existencial quando há uma quantificação universal em seu escopo.

Quando se avalia uma expressão quantificada universalmente, ao invés de uma atribuição simples, chega-se a um conjunto de atribuições, onde cada valor do domínio é atribuído à variável especificada. Assim, uma expressão como "$\forall y$K(x, y)" (que precisará ser avaliada primeiro para que depois se possa avaliar composicionalmente "$\exists x \forall y$K(x, y)") vai ser verdadeira em ambos os modelos para os seguintes conjuntos de atribuições: 1) $\{g^{x/a,\ y/a}, g^{x/a,\ y/b}, g^{x/b,\ y/c}\}$, para o modelo *M*, e 2) $\{g^{x/a,\ y/a}, g^{x/a,\ y/b}, g^{x/a,\ y/c}\}$, para o modelo *M'*.

'Model = m' e 'G = [[x, b], Y, Z]', exatamente igual à primeira, não fosse a falta dos colchetes externos, mas inadequada ao formato de conjunto de atribuições estabelecido para as fórmulas.

Esses conjuntos de atribuições não permitem que o predicado "some/3" seja casado com a primeira cláusula de sua definição, que, como já vimos, serve exclusivamente para conjuntos de atribuições unitárias. Casando com a segunda cláusula da definição, sua avaliação vai ser transferida para "some_aux/3", estabelecendo ainda uma das entidades do modelo para ser testada ("a(Model, Entity)"); o teste é para saber se, em cada atribuição do conjunto de atribuições ('[Assignment | Assignments]'), o mesmo valor é atribuído à variável especificada. Isso é feito confirmando se a primeira atribuição do conjunto resulta da aplicação do valor à variável ("apply(Variable, Assignment, Entity)"), enviando o resto desse conjunto recursivamente para a reavaliação através do mesmo predicado "some_aux/3", até que reste apenas uma atribuição nesse conjunto; nesse caso, o predicado casa com a primeira cláusula da definição de "some_aux/3", terminando satisfatoriamente o procedimento se essa última atribuição também resultar da aplicação do mesmo valor à mesma variável.

Dessa maneira, a avaliação de "$\exists x \forall y K(x,y)$" pela cláusula "semval([some, x, [all, y, 'K'(x, y)]], Model, G)." apresenta um único resultado: 'Model = m1' e 'G = [[[x, a], [y, a], Z1], [[x, a], [y, b], Z2], [[x, a], [y, c], Z3]]'.

(Para os leitores que ainda tenham alguma curiosidade, a expressão "$\forall y \exists x K(x, y)$" (avaliada através de "semval([all, y, [some, x, 'K'(x, y)], Model, G).") é verdadeira em ambos os modelos: 1) em *M*, para o conjunto de atribuições "[[[x, a], [y, a], Z1], [[x, a], [y, b], Z2], [[x, b], [y, c], Z3]]", e 2) em *M'*, para o conjunto "[[[x, a], [y, a], Z1], [[x, a], [y, b], Z2], [[x, a], [y, c], Z3]]". Outras combinações além das discutidas aqui são falsas em ambos os modelos, e também não apresentam nenhuma solução quando avaliadas pelo Programa 15.)

4.4. Língua L_{type}

4.4.1. Sintaxe de L_{type}

A última língua desenvolvida até aqui só dispunha de variáveis para as categorias que designavam indivíduos; línguas desse tipo são conhecidas como línguas de primeira ordem (*first order languages*). As línguas que apresentam variáveis para a categoria das relações e das propriedades são chamadas de língua de segunda ordem (*second order language*). No entanto, a língua L_{type}, que passaremos a apresentar agora, é uma língua que apresenta variáveis para quaisquer de suas categorias; línguas como essas são conhecidas como línguas de ordem superior (*high order languages*).

Assim, além das quantificações para variáveis de indivíduos, a língua L_{type} vai permitir quantificar variáveis para qualquer tipo de expressão. Com as línguas anteriores, poderíamos expressar apenas coisas como 'existe alguém tal que ele corre velozmente' – "$\exists X[(v(c))(X)]$";[86] mas com L_{type} podemos exprimir também coisas como 'existe alguma coisa que o João faz velozmente' – "$\exists P[(v(P))(j)]$", ou como 'existe um certo modo no qual João corre' – "$\exists M(M(c))(j)]$".

Para isso, antes de qualquer coisa, precisamos de um meio de designar as infinitas categorias de L_{type}.[87] Isso pode ser feito através de uma definição recursiva dos tipos de L_{type}:

1. "e" é um tipo,
2. "t" é um tipo,
3. se X e Y são tipos, então $\langle X, Y\rangle$ é um tipo.

---

[86] Aqui, o "c" representa o predicado que designa a ação de correr e o "v" representa um modificador de predicados, que em língua natural normalmente corresponde aos advérbios, que indica que ele é praticado de forma veloz. Estamos usando também o mesmo recurso notacional do Prolog, indicando as variáveis através de letras maiúsculas.

[87] Como deve ter ficado claro, esses são apenas meios para designarmos as categorias. Há aqui uma distinção entre as categorias sintáticas, que são conjuntos de expressões, e os seus rótulos, que são os indicadores dessas categorias.

Através das duas primeiras regras dessa definição, "e" e "t" são tipos; pela aplicação da terceira regra a esses dois primeiros tipos, sabemos que "⟨e, t⟩", "⟨t, e⟩", "⟨e, e⟩" e "⟨t, t⟩" também são tipos. Reaplicando essa terceira regra a todos esses seis tipos, obtemos novos tipos: "⟨e, ⟨e, t⟩⟩", "⟨⟨e, t⟩, t⟩" e "⟨⟨e, t⟩, ⟨e,t⟩⟩", por exemplo; e a reaplicação recursiva dessa terceira regra nos leva a tipos cada vez mais complexos: "⟨e, ⟨e, ⟨e, t⟩⟩⟩", "⟨⟨⟨e, t⟩, ⟨e, t⟩⟩, t⟩", e assim por diante. Essa definição estabalece uma quantidade infinita de tipos através da recursividade em sua terceira cláusula.

A implementação dessa definição em Prolog é extremamente simples, como se pode constatar no Programa 16 abaixo.[88]

```
% Type.ari
% Recursive definition of the set of types of L-type
% Dowty, Wall & Peters 1980: 89

type(e).
type(t).
type([X,Y]) :- type(X), type(Y).
```

Programa 16 – Definição em Prolog dos tipos de L_{type}

A única diferença entre a definição em Prolog e a definição discursiva anterior é que em Prolog os tipos compostos são delimitados por colchetes, enquanto que na primeira os delimitadores eram parênteses angulados. Assim, o que era "⟨e, t⟩" na definição discursiva, em Prolog será "[e,t]"; o mesmo acontece com os seguintes pares:

- "⟨e, ⟨e, t⟩⟩" e "[e,[e,t]]",
- "⟨⟨e, t⟩, ⟨e, t⟩⟩" e "[[e,t],[e,t]]",

[88] Devido ao não-determinismo desta implementação, esse programa não se adequaria bem a um gerador de linguagem; uma maneira de solucionar isso seria implementando uma gramática para tipos finitos. (Se solicitado, o programa geraria seqüencialmente os seguintes tipos: "e", "t", "[e,e]", "[e,t]", "[e,[e,e]]", "[e,[e,t]]", "[e,[e,[e,e]" etc. Como se pode ver, como a recursividade se dá pela esquerda, o programa tem dificuldade para atingir soluções que começasse, por exemplo, com "[t,...]".) No apêndice, pode-se encontrar três dessas implementações (seções 8.2.1, 8.2.2 e 0); a diferença entre elas está apenas na posição em que a complexidade se localiza. (Todos os programas do apêndice geram seqüencialmente: "e", "t", "[e,e]", "[e,t]", "[t,e]", "[t,t]", "[e,[e,e]]", "[e,[e,t]]", "[e,[t,e]]", "[e,[t,t]]", "[t,[e,e]]" etc. No entanto, o tipo mais complexo que eles geram é "[[[t,t],[t,t]],[[t,t],[t,t]]]"; a definição em DCG gera apenas até o tipo "⟨⟨t, t⟩, ⟨t, t⟩⟩", no seguinte formato: "[<,<,t,',',t,>,<,t,',',t,>,>]", mas que também pode ser apresentado no formato anterior.) Da forma como está implementada aqui, essa gramática infinita poderia causar problemas às regras sintáticas de quantificação; porém, essa dificuldade foi superada no próprio programa do analisador sintático,

- "⟨e, ⟨e, ⟨e, t⟩⟩⟩" e "[e,[e,[e,t]]]",
- e, finalmente, "⟨⟨⟨e,t⟩, ⟨e, t⟩⟩, t⟩" e "[[[e,t],[e,t]],t]".

A partir dessa distinção entre categorias sintáticas e seus respectivos rótulos, podemos introduzir o símbolo ME_a (do inglês *meaningful expression*) para designar então os próprios conjuntos de expressões de tipo *a*. Com isso, se considerarmos que as sentenças são expressões do tipo "t" e os termos (como "j", no exemplo acima) são do tipo "e", podemos dizer que um predicado como "c" (ainda do exemplo acima) é do tipo "⟨e, t⟩", ou seja, um predicado de um lugar: uma expressão que toma outra de tipo "e" para formar uma expressão de tipo "t". Dessa mesma forma, o nosso modificador "v" de predicados de um lugar pertence então à classe das expressões que tomam um predicado de um lugar para formar um novo predicado de um lugar; ou seja, uma expressão do tipo "⟨⟨e, t⟩, ⟨e, t⟩⟩". Agora então, podemos dizer que em "∃P[(v(P))(j)]", "P" é uma variável de tipo "⟨e, t⟩", já que ela ocupa o lugar de um predicado de um lugar; já em "∃M[(M(c))(j)]", "M" é uma variável de tipo "⟨⟨e, t⟩, ⟨e, t⟩⟩", pois ela ocupa o mesmo lugar que um modificador de predicados de um lugar.

Em L_{type}, essas variáveis, bem como as constantes também, vão ser representadas por símbolos que deixam essa característica mais clara. As constantes de L_{type} serão designadas através de símbolos como "$c_{n, a}$", onde "n" é um número natural e "a" um tipo; assim, por exemplo, "$c_{0, e}$" é a primeira constante do tipo "e", "$c_{2, \langle e, t\rangle}$" é a terceira constante do tipo "⟨e, t⟩" e "$c_{6, \langle\langle e, t\rangle, \langle e, t\rangle\rangle}$" é a sétima constante do tipo "⟨⟨e, t⟩, ⟨e, t⟩⟩". As variáveis, por sua vez, serão designadas como "$v_{n, a}$"; assim como com as constantes, portanto, "$v_{1, e}$" é a segunda variável do tipo "e", "$v_{4, \langle e, t\rangle}$" é a quinta variável do tipo "⟨e, t⟩" e "$v_{9, \langle\langle e, t\rangle, \langle e, t\rangle\rangle}$" é a décima variável do tipo "⟨⟨e, t⟩, ⟨e, t⟩⟩".

Admitindo então que "j" corresponda a "$c_{0, e}$", "c" a "$c_{2, \langle e, t\rangle}$", "v" a "$c_{6, \langle\langle e, t\rangle, \langle e, t\rangle\rangle}$", "X" a "$v_{1, e}$", "P" a "$v_{4, \langle e, t\rangle}$" e "M" a "$v_{9, \langle\langle e, t\rangle, \langle e, t\rangle\rangle}$", as três sentenças dadas como exemplo de quantificação acima podem ser traduzidas para L_{type}:

---

estabelecendo uma exigência em relação ao tipo de variável nas próprias regras de quantificação, ao contrário do que fazia supor a definição discursiva de DWP: 92 (ver adiante, p. 122).

- "$\exists X[(v(c))(X)]$" fica como "$\exists v_{1,\,e}[(c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}(c_{2,\,\langle e,\,t\rangle}))(v_{1,\,e})]$"
- "$\exists P[(v(P))(j)]$" fica como "$\exists v_{4,\,\langle e,\,t\rangle}[(c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}(\,v_{4,\,\langle e,\,t\rangle}))(c_{0,\,e})]$"
- "$\exists M[(M(c))(j)]$" fica como "$\exists v_{9,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}[(v_{9,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}(c_{2,\,\langle e,\,t\rangle}))(c_{0,\,e})]$"

Para lidar com expressões como essas, os autores (DWP: 91-92) apresentam as seguintes regras:

A. O conjunto de *tipos* de L_{type} é dado pela definição recursiva em 1-3 acima.

B. As *expressões básicas* de L_{type} (*BE_a – Basic Expressions*) são constantes e variáveis, de forma que:

   1. para todo tipo *a*, o conjunto de *constantes não-lógicas de tipo a*, denotado por *Con_a*, contém constantes "$c_{n,\,a}$", para cada número natural *n*, de forma que $Con_a \supseteq BE_a$;

   2. para todo tipo *a*, o conjunto de *variáveis de tipo a*, denotado por *Var_a*, contém variáveis "$v_{n,\,a}$", para cada número natural *n*, de forma que $Var_a \supseteq BE_a$.

C. As regras sintáticas que definem o conjunto das *expressões significadoras de tipo a*, denotado por "*ME_a*" (*Meaningful Expressions*), para cada tipo *a* (que é apenas o conjunto das expressões bem-formadas de cada tipo), são constituídas pela seguinte definição recursiva:

   1. Para qualquer tipo *a*, se $\alpha \in BE_a$, então $\alpha \in ME_a$.

   2. Para quaisquer tipos *a* e *b*, se $\alpha \in ME_{\langle a,\,b\rangle}$ e $\beta \in ME_a$, então $\alpha(\beta) \in ME_b$.

   3. - 7. Se ϕ e $\psi \in ME_t$, então o mesmo acontece com:

      3. $\neg\phi$
      4. $[\phi \wedge \psi]$
      5. $[\phi \vee \psi]$
      6. $[\phi \rightarrow \psi]$
      7. $[\phi \leftrightarrow \psi]$

8. Se $\phi \in ME_t$ e u é uma variável (de qualquer tipo), então $\forall u\phi \in ME_t$.

9. Se $\phi \in ME_t$ e u é uma variável (de qualquer tipo), então $\exists u\phi \in ME_t$.

Assim, expressões como "$\exists v_{1,\,e}c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}(c_{2,\,\langle e,\,t\rangle})(v_{1,\,e})$",[89] por mais complexas que pareçam, podem ser analisadas pelas regras acima, como é possível ver no quadro.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{v_{1,\,e}}\,B.2 \quad \dfrac{}{c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}}\,B.1}{c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}(c_{2,\,\langle e,\,t\rangle})}\,2 \quad \dfrac{}{c_{2,\,\langle e,\,t\rangle}}\,B.1}{c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}(c_{2,\,\langle e,\,t\rangle})(v_{1,\,e})}\,2 \quad \dfrac{}{v_{1,\,e}}\,B.2}{\exists v_{1,\,e}c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}(c_{2,\,\langle e,\,t\rangle})(v_{1,\,e})}\,C.9$$

Quadro 12 – Análise sintática de "$\exists v_{1,\,e}c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}(c_{2,\,\langle e,\,t\rangle})(v_{1,\,e})$"

A principal diferença de L_{type} em relação à sintaxe das gramáticas anteriores é que, devido à inclusão do cálculo de tipos, ao invés das várias regras para as fórmulas constituídas por predicados, temos uma única regra para o chamado cancelamento de tipo (determinado na regra 2), de modo que a formação dessas expressões decorre apenas dessa única regra. Para fórmulas compostas a partir de predicados de apenas um lugar, somente um cancelamento de tipo basta: como os predicados de um lugar são expressões do tipo "⟨e, t⟩", enquanto seus argumentos são expressões de tipo "e", a regra 2 reúne duas expressões de cada um desses tipos, resultando numa expressão de tipo "t", que é o tipo das fórmulas. Já para fórmulas compostas a partir de predicados de dois lugares, precisaremos recorrer duas vezes ao cancelamento de tipos: na primeira vez, a regra 2 reúne expressões de tipo "⟨e, ⟨e, t⟩⟩" (que é o tipo dos predicados de dois lugares) e "e", resultando numa expressão

[89] Observe que aqui o escopo da quantificação não é mais indicado pelos colchetes; da forma como as regras estão estipuladas, os colchetes só aparecem quando se emprega algum dos conectivos lógicos. Isso seria, inclusive, desnecessário, pois os tipos dos termos, junto com o cancelamento de tipos definido pela regra C.2. e ainda uma exigência sobre a conexidade das expressões (que ainda não foi feita explicitamente), bastariam para demarcar esse escopo. Repare, por exemplo, que uma análise alternativa resultaria de combinar primeiro "$c_{2,\,\langle e,\,t\rangle}(v_{1,\,e})$", resultando numa expressão de tipo "t"; no entanto, isso impediria a combinação com $c_{6,\,\langle\langle e,\,t\rangle,\,\langle e,\,t\rangle\rangle}$, o que nos deixaria com uma expressão desconexa. Apesar de não ter sido explicitamente mencionado, o conceito de conexidade é importante nesse tipo de gramática (sobre esse conceito, ver Ajdukiewicz 1935). Essa opção permite ainda algumas expressões ambíguas, o que vai seria bem aproveitado na adaptação desse tipo de gramática para analisar o português que não possui esses recursos desambiguadores.

de tipo "⟨e, t⟩", exatamente o mesmo tipo dos predicados de um lugar; um segundo cancelamento reúne esse nova expressão de tipo "⟨e, t⟩" a outra expressão de tipo "e", resultando mais uma vez numa expressão de tipo "t".

No entanto, essa diferença é responsável por uma das principais vantagens sintáticas da adoção do cálculo de tipos: como já estivemos considerando, não há necessidade da inclusão de mais nenhuma regra para lidarmos com os modificadores dos predicados (que são expressões que geralmente correspondem aos advérbios das línguas naturais). Ao serem classificados como expressões do tipo "⟨⟨e, t⟩, ⟨e, t⟩⟩", os modificadores dos predicados de um lugar formam novos predicados de um lugar ao serem reunidos a predicados desse mesmo tipo, recorrendo-se à mesma regra 2. Da mesma forma, qualquer combinação entre duas expressões quaisquer pode ser considerada como um cancelamento de tipo; um advérbio sentencial, por exemplo, pode ser considerado como uma expressão do tipo "⟨t, t⟩", e, quando combinado a uma sentença (cujo tipo é "t"), forma uma outra sentença. Essa mesma regra 2, em português, reuniria o advérbio "felizmente", de tipo "⟨t, t⟩", e a sentença "João saiu", de tipo "t", formando a sentença "felizmente João saiu", cujo tipo também é "t".

Mas como nem tudo dá sempre assim tão certo, essa vantagem tem lá o seu preço em relação às línguas naturais. Em L_{type}, os modificadores de predicados recebem tipos diferentes de acordo com os tipos de predicados que eles modificam; assim, os modificadores de predicados de um lugar (de tipo "⟨e, t⟩") são do tipo "⟨⟨e, t⟩, ⟨e, t⟩⟩", os modificadores de predicados de dois lugares ("⟨e, ⟨e, t⟩⟩") são do tipo "⟨⟨e, ⟨e, t⟩⟩, ⟨e, ⟨e, t⟩⟩⟩", e assim por diante. Nas línguas naturais, os advérbios parecem não fazer essa distinção; em português, por exemplo, o advérbio "rápido" pode modificar o verbo "correr" quando ele é usado tanto intransitiva, quanto transitivamente: "João corre rápido" e João corre rápido a maratona"; na primeira sentença "rápido" seria do tipo "⟨⟨e, t⟩, ⟨e, t⟩⟩", enquanto que na segunda seria "⟨⟨e, ⟨e, t⟩⟩, ⟨e, ⟨e, t⟩⟩⟩".[90]

---

[90] Essa, porém, é uma restrição que se aplica apenas às gramáticas categoriais da época em que o manual de DWP foi escrito; as gramáticas categoriais modernas dispõem de recursos para solucionar esse problema.

Uma outra diferença em relação à sintaxe das línguas anteriores é apontada pelos próprios autores em um dos exercícios propostos (DWP: 92): ao contrário das línguas anteriores, onde os conectivos lógicos podiam ser tratados tanto categoremática quanto sincategorematicamente, o tratamento categoremático dos conectivos binários em L_{type}, por exemplo, estipulados como do tipo "⟨t, ⟨t, t⟩⟩", levaria à construção de expressões inadequadas como "∧ $v_{3,t}$", do tipo "⟨t, t⟩". Sendo assim, vamos manter aqui o tratamento sincategoremático.

Feitas essas observações, podemos passar à implementação em Prolog.

```
% lt_syn1.ari
% Syntax of Ltype
% Dowty, Wall & Peters 1981: 91-2

%%%%%%%%%%%%%%%%%%%%%%%
% A. Type definition %
%%%%%%%%%%%%%%%%%%%%%%%

:- reconsult('type.ari').

%%%%%%%%%%%%%%%%%%%%%%%%%
% B. Basic expressions %
%%%%%%%%%%%%%%%%%%%%%%%%%

% Constants
be(Constant,Type) :-
      Constant =.. [c,Number,Type],
      integer(Number),
      type(Type).
% Variables
be(Variable,A) :-
      Variable =.. [v,Number,Type],
      integer(Number),
      type(Type).

%%%%%%%%%%%%%%%%%%%%%%%
% C. Syntactic rules %
%%%%%%%%%%%%%%%%%%%%%%%

% 9. Existential quantification
me([some,Variable|Expression],t) :-
      Variable =.. [v,Number,Type],
      integer(Number),
      type(Type),
      me(Expression,t).
% 8. Universal quantification
me([all,Variable|Expression],t) :-
      Variable =.. [v,Number,Type],
      integer(Number),
      type(Type),
      me(Expression,t).
```

```
% 7. Equivalence
me([Expression1,iff,Expression2],t) :-
      me(Expression1,t),
      me(Expression2,t).
% 6. Implication
me([Expression1,if,Expression2],t) :-
      me(Expression1,t),
      me(Expression2,t).
% 5. Disjunction
me([Expression1,or,Expression2],t) :-
      me(Expression1,t),
      me(Expression2,t).
% 4. Conjunction
me([Expression1,and,Expression2],t) :-
      me(Expression1,t),
      me(Expression2,t).
% 3. Negation
me([not,Expression],t) :-
      me(Expression,t).
% 2. Type cancelation
  % Left association, right branching
me([X,Y|Z],Type) :-
      be(X,XType),
      be(Y,YType),
      cancel(XType,YType,XYType),
      me(Z,ZType),
      cancel(XYType,ZType,Type).
  % Right association, left branching
me([X|Y],Type) :-
      be(X,XType),
      me(Y,YType),
      cancel(XType,YType,Type).
% 1. Basic expressions
me([BasicExpression],Type) :-
      be(BasicExpression,Type).

%%%%%%%%%%%%%%%%%%%%%
% Type cancellation %
%%%%%%%%%%%%%%%%%%%%%

cancel([A,B],A,B).
```

Programa 17 – Sintaxe de L_{type} em Prolog

O Programa 17, acima, é uma mera adaptação do Programa 14, que faz a análise de L_1. A primeira adaptação ocorreu nos termos (constantes e variáveis), pois, em L_{type}, eles têm a forma "$x_{n,\ a}$" (onde x é "c", para as constantes, ou "v", para as variáveis; n é um número natural; e a é um tipo); portanto, em Prolog, eles serão ou "c(Number, Type)" ou "v(Number, Type)". Assim, para constatar se uma expressão é uma expressão básica ("be/2"), o primeiro passo é desmembrá-la através do predicado "=..", exigindo que o primeiro elemento da lista obtida seja ou um "c" ou um "v"; para os elementos restantes,

exige-se respectivamente que sejam um número inteiro ("integer(Number)")[91] e um tipo ("type(Type)"). Com isso, alguns exemplos de expressões básicas permitidas pelo Programa 17 são: "c(0, e)", "c(2, [[e, t], [e, t]])", "v(47, [e, [e, t]])" e "v(508, [[[e, t], [t, e]], [[e, e], [t, t]]])".

Definidas as expressões básicas, as expressões compostas de L_{type} são estipuladas pelo predicado "me/2" (*meaningful expression*). As cláusulas de sua definição são fundamentalmente iguais às da definição das fórmulas de L_1. A principal diferença é a introdução da regra de cancelamento de tipo, substituindo as antigas cláusulas relativas às fórmulas constituídas com predicados. No entanto, como esse cancelamento é recursivo, ele é obtido através das cláusulas "me([X, Y | Z], Type) :- be(X, XType), be(Y, YType), cancel(XType, YType, XYType), me(Z, ZType), cancel(XYType, ZType, Type)." e "me([X | Y], Type) :- be(X, XType), me(Y, YType), cancel(XType, YType, Type).", junto com a cláusula "cancel([A, B], A, B)." (onde o cancelamento de tipo é efetivamente definido).[92] As duas primeiras cláusulas garantem a aplicação recursiva, de forma que a primeira delas é responsável pela ramificação à direita enquanto a segunda faz a ramificação pela esquerda. Com isso, as estruturas ramificadas à direita devem ser mais facilmente reconhecidas do que as de ramificação à esquerda; ou seja, mesmo com ambas conseguindo reconhecer as mesmas cadeias, as expressões estruturadas pela direita devem ser processadas em menos etapas do que as estruturadas pela esquerda. Mas isso é o máximo que vamos nos permitir comentar sobre mais essa questão de desempenho do programa.

A outra mudança está relacionada à ausência de marcação para o escopo dos quantificadores. Como as fórmulas quantificadas de L_1 eram do tipo "[Quantificador, Variável, Fórmula]", a exigência de que a variável "Fórmula" fosse uma fórmula fazia com

---

[91] O predicado "integer/1" não aceita apenas números naturais. Como os inteiros ainda incluem os negativos, o programa permite expressões básicas do tipo "c(-3,...)". Essa divergência, no entanto, não chega a ser substancial, e por isso será desprezada. De qualquer maneira, alternativas para esse problema já foram apresentadas na nota 75 (página 97).

[92] Tentei aplicar o método de execução parcial, distribuindo o cancelamento pelas cláusulas de recursão ("me([X|Y],B) :- me([X],[A,B]), me(Y,A), !." e "me([X,Y|Z],B) :- me([X,Y],[A,B]), me(Z,A)."), mas isso não funcionou; como essa é uma questão eminentemente de desempenho do programa, não tentarei explorar o tema aqui.

que esta variável sempre fosse instanciada por uma lista. Em L_{type} não há esta necessidade: como através do cálculo categorial é possível deduzir esse escopo, não é preciso delimitá-lo. Dessa maneira, as expressões quantificadas podem ter o formato "[Quantificador, Variável | Fórmula]"; nesse caso, para o quantificador "all", a variável "v(0, e)" e a lista "[c(0, [e,t]), c(0, e)]", obteríamos a lista "[all, v(0,e) | [c(0, [e,t]), c(0, e)]]", que é equivalente à lista "[all, v(0,e), c(0, [e,t]), c(0, e)]". Isso é feito sincategorematicamente para os dois quantificadores pelas cláusulas "me([all, Variable | Expression], t) :- Variable =.. [v, Number, Type], integer(Number), type(Type), me(Expression, t)." e "me([some, Variable | Expression], t) :- Variable =.. [v, Number, Type], integer(Number), type(Type), me(Expression, t)".

Contudo, essa implementação do cancelamento sofre limitações na aplicação da sua recursividade. Da forma como está definida, ela só permite que duas expressões básicas combinem entre si, combinando depois com uma terceira expressão coesa; ou então que uma única expressão combine com outra expressão coesa. Isso impede, o reconhecimento, dentre outras, de expressões com estruturas do tipo "[[a [b c]] d]", como em "[c(0, [e, [e, t]]), c(0, [e, e]), c(0, e), c(1, e)]", apresentada no Quadro 13.[93]

$$\frac{\dfrac{\dfrac{\dfrac{\dfrac{}{c(0,[e,[e,t]])}\,[e,[e,t]] \quad \dfrac{}{c(0,[e,e])}\,[e,e] \quad \dfrac{}{c(0,e)}\,e}{[c(0,[e,e]),c(0,e)]}\,e \quad \dfrac{}{c(1,e)}\,e}{[c(0,[e,[e,t]]),c(0,[e,e]),c(0,e)]}\,[e,t]}{[c(0,[e,[e,t]]),c(0,[e,e]),c(0,e),c(1,e)]}\,t$$

Quadro 13 –Estrutura de "[c(0,[e,[e,t]]),c(0,[e,e]),c(0,e),c(1,e)]"

Essa limitação está diretamente ligada ao modo como as listas são estabelecidas em Prolog. Como toda lista é sempre composta por um átomo seguido de uma lista (relembrando, uma lista como "[a, b, c, d]", na verdade, é tratada como "[a | [b | [c | [d | []]]]]"), ela é uma estrutura com ramificação sempre pela direita. Além disso, o Programa 17 não consegue lidar ainda com os conectivos lógicos binários ("and", "or", "if" e "iff").

[93] Diferentemente do que foi feito no Quadro 12, a partir de agora serão omitidas as referências às regras de formação; por simplicidade, marcaremos apenas o tipo resultante das combinações.

Mas mesmo que elas pudessem ser processadas, a negação e as quantificações só apareceriam com escopo largo (como, para a expressão "[not, c(0, [e, t]), c(0, e), and, c(0, [e, [e, t]]), c(1, e), c(2, e)]", no Quadro 14); no entanto, em L_{type}, a mesma expressão ainda poderia ser analisada com o escopo estreito do primeiro conectivo (como no Quadro 15).[94]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{not}}\ \dfrac{}{c(0,[e,t])}^{[e,t]}\ \dfrac{}{c(0,e)}^{e}}{[c(0,[e,t]),c(0,e)]}^{t}\quad \text{and}\quad \dfrac{\dfrac{}{c(0,[e,[e,t]])}^{[e,[e,t]]}\ \dfrac{}{c(1,e)}^{e}}{[c(0,[e,[e,t]]),c(1,e)]}^{[e,t]}\ \dfrac{}{c(2,e)}^{e}}{[c(0,[e,[e,t]]),c(1,e),c(2,e)]}^{t}}{[c(0,[e,t]),c(0,e),\text{and},c(0,[e,[e,t]]),c(1,e),c(2,e)]}^{t}}{[\text{not},c(0,[e,t]),c(0,e),\text{and},c(0,[e,[e,t]]),c(1,e),c(2,e)]}^{t}$$

Quadro 14 – Exemplo de análise com escopo largo

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\text{not}}\ \dfrac{}{c(0,[e,t])}^{[e,t]}\ \dfrac{}{c(0,e)}^{e}}{[c(0,[e,t]),c(0,e)]}^{t}\quad \text{and}\quad \dfrac{\dfrac{}{c(0,[e,[e,t]])}^{[e,[e,t]]}\ \dfrac{}{c(1,e)}^{e}}{[c(0,[e,[e,t]]),c(1,e)]}^{[e,t]}\ \dfrac{}{c(2,e)}^{e}}{[\text{not},c(0,[e,t]),c(0,e)]}^{t}\quad \dfrac{}{[c(0,[e,[e,t]]),c(1,e),c(2,e)]}^{t}}{[\text{not},c(0,[e,t]),c(0,e),\text{and},c(0,[e,[e,t]]),c(1,e),c(2,e)]}^{t}$$

Quadro 15 – Exemplo de análise com escopo estreito

O Programa 17, contudo, não consegue se beneficiar dessa característica de L_{type}. Como as expressões com conectivos lógicos, por exemplo, são construídas através do encadeamento de uma variável, o conectivo e outra variável ("[Expression1, and, Expression2]"), e dado ainda que todas as expressões de tipo "t" serem também uma lista, o resultado disso é que as expressões com conectivos serem sempre constituídas de sub-listas, e não de simples encadeamento de expressões básicas. Para solucionar estes problemas, é preciso eliminar essa característica da construção das expressões, introduzindo dois

[94] Observe como, através do cálculo categorial, a expressão com escopo estreito ocupa menos linhas no quadro do que a mesma expressão com escopo largo. Na possibilidade de haver um processamento concomitante, a quantidades de linhas no quadro poderia sugerir uma métrica que parece estar de acordo com as observações psicolingüísticas de que as estruturas ramificadas à esquerda são priorizadas no processamento lingüístico humano (remeter à bibliografia ou a discussão posterior).

mecanismos de decisão para localizar as várias possibilidades de combinação das expressões de L_{type}. É exatamente isso o que o Programa 18, abaixo, faz. [95]

```
% lt_syn2.ari
% Syntax of Ltype with split and append
% Dowty, Wall & Peters 1981: 91-2

%%%%%%%%%%%%%%%%%%%%
% A. Set of types %
%%%%%%%%%%%%%%%%%%%%

% Loading 'type/2' definition
:- reconsult('type.ari').

%%%%%%%%%%%%%%%%%%%%%%%%%
% B. Basic expressions %
%%%%%%%%%%%%%%%%%%%%%%%%%

% 1. Non-logical constants
con(Con,Type) :-
      Con =.. [c,Number,Type],
      integer(Number),
      type(Type).

% 2. Variables
var(Var,Type) :-
      Var =.. [v,Number,Type],
      integer(Number),
      type(Type).

%%%%%%%%%%%%%%%%%%%%%%
% C. Syntactic rules %
%%%%%%%%%%%%%%%%%%%%%%

% 1. Basic expressions
% a. Non-logical constants
me([Con],Type,Con) :-
      con(Con,Type).
% b. Variables
me([Var],Type,Var) :-
      var(Var,Type).

% 2. Type cancelation
me(ME,Type1,[Structure1,Structure2]) :-
      split(ME,ME1,ME2),
      me(ME1,[Type2,Type1],Structure1),
      me(ME2,Type2,Structure2).
```

[95] No apêndice, há ainda uma versão da sintaxe de L_{type} em DCG (ver 8.3.1, na página 166) e outra por deslocamento-e-redução (ver 8.3.2, na página 168). A versão em DCG, como está, não funciona direito: apesar dela conseguir apresentar um primeiro resultado, caso ela seja solicitada a buscar outras alternativas ela entra numa recursão típica de recursividade à esquerda em algoritmo de busca em profundidade; como a versão escolhida foi outra, não nos foi possível aprofundar aqui nesta questão.

```
% 3. Negation
me([not|ME],t,[not,Structure]) :-
       me(ME,t,Structure).

% 4. Conjunction
me(ME,t,[Structure1,and,Structure2]) :-
       append(ME1,[and|ME2],ME),
       me(ME1,t,Structure1),
       me(ME2,t,Structure2).

% 5. Disjunction
me(ME,t,[Structure1,or,Structure2]) :-
       append(ME1,[or|ME2],ME),
       me(ME1,t,Structure1),
       me(ME2,t,Structure2).

% 6. Implication
me(ME,t,[Structure1,if,Structure2]) :-
       append(ME1,[if|ME2],ME),
       me(ME1,t,Structure1),
       me(ME2,t,Structure2).

% 7. Equivalence
me(ME,t,[Structure1,iff,Structure2]) :-
       append(ME1,[iff|ME2],ME),
       me(ME1,t,Structure1),
       me(ME2,t,Structure2).

% 8. Universal quantification
me([all,Var|ME],t,[all,Var,Structure]) :-
       var(Var,Type),
       me(ME,t,Structure).

% 9. Existential quantification
me([some,Var|ME],t,[some,Var,Structure]) :-
       var(Var,Type),
       me(ME,t,Structure).

%%%%%%%%%%%%%%%%%%%%%%%
% Auxiliar predicates %
%%%%%%%%%%%%%%%%%%%%%%%

% Loading 'split/3' definition
:- reconsult('split.ari').

% Loading 'append/3' definition
:- reconsult('append.ari').

% Drive predicate
parse(Expression) :-
       rule(Expression,Type,Structure),
       write('Type: '),
       write(Type),
       nl,
       write('Structure: '),
       nl,
       write(Structure),
```

```
        nl.
```

Programa 18 – Sintaxe de L_{type} por divisão da expressão

As expressões básicas desse Programa 18 são essencialmente semelhantes às do Programa 17; há, no entanto, uma insignificante diferença formal: antes, era definido o conjunto de expressões básicas da língua; agora, os conjuntos das constantes não-lógicas e o das variáveis são individualmente definidos. A opção pela definição independente dos conjuntos das variáveis e das constantes parece estar mais de acordo com as duas regras B.1 e B.2, da definição discursiva. As expressões básicas são definidas como expressões significadoras através das duas primeiras cláusulas do predicado 'me/3': o primeiro ("me([Con], Type, Con) :- con(Con, Type).") trata das constantes, enquanto o segundo ("me([Var], Type, Var) :- var(Var, Type).") diz respeito às variáveis.

A regra C.2, responsável pelo cancelamento de tipo, é executada no programa pela cláusula "me(ME, Type1, [Structure1, Structure2]) :- split(ME, ME1, ME2), me(ME1, [Type2, Type1], Structure1), me(ME2, Type2, Structure2).". Aqui aparece o primeiro mecanismo de decisão mencionado acima: o predicado 'split/3', definido no Programa 19, abaixo, vai permitir explorar as várias possibilidades de se dividir uma lista em duas, de forma que nenhuma delas seja uma lista vazia. É este predicado 'split/3' que vai permitir separar uma expressão complexa em duas partes que permitam o cancelamento de tipo. A cláusula pode ser "lida" da seguinte maneira: uma expressão "ME" é uma expressão significadora do tipo "Type1" e com estrutura "[Structure1, Structure2]" se pudermos dividi-la em duas outras expressões "ME1" e "ME2", tal que "ME1" seja uma expressão significadora de tipo "[Type2, Type1]" e com estrutura "Structure1", e que "ME2" seja uma expressão significadora de tipo "Type2" (que unifica com o primeiro tipo da lista de tipos de "ME1", induzindo o cancelamento de tipos) e com estrutura "Structure2" (assim também se constrói uma representação através dos colchetes da estrutura da expressão).

```
% split.ari
% Spliting a list on two lists
%
% :- split([a],_,_).          > fail
% :- split([a,b],X,Y).        > X = [a]    Y = [b]
% :- split([a,b,c,d],X,Y).    > X = [a]    Y = [b,c,d]
%                       > X = [a,b] Y = [c,d]
%                       > X = [a,b,c]     Y = [d]

split([A],_,_) :- !, fail.
```

```
split([A|B],[A],B).
split([A|D],[A|B],C) :- split(D,B,C).
```

Programa 19 – Definição de 'split/3' para dividir uma lista em duas

Essa cláusula é a que vai ser usada na análise de expressões como "[c(0, [e, t]), c(0, e)]" e "[c(0, [e, [e, t]]), c(1, e), c(2, e)]" que, ao solicitarmos sua análise através das cláusulas "me([c(0, [e, t]), c(0, e)])." e "me([c(1, [e, [e, t]]), c(1, e), c(2, e)]).",[96] vão resultar respectivamente em: 1)"Type: t" e "Structure: [c(0, [e, t]), c(0, e)]", e 2) "Type: t" e "Structure: [[c(1, [e, [e, t]]), c(1, e)], c(2, e)]" (na segunda resposta, os colchetes reunindo as duas primeiras expressões básicas indicam que essas duas expressões se combinam primeiro, para só depois se combinarem com a terceira expressão básica).

A regra sincategoremática da negação (regra C.3) está implementada na cláusula "me([not | ME], t, [not, Structure]) :- me(ME, t, Structure)." que é satisfeita quando encontra uma expressão encabeçada por "not" e o restante da expressão é do tipo "t", construindo ainda sua representação estrutural. Para a expressão "[not, c(0, [e, [e, t]]), c(1, e), c(2, e)]", o resultado da análise é: "Type: t" e "Structure: [not, [[c(0, [e, [e, t]]), c(1, e)], c(2, e)]]" (os colchetes introduzidos agora indicam primeiro a combinação de "c(0, [e, [e, t]])" com "c(1, e)", depois a combinação desses dois juntos com "c(2, e)" e finalmente a combinação de "not" com "[[c(0, [e, [e, t]]), c(1, e)], c(2, e)]").

As regras C.4 a C.7, que constroem as expressões com os conectivos binários também sincategorematicamente, funcionam todas exatamente da mesma maneira: através do segundo mecanismo de decisão, definido pelo predicado 'append/3' (ver capítulo 3.5.2, na página 44), busca-se na expressão complexa um dos conectivos binários. Vamos explicar através da regra C.6, que forma as implicações: a cláusula é "me(ME, t, [Structure1, if, Structure2]) :- append(ME1, [if | ME2], ME), me(ME1, t, Structure1), me(ME2, t, Structure2).". Esta cláusula pode ser entendida como: uma expressão "ME" é

---

[96] O Programa 18 conta com uma cláusula de inicialização "me(ME) :- me(ME, Type, Structure), write('Type: '), write(Type), nl, write('Structure: '), nl, write(Structure), nl.", que invoca o analisador ("me(ME, Type, Structure)") e depois apresenta o resultado no seguinte formato (onde as reticências são substituídas pelos valores encontrados):

Type: ...
Structure:
...

do tipo "t" e tem estrutura "[Structure1, if, Structure2]" quando ela puder ser dividida em duas partes, a primeira delas "ME1" e a segunda "[if | ME2]" (ou seja, uma lista encabeçada por "if", cujo resto é "ME2"), tal que "ME1" e "ME2" sejam de tipo "t" e apresentem, respectivamente as estruturas "Structure1" e "Structure2". Assim, solicitada a análise da expressão "[c(0, [e, t]), c(0, e), if, c(0, [e, [e, t]]), c(1, e), c(2, e)]", através da cláusula "me([c(0, [e, t]), c(0, e), if, c(0, [e, [e, t]]), c(1, e), c(2, e)]).", o programa responde: "Type: t" e "Structure: [[c(0, [e, t]), c(0, e)], if, [[c(0, [e, [e, t]]), c(1, e)], c(2, e)]]".

Finalmente, as regras de quantificação universal (C.8) e existencial (C.9) são implementadas pelas cláusulas "me([all, Var | ME], t, [all, Var, Structure]) :- var(Var, Type), me(ME, t, Structure)." e "me([some, Var | ME], t, [some, Var, Structure]) :- var(Var, Type), me(ME, t, Structure).". Essas cláusulas vão exigir que as expressões quantificadas sejam constituídas por uma expressão sincategoremática ("all" ou "some"), uma variável de qualquer tipo ("var(Var, Type)") e uma expressão de tipo "t" (me(ME, t, Structure)"); a estrutura é constituída, como já deve ter ficado evidente, como "[some, Var, ME]" ou "[all, Var, ME]". Para os exemplos, "[some, v(0, e), c(0, [e, t]), v(0, e)]" e "[all, v(0, [e, [e, t]]), v(0, [e, [e, t]]), c(1, e), c(2, e)]", obtemos como respostas: 1) "Type: t" e "Structure: [some, v(0, e), [c(0, [e, t]), v(0, e)]]", e 2) "Type: t" e "Structure: [all, v(0, [e, [e, t]]), [[v(0, [e, [e, t]]), c(1, e)], c(2, e)]]".

Para encerrar, resta ainda mostrar como esse Programa 18 lida com as expressões ambíguas que sua implementação permite. Usando a expressão "[not, c(0, [e, t]), c(0, e), and, c(0, [e, [e, t]]), c(1, e), c(2, e)]" (que, como já dissemos, pode ser analisada com o escopo estreito da negação, como no Quadro 15, ou com o seu escopo largo, como no Quadro 14), o programa vai responder:

1. Type: t
   Structure:
   [[not, [c(0, [e, t]), c(0, e)]], and, [[c(0, [e, [e, t]]), c(1, e)], c(2, e)]]
2. Type: t
   Structure:
   [not, [[c(0, [e, t]), c(0, e)], and, [[c(0, [e, [e, t]]), c(1, e)], c(2, e)]]]

A primeira resposta, relativa ao escopo estreito da negação, se deve ao fato de que a cláusula que define a conjunção aparece antes da que define a negação; assim, primeiro o analisador divide a expressão em duas partes, o que está antes e o que vem depois do "and", testando se cada uma dessas duas partes é do tipo "t". Depois de atingir uma resposta afirmativa, o analisador pode tentar uma segunda resposta, testando se depois de "not" há uma expressão também do tipo "t"; de forma que se chega à segunda solução, onde aparece o escopo estreito da negação.[97]

Um último exemplo, agora envolvendo quantificação, seria o de "[all, v(0, e), c(0, [e, t]), v(0, e), if, c(0, [e, [e, t]]), c(1, e), v(0, e)]", para o qual o analisador encontraria:

1. Type: t
   Structure:
   [[all, v(0, e), [c(0, [e, t]), v(0, e)]], and, [[c(0, [e, [e, t]]), c(1, e)], v(0, e)]]
2. Type: t
   Structure:
   [all, v(0, e), [[c(0, [e, t]), v(0, e)], and, [[c(0, [e, [e, t]]), c(1, e)], v(0, e)]]]

Como se pode ver, apenas na segunda análise é que todas as instâncias da variável "v(0, e)" estão no escopo do quantificador; na primeira, apenas a primeira ocorrência dessa variável é que está neste escopo, que se encerra antes do "and".

E aqui termina a apresentação do analisador sintático de L_{type}.

4.4.2. Semântica de L_{type}

A semântica de L_{type} ainda se parece muito com a de L_1. As constantes não-lógicas continuam sendo semanticamente avaliadas através da função *F*, pela atribuição de valores (entidades não-lingüísticas) a estas constantes; as variáveis de L_{type} também são avaliadas

---

[97] O resultado poderia ser invertido caso fosse alterada a ordem das regras: se a da negação estivesse antes da conjunção, primeiro encontraríamos uma resposta com o escopo largo da negação e só depois é que poderíamos chegar à do escopo estreito. Como alguns resultados de experimentos psicolingüísticos parecem apontar para preferência de estruturas do primeiro tipo, preferimos adotar a ordem que apresenta primeiro o escopo estreito.

através da mesma AAI *g*, que atribui arbitrariamente valores apropriados a cada uma das variáveis. (A única diferença, como veremos abaixo, é o papel que o conceito de denotação possível passa a desempenhar nessas atribuições.) Essa atribuição de valores às expressões básicas é o que as regras 1.a e 1.b, abaixo, fazem.

1. Se $\alpha \in BE_a$, então:

   a. Se $\alpha \in Con_a$, então $[\![\alpha]\!]^{M,g} = F(\alpha)$.

   b. Se $\alpha \in Var_a$, então $[\![\alpha]\!]^{M,g} = g(\alpha)$.

2. Se $\alpha \in ME_{\langle a,b \rangle}$ e $\beta \in ME_a$, então $[\![\alpha(\beta)]\!]^{M,g} = [\![\alpha]\!]^{M,g}([\![\beta]\!]^{M,g})$.

3. – 7. Se ϕ e $\psi \in ME_t$, então:

   3. $[\![\neg\phi]\!]^{M,g} = 1$ se $[\![\phi]\!]^{M,g} = 0$, senão $[\![\neg\phi]\!]^{M,g} = 0$;

   4. $[\![\phi \wedge \psi]\!]^{M,g} = 1$ se $[\![\phi]\!]^{M,g} = 1$ e $[\![\psi]\!]^{M,g} = 1$, senão $[\![\phi \wedge \psi]\!]^{M,g} = 0$;

   5. $[\![\phi \vee \psi]\!]^{M,g} = 1$ se $[\![\phi]\!]^{M,g} = 1$ ou $[\![\psi]\!]^{M,g} = 1$, senão $[\![\phi \vee \psi]\!]^{M,g} = 0$;

   6. $[\![\phi \rightarrow \psi]\!]^{M,g} = 1$ se $[\![\phi]\!]^{M,g} = 0$ ou $[\![\psi]\!]^{M,g} = 1$, senão $[\![\phi \rightarrow \psi]\!]^{M,g} = 0$;

   7. $[\![\phi \leftrightarrow \psi]\!]^{M,g} = 1$ se ou $[\![\phi]\!]^{M,g} = 1$ e $[\![\psi]\!]^{M,g} = 1$, ou $[\![\phi]\!]^{M,g} = 0$ e $[\![\psi]\!]^{M,g} = 0$, senão $[\![\phi \leftrightarrow \psi]\!]^{M,g} = 0$.

8. Se $\phi \in ME_t$ e $u \in Var_a$, então $[\![\forall u\phi]\!]^{M,g} = 1$ se e apenas se para todo $e \in D_a$, $[\![\phi]\!]^{M,g[u/e]} = 1$.

9. Se $\phi \in ME_t$ e $u \in Var_a$, então $[\![\exists u\phi]\!]^{M,g} = 1$ se e apenas se para algum $e \in D_a$, $[\![\phi]\!]^{M,g[u/e]} = 1$.

Uma diferença maior pode ser observada na ausência das regras de predicação, substituídas (como já dissemos na seção anterior) por uma única regra única de cancelamento de tipos. Lembramos ainda que essa regra de cancelamento tem ainda a vantagem de poder ser generalizada a todos os tipos de modificadores. A semântica dessa regra de cancelamento de tipos é obtida pela chamada aplicação funcional (regra 2): a

denotação de uma expressão de tipo "⟨x, y⟩" é uma função que toma a denotação de uma expressão de tipo "x", resultando numa denotação da mesma espécie das expressões de tipo "y". Isso é garantido pela definição recursiva em 10-12, abaixo, das denotações possíveis (DWP: 84). (Na notação usada abaixo, D_x é o conjunto das denotações possíveis para qualquer expressão de tipo "x"; lembramos ainda que o conjunto A é o domínio do modelo, ou universo do discurso, no qual a língua estiver sendo avaliada.)

As possíveis denotações para cada tipo são:

10. $D_e = A$;

11. $D_t = \{1, 0\}$;

12. para quaisquer categorias sintáticas a e b, $D_{\langle a, b\rangle} = D_b^{D_a}$ (ou seja, o conjunto de todas as funções de D_a para D_b).[98]

Com isso, a denotação de uma expressão como "$c_{0, \langle e, t\rangle}$" ($D_{\langle e, t\rangle} = D_t^{D_e}$, pela regra 12), por exemplo, vai pertencer ao conjunto de todas as funções do domínio ($D_e = A$, pela regra 10) para valores de verdade ($D_t = \{1, 0\}$, pela regra 11); ou seja, a denotação de "$c_{0, \langle e, t\rangle}$" é alguma dentre todas as funções que, tomando alguma entidade do domínio como argumento, resulta ou no valor verdadeiro ou no valor falso (na notação usada, isso se representa do seguinte modo: $\{1, 0\}^A$).[99] Esse tipo de notação para as denotações possíveis, no entanto, vai ficando cada vez mais difícil de se acompanhar à medida que a complexidade dos tipos envolvidos aumenta. Assim, uma constante do tipo "⟨e, ⟨e, t⟩⟩" (o tipo que vai corresponder aos verbos transitivos diretos do português, por exemplo) vai apresentar $(D_t^{D_e})^{D_e}$ como possível denotação (ou ainda: $(\{1, 0\}^A)^A$); ou seja, uma função do domínio para uma função do domínio para valores de verdade. Os modificadores de predicados de um lugar, de tipo "⟨⟨e, t⟩, ⟨e, t⟩⟩", por sua vez, vão apresentar como denotação possível $(D_t^{D_e})^{D_t^{D_e}}$; ou seja, uma função de uma função do domínio para valores de verdade

---

[98] A locução "função de x para y" deve ser entendida como 'função que toma x como argumento, resultando em y'.

[99] Essa função que divide o domínio exatamente em duas parte (aqueles para os quais a função é verdadeira e os outros para os quais ela é falsa) é chamada de "função característica".

para uma função do domínio para valores de verdade.[100] Como é possível perceber, fica praticamente impossível processar lingüisticamente tanto encaixamento. E se isso ainda não foi suficiente, o leitor é convidado a tentar lidar com a denotação possível dos modificadores de predicados de dois lugares, cujo tipo é "⟨⟨e, ⟨e, t⟩⟩, ⟨e, ⟨e, t⟩⟩⟩". Isso, por si só, já parece justificar a necessidade do auxílio de algum processamento automatizado. Por outro lado, apesar da complexidade, a construção dessas denotações é completamente regular e bastante decomponível: tendo visto que a denotação de uma expressão "⟨e, t⟩"[101] é uma função do domínio para valores de verdade, fica fácil entender a denotação das expressões "⟨e, ⟨e, t⟩⟩" como $D_{\langle e,t\rangle}^{D_e}$ (que é o mesmo que $(D_t^{D_e})^{D^e}$); da mesma maneira, a denotação das expressões "⟨⟨e, t⟩, ⟨e, t⟩⟩" é $D_{\langle e,t\rangle}^{D_{\langle e,t\rangle}}$.

Apesar de cumprir um papel importante na determinação dos valores semânticos das expressões, os autores (DWP: 84) observam que "esta definição não pretende dizer *como* as denotações das expressões das várias categorias são determinadas, e nós até já vimos que elas são determinadas de modos diferentes. Uma expressão sintaticamente básica, como por exemplo um predicado de dois lugares ou um termo, terá sua denotação atribuída através da função F no modelo (se ela for uma constante) ou através da atribuição de valores g (se ela for uma variável)." Com isso, a definição das denotações possíveis em 10, 11 e 12 "deve então ser concebida como a determinação de um princípio que o modelo, a atribuição de valores e as regras semânticas precisam todos 'conspirar' para preservar" (DWP: 85). Dessa maneira, para preservar essa espécie de 'princípio estruturador' da interpretação, a função F só pode atribuir às constantes valores do tipo apropriado (ou seja, uma constante do tipo "e" só pode estar relacionada a alguma entidade do domínio, enquanto a uma constante do tipo "⟨e, t⟩" só pode ser atribuída uma função do domínio para valores de verdade); ainda por esse mesmo princípio, a AAI g não pode atribuir a uma variável de tipo "⟨e, t⟩" alguma entidade do domínio, mas sim alguma função do domínio para valore de verdade; e, finalmente, uma regra semântica não pode combinar uma função com qualquer argumento, mas apenas com o tipo de argumento determinado pela função

---

[100] Se eu não perdi nada...

[101] Estou tomando aqui a liberdade para encurtar as locuções "expressão de tipo..." por "expressões...".

(por exemplo, uma função do domínio para valores de verdade, quando toma como argumento uma entidade do domínio, só pode resultar em um valor de verdade).

De volta ao algoritmo de interpretação semântica, as regras semânticas 3 a 7 são exatamente iguais às de L_1. As regras de quantificação 8 e 9, por outro lado, como agora atuam sobre qualquer tipo de variável, precisam fazer menção a esse recurso; assim, o valor de verdade das sentenças com quantificadores vai ser calculado em relação ao conjunto de denotações possíveis para a variável quantificada: a quantificação existencial vai exigir que pelo menos um elemento do conjunto das denotações possíveis para o tipo da variável satisfaça a expressão quantificada, enquanto que a quantificação universal obriga que todos os elementos desse conjunto satisfaçam a expressão que está sendo quantificada. Mas isso também não é, em essência, muito diferente da quantificação em L_1: a única diferença é a menção às denotações possíveis, onde antes se falava apenas em universo do discurso.

Dessa forma, como tem acontecido, a nova língua L_{type} contém L_1. E com isso, todas as expressões de L_1 também podem ser analisadas e interpretadas pela gramática de L_{type}, contanto que os elementos lexicais daquela também sejam incluídos nesta. Para facilitar o desenvolvimento da apresentação, inclusive, DWP: 90-91 sugerem que se continue abreviando por "j", "d", "m" e "n" as primeiras quatro constantes do tipo "e" ("$c_{0, e}$", "$c_{1, e}$", "$c_{2, e}$" e "$c_{3, e}$"), por "M" e "B" as duas primeiras constantes do tipo "⟨e, t⟩" ("$c_{0, \langle e, t\rangle}$" e "$c_{1, \langle e, t\rangle}$"), por "K" e "L" as duas primeiras constantes do tipo "⟨e, ⟨e, t⟩⟩" ("$c_{0, \langle e, \langle e, t\rangle\rangle}$" e "$c_{1, \langle e, \langle e, t\rangle\rangle}$") e as primeiras três variáveis do tipo "e" por "x", "y" e "z" ("$v_{0, e}$", "$v_{1, e}$" e "$v_{2, e}$").

Assim, uma expressão de segunda ordem como "$\forall v_{0, \langle e, t\rangle}[v_{0, \langle e, t\rangle}(j) \rightarrow v_{0, \langle e, t\rangle}(d)]$" pode ser parafraseada por 'para cada uma das denotações possíveis de predicados de um lugar, no modelo considerado, se ela é verdadeira para *j*, ela também é verdadeira para *d*'. Da mesma forma, uma expressão de terceira ordem como "$\exists v_{0, \langle\langle e, t\rangle, t\rangle}[v_{0, \langle\langle e, t\rangle, t\rangle}(B) \leftrightarrow v_{0, \langle\langle e, t\rangle, t\rangle}(M)]$" será entendida como 'para alguma das denotações possíveis do tipo "⟨⟨e, t⟩, t⟩" (o tipo das expressões que tomam um predicado de um lugar e resultam numa sentença; ou seja, exatamente o tipo que será atribuído aos sujeitos das línguas naturais), se ela é satisfeita pelo predicado *B*, ela também é satisfeita pelo predicado *M*, e vice-versa'. Em

termos esquemáticos, essas duas análises podem ser vistas nos Quadro 16 e Quadro 17, respectivamente.

1.	$[\![\forall v_{0,\langle e,t\rangle}[v_{0,\langle e,t\rangle}(j) \rightarrow v_{0,\langle e,t\rangle}(d)]]\!]^{M,g} = 1$ se, para todo g', $[\![[v_{0,\langle e,t\rangle}(j) \rightarrow v_{0,\langle e,t\rangle}(d)]]\!]^{M,g'} = 1$	pela regra 8
2.	$[\![[v_{0,\langle e,t\rangle}(j) \rightarrow v_{0,\langle e,t\rangle}(d)]]\!]^{M,g'} = 1$ se $[\![v_{0,\langle e,t\rangle}(j)]\!]^{M,g'} = 0$ ou $[\![v_{0,\langle e,t\rangle}(d)]\!]^{M,g'} = 1$	de 1, pela regra 6
3.	$[\![v_{0,\langle e,t\rangle}(j)]\!]^{M,g'} = [\![v_{0,\langle e,t\rangle}]\!]^{M,g'}([\![j]\!]^{M,g'})$	de 2, pela regra 2
4.	$[\![v_{0,\langle e,t\rangle}(d)]\!]^{M,g'} = [\![v_{0,\langle e,t\rangle}]\!]^{M,g'}([\![d]\!]^{M,g'})$	de 2, pela regra 2

Quadro 16 - Esquema de interpretação de "$\forall v_{0,\langle e,t\rangle}[v_{0,\langle e,t\rangle}(j) \rightarrow v_{0,\langle e,t\rangle}(d)]$"

1.	$[\![\exists v_{0,\langle\langle e,t\rangle,t\rangle}[v_{0,\langle\langle e,t\rangle,t\rangle}(B) \leftrightarrow v_{0,\langle\langle e,t\rangle,t\rangle}(M)]]\!]^{M,g} = 1$ se, para algum g', $[\![[v_{0,\langle\langle e,t\rangle,t\rangle}(B) \leftrightarrow v_{0,\langle\langle e,t\rangle,t\rangle}(M)]]\!]^{M,g'} = 1$	pela regra 9
2.	$[\![[v_{0,\langle\langle e,t\rangle,t\rangle}(B) \leftrightarrow v_{0,\langle\langle e,t\rangle,t\rangle}(M)]]\!]^{M,g'} = 1$ se $[\![v_{0,\langle\langle e,t\rangle,t\rangle}(B)]\!]^{M,g'} = [\![v_{0,\langle\langle e,t\rangle,t\rangle}(M)]\!]^{M,g'}$	de 1, pela regra 7
3.	$[\![v_{0,\langle\langle e,t\rangle,t\rangle}(B)]\!]^{M,g'} = [\![v_{0,\langle\langle e,t\rangle,t\rangle}]\!]^{M,g'}([\![B]\!]^{M,g'})$	de 2, pela regra 2
4.	$[\![v_{0,\langle\langle e,t\rangle,t\rangle}(M)]\!]^{M,g'} = [\![v_{0,\langle\langle e,t\rangle,t\rangle}]\!]^{M,g'}([\![M]\!]^{M,g'})$	de 2, pela regra 2

Quadro 17 – Esquema de interpretação de "$\exists v_{0,\langle\langle e,t\rangle,t\rangle,t\rangle}[v_{0,\langle\langle e,t\rangle,t\rangle,t\rangle}(B) \leftrightarrow v_{0,\langle\langle e,t\rangle,t\rangle,t\rangle}(M)]$"

No entanto, essa definição das regras para a quantificação parece defeituosa; não apenas em relação a uma implementação computacional, mas incoerente em si mesma. É o que se pretende demonstrar a seguir.

Tome-se como exemplo a expressão "$\exists v_{0,\langle e,t\rangle}\, v_{0,\langle e,t\rangle}(j)$",[102] com quantificação de segunda ordem, que em termos semânticos pode ser parafraseada por 'para alguma propriedade definida no modelo, ela é verdadeira quando é atribuída à entidade "a" (designada na língua L_{type} pela constante "j")'. No modelo considerado (o mesmo usado para interpretar L_1, em DWP: 61), intuitivamente, sabemos que essa expressão é verdadeira por causa do predicado "M", que denota uma propriedade verdadeira para todas as entidades do modelo, mas não por "B", que denota uma propriedade que só é verdadeira para as entidades "b" e "c".

Mas vejamos como a expressão é interpretada segundo as regras propostas para isso. Para interpretar uma expressão quantificada existencialmente, segundo a regra 9, devemos encontrar uma entidade *e* dentre as denotações possíveis para a variável "$v_{0,\langle e,t\rangle}$", de tipo "⟨e, t⟩" (ou seja, na notação usada, $e \in D_{\langle e,t\rangle}$), que torna verdadeira a expressão "$v_{0,\langle e,t\rangle}(j)$", quando atribuímos a entidade *e* à variável "$v_{0,\langle e,t\rangle}$" (ou seja, $\left\|v_{0,\langle e,t\rangle}(j)\right\|^{M,g'} = 1$, a partir da AAI *g*, de forma que $g' = g[v_{0,\langle e,t\rangle}/e]$). Caso encontremos essa entidade, a expressão é verdadeira; caso contrário, é falsa.

Sendo assim, vejamos qual é o conjunto $D_{\langle e,t\rangle}$, das denotações possíveis para o tipo "⟨e, t⟩". Como no modelo proposto as entidades são apenas "a", "b" e "c" e os valores de verdade são apenas o verdadeiro e o falso (representados por "1" e "0", respectivamente), o conjunto de todas as funções dessas entidades para valores de verdade ($D_t^{D_e}$) é o seguinte:

$$D_{\langle e,t\rangle} = \left\{ \begin{bmatrix} a \to 1 \\ b \to 1 \\ c \to 1 \end{bmatrix}, \begin{bmatrix} a \to 1 \\ b \to 1 \\ c \to 0 \end{bmatrix}, \begin{bmatrix} a \to 1 \\ b \to 0 \\ c \to 1 \end{bmatrix}, \begin{bmatrix} a \to 1 \\ b \to 0 \\ c \to 0 \end{bmatrix}, \begin{bmatrix} a \to 0 \\ b \to 1 \\ c \to 1 \end{bmatrix}, \begin{bmatrix} a \to 0 \\ b \to 1 \\ c \to 0 \end{bmatrix}, \begin{bmatrix} a \to 0 \\ b \to 0 \\ c \to 1 \end{bmatrix}, \begin{bmatrix} a \to 0 \\ b \to 0 \\ c \to 0 \end{bmatrix} \right\}$$

Para avaliarmos a verdade da expressão "$v_{0,\langle e,t\rangle}(j)$", recorremos à regra 2, do cancelamento de tipos, já que "$v_{0,\langle e,t\rangle}$" é uma variável de tipo "⟨e, t⟩" e "j" uma constante de tipo "e", e a combinação dessas duas expressões resulta numa expressão de tipo "t". Segundo essa regra, o valor semântico da expressão será obtido através da aplicação do valor semântico relativo à variável "$v_{0,\langle e,t\rangle}$" ao valor semântico relativo à constante "j".

---

[102] Sem as abreviações sugeridas, essa expressão corresponderia a "$\exists v_{0,\langle e,t\rangle}\, v_{0,\langle e,t\rangle}(c_{0,e})$".

Como, pelo modelo, o valor semântico de "j" é entidade "a" ($F(j) = a$), encontramos em $D_{\langle e, t\rangle}$ quatro funções que, quando aplicadas a "a", resultam na verdade: qualquer uma das quatro primeiras funções de $D_{\langle e, t\rangle}$, quando aplicadas a "a", resultam no valor verdadeiro.

Como encontramos quatro valores para $e \in D_{\langle e, t\rangle}$ que tornam a expressão "$v_{0, \langle e, t\rangle}(j)$" verdadeira, conseqüentemente, a expressão "$\exists v_{0, \langle e, t\rangle}\ v_{0, \langle e, t\rangle}(j)$", para a qual apenas um bastava, também é verdadeira. No entanto, o que é mais importante é que isso foi feito sem se tomar conhecimento das propriedades estabelecidas pelo modelo. O resultado disso é que, através das regras propostas, qualquer quantificação existencial (excetuada a de primeira ordem) será sempre trivialmente verdadeira, independentemente do modelo no qual esteja sendo interpretada.

Mas o defeito não está apenas na regra da quantificação existencial, a da quantificação universal também não funciona: seguindo as regras, nunca construiríamos com ela (excetuando novamente a quantificação de primeira ordem) uma sentença verdadeira em L_{type}. Tomemos como exemplo agora a expressão "$\forall v_{1, \langle e, t\rangle}\ v_{1, \langle e, t\rangle}(d)$", que poderia ser parafraseado por 'para toda propriedade definda no modelo, ela é verdadeira quando atribuída à entidade "b" (designada pela constante "d")'. Mais uma vez, essa sentença seria intuitivamente verdadeira porque as duas propriedades (designadas por "M" e "B") definidas pelo modelo são verdadeiras quando aplicadas à entidade "b" ($F(M) = \{a, b, c\}$ e $F(B) = \{b, c\}$).[103]

Vejamos como a expressão é avaliada quando seguimos as regras. Segundo a regra 8, para interpretação da quantificação universal, a expressão "$\forall v_{1, \langle e, t\rangle}\ v_{1, \langle e, t\rangle}(d)$" será verdadeira se para todo $e \in D_{\langle e, t\rangle}$, a expressão "$v_{1, \langle e, t\rangle}(d)$" for verdadeira quando atribuímos à variável "$v_{1, \langle e, t\rangle}$" cada um desses valores de e. Ou seja, "$\forall v_{1, \langle e, t\rangle}\ v_{1, \langle e, t\rangle}(d)$" será verdadeira se "$v_{1, \langle e, t\rangle}(d)$" for verdadeira para todas as funções de $D_{\langle e, t\rangle}$ (que já vimos

---

[103] No texto, estamos empregando a mesma notação de conjunto usada na interpretação de L_1; representadas com a notação empregada na interpretação de L_{type}, elas corresponderiam a:

$$F(M) = \begin{bmatrix} a \rightarrow 1 \\ b \rightarrow 1 \\ c \rightarrow 1 \end{bmatrix} \text{ e } F(B) = \begin{bmatrix} a \rightarrow 0 \\ b \rightarrow 1 \\ c \rightarrow 1 \end{bmatrix}$$

acima). Não é difícil ver que o conjunto $D_{\langle e, t\rangle}$ de denotações possíveis para a variável "$v_{1, \langle e, t\rangle}$", sempre vai conter funções que atribuem o valor verdadeiro e também o falso para o mesmo indivíduo; para o modelo proposto, por exemplo, para o indivíduo "b", que é a denotação da constante "d", vamos ter algumas funções que resultam em "0" (no conjunto de $D_{\langle e, t\rangle}$ acima, a terceira, a quarta, a sétima e a oitava) e outras que resultam em "1" (no mesmo conjunto, a primeira, a segunda, a quinta e a sexta).

No entanto, essa deficiência não afeta a quantificação de primeira ordem. Como o conjunto de denotações possíveis para as variáveis de tipo "e" coincide com o domínio, que é estabelecido pelo conjunto "A" no modelo, as sentenças quantificadas universal e existencialmente podem ser adequadamente avaliadas pelo mesmo procedimento acima, que não funciona para quantificações de ordem superior à primeira.

Tome-se como exemplo uma expressão como "$\exists$x B(x)".[104] O que essa expressão nos diz é que para alguma atribuição *g'*, semelhante a *g* excluindo talvez a atribuição de valor à variável "x", a sentença "B(x)" é verdadeira segundo o modelo considerado e essa atribuição *g'*. Intuitivamente (talvez recorrendo a um recurso substituicional de avaliação), sabemos que a expressão "$\exists$x B(x)" é verdadeira para o modelo M, apresentado anteriormente, já que nele o predicado "B" é verdadeiro quando toma como argumento as expressões "d" e "n", relativas às entidades "b" e "c". Porém a avaliação semântica apresentada pelos autores não é a substituicional, e sim a atribuicional;[105] além disso, e até independentemente dessa distinção, a sentença acima seria trivialmente verdadeira, mesmo num modelo em que não fosse atribuído nenhum valor ao predicado "B" (ou, alternativamente, quando não houvesse nomes para as entidades que são verdadeiras para o predicado "B").

---

[104] Essa expressão é equivalente a "$\exists v_{0, e}\, c_{1, \langle e, t\rangle}(v_{0, e})$".

[105] Essa distinção entre abordagem substituicional e atribuicional é feita, por exemplo, em Carpenter (1997: 46), que diz que "substituições mapeiam variáveis em termos, enquanto que atribuições mapeiam variáveis em objetos do domínio". Ela já aparecia também em Chierchia & McConnell-Ginet (2000: 114), onde a abordagem substituicional "enfatiza a relação das sentenças com nomes próprios no lugar das expressões quantificacionais", enquanto na abordagem objetual (correspondente à atribuicional) há um "desenvolvimento da relação entre sentenças e pronomes".

Comecemos: o valor semântico de "$\exists x$ B(x)" é calculado pela regra 9, para a quantificação existencial, que diz que o valor semântico dessa expressão será 1 caso haja algum elemento *e* do conjunto das denotações possíveis para o tipo da variável "x" (como a variável "x" é do tipo "e", isso é representado por $e \in D_e$). Se atribuírmos o valor semântico "a" à variável "x", a sentença "B(x)" seria falsa, já que a denotação do predicado "B" no modelo proposto resulta em 0 quando aplicado a "a"; mas quando atribuímos à variável o valor "b", a sentença resulta verdadeira, porque sua aplicação a esse valor leva a 1; a atribuição do valor "c" à variável daria o mesmo resultado.

Para a quantificação universal, a avaliação semântica também é eficaz. Uma sentença como "$\forall y$ M(y)"[106] também seria verdadeira segundo o mesmo modelo, já que o predicado "M" denota uma função que sempre resulta verdadeira para todos os indivíduos do modelo. E é exatamente isso que se consegue seguindo as regras: pela regra 8, da quantificação universal, sabemos que a sentença acima será verdadeira caso todas as entidades $e \in D_e$, quando atribuídas à variável "y", resultem na verdade de "M(y)" segundo essas atribuições.

Como foi possível demonstrar, apesar de ainda continuar valendo para a quantificação de primeira ordem, a semântica de L_{type} não consegue, ao contrário do que os autores afirmam, dar conta da quantificação de ordens superiores à primeira. Mesmo sendo um manual amplamente utilizado para introdução e difusão da semântica formal, e principalmente da semântica montagoveana, a definição da semântica de L_{type} não está suficiente formalizada, pois as entidades de ordem superior à primeira não estão devidamente formalizadas para permitir o tratamento da quantificação das variáveis relativas a estes tipos. Essa deficiência foi revelada exclusivamente pela tentativa de implementação computacional do interpretador semântico da língua: como o programa parecia estar estritamente executando aquilo que as regras exigiam, ele deveria funcionar como era de se esperar. Contudo, o programa nunca chegava aos resultados devidos; e por mais que se revisse o algoritmo para que ele fosse o mais fiel possível às regras, ele continuava sem funcionar. Depois de várias tentativas de revisão do programa, revendo as

---

[106] Essa expressão é equivalente a "$\forall v_{1,e}\, c_{0,\langle e,t\rangle}(v_{1,e})$".

regras mais detidamente, ainda tentando encontrar alguma deficiência no algoritmo do programa que não estivesse de acordo com a definição das regras, foi possível perceber que o defeito estava na diferença de tratamento entre a denotação das variáveis de primeira ordem e as de outras ordens: de alguma maneira, ao determinar o conjunto "A", do universo do discurso, o modelo está definindo explicitamente o domínio pertinente de entidades a serem atribuídas às variáveis de primeira ordem; isso não acontece com as variáveis de ordem superior. Não há no modelo proposto nenhuma definição desse tipo para essas variáveis;[107] elas dependem exclusivamente de uma especificação que gera absolutamente todo tipo de denotação possível para aquela variável, sem especificar apenas aquelas que são pertinentes à atribuição para a variável.

Mas o que é mais impressionante é que isso parece ter passado desapercebido por todos os outros autores que trabalharam nessa perspectiva montagoveana, mesmo aqueles que supostamente estavam trabalhando dentro de uma abordagem computacional, como Carpenter 1997. Em outros autores, como Cann 1993, poderíamos atribuir essa distração à exclusiva atenção à quantificação de indivíduos: apesar de introduzir uma língua de quantificação ilimitada, o autor se restringe à quantificação do sintagma nominal, que versa exclusivamente sobre a quantificação de indivíduos. Uma desculpa parecida poderia ser oferecida também para o livro de Carpenter mencionado acima, já que nele o tratamento escolhido é o da lógica combinatória, que dispensa o aparato das variáveis.

4.4.3. Ajustando a semântica de L_{type}

Como acabamos de demonstrar, há uma pequena impropriedade na formalização da semântica de L_{type} apresentada por DWP. Mas antes de sugerir qualquer alternativa para corrigir esse defeito, vamos observar a relação entre algumas expressões e os

---

[107] Para ser mais preciso, nem chega a haver a determinação explícita de um modelo. Um pouco pela semelhança com L_1, e um pouco pela falta dessa determinação, parece ser razoável usar o modelo empregado para L_1. De qualquer forma, os autores nem chegam a ilustrar o funcionamento da semântica de L_{type} através da aplicação das regras a modelos e a atribuições para variáveis, eles se restringem a explicar o funcionamento de alguns operadores de ordem superior indiferentemente a modelos e atribuições para variáveis.

modelos nos quais eles são avaliados. Essa observação ainda será feita de um ponto de vista substituicional, para que sejam evidenciadas algumas questões importantes dessa relação entre expressões e modelos.

Sendo assim, que tipo de modelo nós precisaríamos para que "∃X[X(j)]" fosse verdadeira? Desse ponto de vista substituicional, seria preciso que algum dentre os predicados de um lugar (de tipo "⟨e, t⟩") resultasse verdadeiro quando aplicado a "j". Assim, para um modelo composto da seguinte maneira:[108]

A = {a, b, c}

$[\![j]\!]$ = a

$[\![d]\!]$ = b

$[\![n]\!]$ = c

$[\![m]\!]$ = a

$[\![M]\!]$ = {a, b, c}

$[\![B]\!]$ = {b, c}

A sentença "∃X[X(j)]" seria verdadeira porque, dos dois predicados especificados no modelo ("B" e "M"), um deles ("M") é verdadeiro quando aplicado a "j", já que a denotação de "j" ('a') faz parte do conjunto denotado por "M" ('{a, b, c}').

Já para o modelo abaixo, a mesma sentença "∃X[X(j)]" seria falsa porque nenhum dos dois predicados especificados, quando aplicados a "j", resulta na verdade: a denotação de "j" ('a') não faz parte nem do conjunto denotado por "M" ('{b, c}'), nem por "B" ('{b, c}').

A = {a, b, c}

$[\![j]\!]$ = a

---

[108] Vou novamente tomar a liberdade de não indicar exaustivamente o modelo e a atribuição nos quais a interpretação está ocorrendo. Mas isso é apenas uma abreviação.

$[\![d]\!] = b$

$[\![n]\!] = c$

$[\![m]\!] = a$

$[\![M]\!] = \{b, c\}$

$[\![B]\!] = \{b, c\}$

Por outro lado, uma quantificação universal da mesma fórmula, acima quantificada existencialmente (ou seja, a sentença "$\forall X[X(j)]$"), não seria verdadeira para nenhum dos dois modelos anteriores, porque o quantificador universal exige que todos os predicados especificados no modelo resultassem na verdade quando aplicados a "j". No primeiro modelo, o indivíduo 'a', que é a denotação de "j", só pertence ao conjunto '{a, b, c}', denotado por "M", mas não ao conjunto denotado por "B" ('{b, c}'); portanto, a denotação de "j" não pertence a todos os conjuntos denotados pelos predicados de um lugar especificados no modelo. No segundo modelo, a denotação de "j" não pertence a nenhum dos conjuntos denotados pelos predicados "$\langle e, t\rangle$".

Mas o primeiro modelo poderia ser facilmente modificado para tornar a sentença "$\forall X[X(j)]$" verdadeira, com a inclusão de 'a' no conjunto denotado por "B":

$A = \{a, b, c\}$

$[\![j]\!] = a$

$[\![d]\!] = b$

$[\![n]\!] = c$

$[\![m]\!] = a$

$[\![M]\!] = \{a, b, c\}$

$[\![B]\!] = \{a, b, c\}$

Agora, para todos os predicados de tipo "⟨e, t⟩" especificados no modelo, eles sempre resultam na verdade quando aplicados a "j", porque 'a' (a denotação de "j") pertence a todos os conjuntos denotados pelos predicados de um lugar no modelo.

No entanto, como dissemos inicialmente, tudo até aqui foi apresentado exclusivamente do ponto de vista substituicional, no qual as variáveis são substituídas pelas expressões dos respectivos tipos. Mas como fica o cálculo denotacional feito atribuicionalmente?

Do ponto de vista atribuicional, uma sentença como "∃x[M(x)]" seria verdadeira se o conjunto de indivíduos denotado por "M" contivesse pelo menos um indivíduo; caso contrário, ela seria falsa. Dizendo de uma maneira mais adequada à perspectiva atribuicional, a sentença seria verdadeira se pudermos encontrar algum indivíduo que possamos atribuir como denotação à variável "x" de forma a tornar "M(x)" verdadeira; essa última fórmula, por sua vez vai ser verdadeira se aquele indivíduo atribuído à variável fizer parte do conjunto denotado por "M". A sentença "∀x[B(x)]" seria verdadeira se, quando cada indivíduo do domínio fosse atribuído enquanto denotação da variável, a fórmula "B(x)" sempre resultasse verdadeira; caso contrário, a sentença seria falsa. Como é possível perceber, o cálculo denotacional por atribuição fala apenas em indivíduos e conjuntos, que são as denotações dos termos e dos predicados de um lugar; todas as operações semânticas são feitas apenas com os valores denotacionais atribuídos às expressões, e não através da substituição de expressões no lugar das variáveis.

Acabamos de recordar a avaliação das sentenças com quantificação de primeira ordem (aquela na qual as variáveis só podem denotar indivíduos), mas como ficaria ainda desse mesmo ponto de vista atribuicional a quantificação de ordens superiores à primeira?

Antes de começar a falar desse cálculo, é importante lembrar ainda que a escolha do modo atribuicional, em detrimento do substituicional, é justificada porque não é necessário que todas as denotações estejam sempre associadas a alguma expressão. Para um modelo no qual nem todas as entidades tivessem nome, e através de uma avaliação exclusivamente substituicional, a principal vantagem da quantificação ficaria anulada: não teríamos como falar das entidades que não estiverem atribuídas a nenhuma

expressão. Num modelo desse tipo, no qual haja mais indivíduos do que nomes para eles, como abaixo, nós não teríamos como avaliar a sentença "∃x[B(x)]" pelo modo substituicional porque não encontraríamos um nome adequado para ocupar o lugar da variável, ainda que, nesse modelo, a sentença seja claramente verdadeira, já que a denotação do predicado "B" contém os indivíduos 'b' e 'c', ainda que não haja nome para nenhum dos dois.

$A = \{a, b, c\}$

$[\![j]\!] = a$

$[\![m]\!] = a$

$[\![M]\!] = \{a, b, c\}$

$[\![B]\!] = \{b, c\}$

Dessa maneira, fica clara a deficiência da avaliação substituicional em relação à atribuicional. E mesmo que a avaliação substituicional seja mais simples e intuitiva do que a atribuicional (o que ainda deve garantir alguma prioridade pelo menos didática a ela), essa deficiência a torna completamente inadequada enquanto método para o cálculo denotacional das línguas naturais (esse método ainda continuaria sendo válido para as línguas artificiais nas quais todas as entidades correspondessem a alguma expressão).

Assim, nesse modo atribuicional, o cálculo da denotação de uma sentença com quantificação de segunda ordem (na qual existem variáveis não só para indivíduos, mas também para conjuntos de indivíduos), como "∃X[X(j)]", por exemplo, precisa ser expresso da seguinte maneira: para que "∃X[X(j)]" seja verdadeira, é preciso que esteja especificado no modelo algum conjunto de indivíduos do qual o indivíduo denotado por "j" faça parte. Dessa forma, portanto, o que parece estar faltando na formulação de DWP é que os conjuntos denotados pelos predicados de um lugar também precisam estar especificados no modelo, incluídos no universo do discurso, para que suas variáveis correspondentes posam ser quantificadas; isso pelos mesmo motivos que, na quantificação de variáveis de indivíduos, é necessária a determinação dos indivíduos desse universo do discurso (ainda pelo mesmo motivo, também as relações precisarão

constar do universo do discurso; além disso, para as quantificações superiores às de segunda ordem, não só os indivíduos, as propriedades e as relações, mas também todos os tipos de qualificadores de propriedades e de relações, e todos os qualificadores de qualificadores imagináveis, precisam estar definidos no universo do discurso).

Por isso, um modelo mais adequado seria:

A = {a, b, c, {b, c}, {a, b, c}}

$[\![j]\!]$ = a

$[\![d]\!]$ = b

$[\![n]\!]$ = c

$[\![m]\!]$ = a

$[\![B]\!]$ = {b, c}

(Observe que, agora, um dos predicados é que não corresponde a nenhuma expressão.) Nesse novo modelo, apesar de não dispormos de uma expressão para denotar a propriedade correspondente ao conjunto "{a, b, c}", vamos poder avaliar como verdadeira uma sentença como "∃X[X(j)]": apesar de não haver nenhum predicado cuja denotação seja um conjunto que contenha o indivíduo 'a' (que é a denotação de "j"), a sentença vai ser verdadeira porque, no modelo, consta uma propriedade a cujo conjunto a denotação do nome "j" pertence.

Decorre dessa discussão que a função "g", da forma como foi definida em DWP (agindo sobre todo o conjunto de denotações possíveis para cada um dos tipos), é importante na construção dos modelos, regulamentando principalmente o que pode ou não constar no universo do discurso (só uma denotação possível pode constar do universo do discurso de um modelo), mas não parece cumprir sozinha nenhum papel relevante na avaliação das denotações das expressões. Para essa avaliação, apenas o próprio universo do discurso parece ser suficiente. Assim, ao invés de permitir a variação de todas as denotações possíveis para o tipo requerido, a função "g" deveria apenas exigir que os seus valores variassem dentro das denotações do requerido tipo que fizessem parte do universo do discurso. Essa parece ser uma formulação mais objetiva e econômica.

(No entanto, do ponto de vista exclusivamente lógico, desconsiderando aspectos de implementação material, poderíamos continuar com a mesma função "g" percorrendo todas as denotações possíveis para o tipo determinado, contanto que fosse incluída uma restrição que só tornasse pertinente as que fizessem parte do universo do discurso. Como vamos explorar aqui a solução anterior, não avançaremos nessa discussão.)

Portanto, as regras para a avaliação semântica das quantificações universal e existencial poderiam ser reexpressas, respectivamente, como 8 e 9.

8. Se $\phi \in ME_t$ e $u \in Var_a$, então $[\![\forall u\phi]\!]^{M,g} = 1$ se e apenas se para todo $e \in (A \cap D_a)$, $[\![\phi]\!]^{M,g[u/e]} = 1$.

9. Se $\phi \in ME_t$ e $u \in Var_a$, então $[\![\exists u\phi]\!]^{M,g} = 1$ se e apenas se para algum $e \in (A \cap D_a)$, $[\![\phi]\!]^{M,g[u/e]} = 1$.

Dessa forma, agora, a regra da quantificação universal exige que todas as denotações do tipo adequado em A (restrição expressa pela intersecção do universo do discurso A com o conjunto D_a de denotações possíveis para o tipo "a"), e não mais todas as denotações possíveis para aquele tipo, é que sejam relevantes para a avaliação da fórmula quantificada. A regra da quantificação existencial, por sua vez, vai exigir que, para apenas uma dessas denotações (uma das entidades na intersecção de A com D_a), a fórmula quantificada resulte verdadeira quando essa denotação é atribuída à variável "u".

Com essas pequenas modificações, nem a quantificação existencial seria sempre trivialmente verdadeira para qualquer modelo, nem a quantificação universal seria sempre trivialmente falsa para todos os modelos.

5. Conclusões e perspectivas

A conclusão mais importante desta tese é a descoberta da deficiência na formulação da semântica de L_{type}. O manual de DWP já foi amplamente utilizado na formação de tantos semanticistas, aparentemente sem que nenhum tivesse feito uma menção explícita a esse defeito. Pelo menos, até o momento nunca tinha encontrado nenhuma referência a qualquer erro neste manual. A solução, apesar de teoricamente mais importante, é secundária no sentido óbvio de que sem um problema não faz sentido se falar em solução.

Desse ponto de vista teórico, a solução apresentada aqui tem conseqüências técnicas bastante importantes: se ela é tecnicamente simples, bastando acrescentar às condições a exigência da intersecção das denotações possíveis com o domínio, ela é ontologicamente bastante complicada. Ao contrário do que a semântica apresentada naquele manual nos fazia supor, para a interpretação semântica das línguas de ordem superior à primeira, não podemos ter apenas indivíduos no domínio do seu modelo: ele precisará conter também relações (incluindo aí as propriedades, como relações unárias), para as línguas de segunda ordem, e todas as entidades correspondentes aos modificadores das outras ordens além da segunda. Para quantificar variáveis para relações, é preciso que estejam especificadas no domínio as relações que valem para aquele determinado modelo e que precisarão ser atribuídas a essas variáveis durante os processos de avaliação semântica; para quantificar variáveis de modificadores de expressões de relações, é preciso que as denotações desses modificadores estejam especificadas no domínio do modelo, de forma a serem atribuídos às variáveis na interpretação semântica; e assim por diante para cada uma das ordens de níveis superiores. Essas denotações não correspondem a nenhuma outra parte do modelo, nem podem ser exclusivamente construídas como o conjunto das denotações possíveis para o tipo da variável. Assim, a conclusão em relação à solução é de que o modelo (e, mais especificamente, o seu domínio) precisa ser tão complexo quanto a língua na interpretação da qual ele estiver participando.

Além dessas duas conclusões anteriores, que poderíamos chamar de diretas, essa tese ainda apresenta evidências indiretas para uma discussão epistemológica sobre como

uma implementação computacional, da forma como foi desenvolvida aqui, pode colaborar para o aprimoramento da própria teoria que a deveria sustentar.

Mas antes de começarmos a falar nessas conseqüências epistemológicas, é preciso fazer uma distinção entre implementações computacionais fortes e fracas, inspirada na distinção mencionada por Crocker (1996: 10-11) entre competência forte e fraca. Segundo essa distinção, um modelo de desempenho que apenas chega aos mesmos resultados determinados por um modelo de competência, sem percorrer os mesmos passos, é chamado de fraco. Um modelo de desempenho forte seria aquele no qual, além de se chegar aos mesmos resultados, são percorridas as mesmas etapas definidas pelo modelo de competência. Assim, também poderíamos falar de implementações computacionais fracas, que apenas atingem aos resultados esperados, e de implementações computacionais fortes, que obtém os resultados esperados exatamente da maneira definida pela teoria.

Quase todos os aplicativos de PLN estão mais preocupados com a eficiência dos resultados do que propriamente com a adequação teórica; assim, freqüentemente, esses aplicativos estão mais preocupados em obter os dados corretos, dando pouca atenção para como esses resultados são obtidos. É a esse tipo de coisa que estou chamando de implementação fraca. Nesse sentido, as implementações computacionais fracas manipulam representações que, por princípio, são de pouco interesse teórico.

A implementação computacional desenvolvida nesta tese, ao invés de se preocupar prioritariamente com aspectos relacionados à eficiência, procurou refletir fielmente a teoria empregada. Esta implementação computacional forte da semântica de Montague não precisava apenas apresentar as mesmas interpretações semânticas previstas pela teoria, mas ela o fazia seguindo exatamente as mesmas etapas determinadas pelas definições formais: se esse objetivo fosse atingido, qualquer montagoveano deveria reconhecer no próprio algoritmo em Prolog, e sem muito esforço (apenas o necessário para compreender a exígua sintaxe do Prolog), aquelas mesmas definições formais que ele está acostumado a manipular para chegar a suas análises.

Assim, a conseqüência indireta para essa discussão epistemológica é a de que, se uma teoria tem que apresentar explicitamente as explicações, sem depender de qualquer

capacidade obscura de quem a pratica, mais do que a simples formalização lógica, a implementação computacional forte pode ajudar a revelar algumas deficiências na elaboração da teoria. Nem sempre a exclusiva formalização garante a precisão suposta por esse tipo de abordagem; enquanto uma teoria, por mais formal que seja, não é colocada para produzir automaticamente os seus próprios resultados, podemos acabar sendo enganados, seja por nossa condescendência, seja por nossa preguiça em conferir todos os detalhes. Já para um computador, é preciso que todos os detalhes estejam minuciosa e completamente determinados, senão nunca conseguimos os resultados desejados.

O que aconteceu na implementação de L_{type} foi exatamente isso: na tentativa de construir um analisador gramatical para essa língua, de acordo com uma implementação computacional forte, como não se conseguia chegar aos resultados esperados, e como ainda não parecia haver nenhum problema de programação, só restou desconfiar da formalização apresentada em DWP, que se demonstrou efetivamente inadequada numa revisão ainda que apenas formal. No entanto, essa revisão formal já foi praticada com as desconfianças suscitadas pelas dificuldades apresentadas na tentativa de implementação computacional.

Terminado o relato sobre as conclusões, resta agora falar um pouco das expectativas futuras abertas por suas conseqüências.

A primeira, e mais evidente de todas, é a da continuação da implementação da própria língua L_{type}, que acabou ficando sem um analisador para ela, e de todas as outras línguas de DWP. Mesmo porque, apesar de todo o interesse nessa língua de ordem superior, o principal atrativo da semântica de Montague é o tratamento das línguas intensionais, já que as línguas naturais parecem empregar esse recurso.[109]

Outra perspectiva de pesquisa aberta por esta tese é a da construção de uma interface para auxiliar a elaboração e a manipulação desses analisadores gramaticais. Essas interfaces seriam diretamente inspiradas em programas já existentes como o

---

[109] Como sugere, por exemplo, o tratamento que Borges (1991) apresenta para os adjetivos.

Syntactica e o Semantica, desenvolvidos por Richard K. Larson e David S. Warren (1997 e 1996), e o Tarski, criado por Jon Barwise e John Etchmendy (1993a e 1993b).[110]

O Tarski é um programa para auxiliar o ensino de cálculo de predicado de primeira ordem, e é constituído basicamente por quatro janelas: 1) uma janela com o teclado, onde se encontram os principais símbolos normalmente empregados (como os quantificadores existencial e universal, os conectivos lógicos, seis letras para variáveis, mais seis letras para as constantes e os predicados com os quais o programa está preparado para lidar), 2) uma janela de sentenças, onde são escritas as expressões do cálculo, 3) uma janela para o mundo no qual as expressões serão avaliadas (consiste basicamente de um tabuleiro quadriculado em que se pode distribuir tetraedros, cubos e dodecaedros, com três tamanhos diferentes) e, finalmente, 4) uma janela de inspeção, na qual se pode auferir nossas opiniões sobre a estrutura sintática da expressão e sua avaliação semântica (se a expressão ativa na janela de sentenças é verdadeira ou não de acordo com o mundo selecionado).[111]

O Syntactica é um programa também composto por várias janelas que permitem basicamente a edição e a aplicação de uma gramática independente de contexto, junto com alguns tipos de transformações. Numa janela de regras, podemos editar as regras de estrutura sintagmática, das quais podemos indicar o núcleo. O programa dispõe também de uma janela para o registro das entradas lexicais na qual, além da informação da categoria do item, podemos indicar os seus complementos (marcando os obrigatórios) e informar alguns traços. Além da janela de sentença e da janela onde a estrutura sintática é construída, o programa ainda inclui uma janela de transformações, que permite fazer adjunções à direita e à esquerda, substituição, apagamento e indexação; as árvores construídas nessa janela podem ser armazenadas numa janela de árvores.

---

[110] Os mesmos autores do Tarski também desenvolveram um programa para ensino e construção de máquinas de Turing, mas até o presente momento não tive oportunidade de conhecer o programa (além disso, só está disponível para Macintosh, aos quais temos pouco acesso).

[111] Nessa janela de inspeção, pode-se jogar um jogo no qual o programa vai explicitando os passos com os quais precisamos nos comprometer caso nos comprometamos com a verdade ou a falsidade das expressões. Mas não entraremos nessa questão aqui; o leitor interessado pode obter mais informações nos próprios livros de Barwise e Etchmendy (1993a e 1993b).

Quanto ao Semantica, como não tive acesso direto a nenhuma versão do programa, não poderei comentar muita coisa. Tudo o que eu sei dele, por enquanto, é que ele permite fazer a avaliação semântica de sentenças, a partir de uma análise sintática construída com o Syntactica e de um mundo que pode ser construído dentro do próprio Semantica. A principal dificuldade em relação a esse dois últimos programas é que eles foram desenvolvidos para o NextStep, que é um sistema operacional completamente desconhecido no Brasil. Em relação ao Syntactica, eu ainda tenho acesso a uma versão beta para Windows conseguido há algum tempo na página de internet do laboratório do Larson (http://semlab2.sbs.sunysb.edu/), mas que não está mais disponível (mesmo assim, essa versão beta parece ter defeitos na parte dos traços lexicais que não foram resolvidos); já do Semantica, o máximo que eu consegui foi uma visão geral, que pode ser vista na mesma página de internet citada acima.

A idéia, então, seria a de investir algum tempo na aprendizagem de algum ambiente de desenvolvimento de interfaces, como o Delphi, de forma que fosse possível projetar um aplicativo como os três mencionados acima para manipular as definições formais da sintaxe e da semântica ao estilo da gramática de Montague, bem como os modelos nos quais as línguas são avaliadas.

Finalmente, a última perspectiva aberta com a implementação computacional da gramática de Montague é a possibilidade do seu uso para o modelamento da capacidade lingüística humana. Apesar de não ser, em si mesma, uma teoria apta para a construção de modelos psicolingüísticos, devido a seu caráter abstrato, a semântica de Montague, junto com uma implementação computacional (construída com critérios fortes), pode começar a fornecer um espaço para nos perguntarmos se ela não funciona como um bom modelo para o processamento lingüístico humano, pelo menos em relação aos fenômenos semânticos para os quais ela apresenta uma solução satisfatória.

6. Summary

A few parsers for some languages, presented in **Introduction to Montague Semantics** (Dowty, Wall & Peters 1981), were developed in the present dissertation. The parsers were developed for languages L_0, L_{0E}, L_1 and L_{type}.

In their book the authors introduce Montague semantics by first presenting a version of parts of predicate calculus followed by its adaptation to the corresponding fragment of English. Additional features are introduced when moving from one language to another. L_0 corresponds to the part of predicate calculus with constants only (names and predicates). L_1 corresponds to the part with individual variables (called first order calculus). L_{type} contains variables for every type (known as high order calculus).

Contrary to what is common to almost all parsers, the ones developed here perform not only syntactic analyses but also semantic interpretations. L_{type} is the only language that has no semantic interpretation associated with it. This is, however, justifiable because during the implementation of L_{type} parser, an error was found on the formal especification of its semantic rules. So instead of simply implementing the parser, it was necessary to describe and solve that problem.

Key-words:

- Montague semantics
- Parsing as deduction
- Prolog

7. Referências bibliográficas

Ajdukiewicz, Kazimierz. 1935. Die syntaktische Konnexität. **Studia Philosophica**, **1**: 1-27. (Traduções – para o inglês: Syntactic connection, *in* S. McCall (ed.), 1967, **Polish Logic**, Oxford: Oxford University Press, pp. 207-231, traduzido por H. Weber – para o italiano: La connessità sintattica, *in* Andrea Bonomi (ed.), 1973, **Strutture Logiche del Linguaggio**, Milano: Bompiani, pp. 345-372, traduzido por Giovanni Piana – para o português: A conexidade sintática, inédito, traduzido por Lígia Negri & José Borges Neto.)

Barwise, Jon & Etchemendy, John. 1993a. **The Language of First-Order Logic (Windows Program, Tarki's World)**. 3rd. edition, revised and expended. Stanford: Center for the Study of Language and Information. *CSLI Lecture Notes*, 34.

Barwise, Jon & Etchemendy, John. 1993b. **Tarski's World: Windows Version 4.0**. Stanford: Center for the Study of Language and Information.

Borges Neto, José. 1991. **Adjetivos – Predicados Extensionais & Predicados Intensionais**. Campinas: Editora da Unicamp.

Cann, Ronnie. 1993. **Formal Semantics**. Cambridge: Cambridge University Press. *Cambridge Textbooks in Linguistics*.

Carpenter, Robert. 1997. **Type-Logical Semantics**. Cambridge, Massachusetts: The MIT Press. *Language, Speech, and Communication.*

Chierchia, Gennaro. 1997. **Semantica**. Bologna: Il Mulino. *Le Strutture Del Linguaggio.*

Chierchia, Gennaro & McConnell-Ginet, Sally. 2000. **Meaning and Grammar – An Introduction to Semantics**. Cambridge, Massachusetts: The MIT Press.

Chomsky, Noam A. 1975. Questions of the form and interpretation. **Linguistic Analysis**, **1**(1): 75-109.

Chomsky, Noam A. 1986. **Knowledge of Language: Its Nature, Origin and Use**. New York: Praeger. *Convergence Series*.

Clocksin, William F. & Mellish, Christopher S. 1981. **Programming in Prolog**. Berlin: Springer.

Covington, Michael A. 1994. **Natural Language Processing for Prolog Programmers**. Englewood Cliffs, New Jersey: Prentice Hall.

Covington, Michael A.; Nute, Donald & Vellino, André. 1998. **Prolog Programming in Depth**. Upper Saddle River, New Jersey: Prentice-Hall.

Crocker, Matthew W. 1996. **Computational Psycholinguitics – An Interdiciplinary Approach to the Study of Language**. Dordrecht: Kluwer. *Studies in Theoretical Psycholinguitics*, 20.

Dowty, David R.; Karttunen, Lauri & Zwicky, Arnold M. (eds.). 1985. **Natural Language Parsing – Psychological, Computational, and Theoretical Perspectives**. Cambridge: Cambridge University Press. *Studies in Natural Language Processing*.

Dowty, David R.; Wall, Robert E. & Peters, Stanley. 1981. **Introduction to Montague Semantics**. Dordrecht: D. Reidel.

Friedman, Joyce & Warren, David Scott. 1978. A parsing method for Montague grammars. **Linguistics and Philosophy**, **2**(3): 347-372.

Hack, Haroldo G.; Gonzalez, Alfredo L.; Catuogno, Pedro J.; Moure, Maria del Carmen & Campbell, Alicia M. 1990. Una semântica computacional del idioma español usando las teorías de R. Montague. **Theoria** (segunda epoca), **5**(12-13): 171-191.

Halvorsen, Per-Kristian & Ladusaw, William. 1979. Montague's 'Universal Grammar': An introduction for the linguist. **Linguistics and Philosophy**, **3**(2): 185-223.

Larson, Richard K. & Warren, David S. 1997. **Semantica – Version 1.0 (for NextStep)**. Cambridge, Massachusetts: The MIT Press.

Larson, Richard K. & Warren, David S. 1996. **Syntactica –NextStep Edition**. Cambridge, Massachusetts: The MIT Press.

Le, Teun van. 1993. **Techniques of Prolog Programming – With Implementation of Logical Negation and Quantified Goals**. New York: John Wiley & Sons.

Matthews, Clive. 1998. **An Introduction to Natural Language Processing through Prolog**. London: Longman. *Learning about Language*.

Pereira, Fernando C. N. 1985. A new characterization of attachment preferences. *In* Dowty, Karttunen & Zwicky (eds.) 1985: 307-319.

Pereira, Fernando C. N. & Shieber, Stuart M. 1987. **Prolog and Natural-Language Analysis**. Stanford: Center for the Study of Language and Information. *CSLI Lecture Notes*, 10.

Pereira, Fernando C. N. & Warren, D. 1983. Parsing as deduction. *In* **Proceedings of the 21st Meeting of the Association for Computational Linguistics**. Association for Computational Linguistics, 137-144.

Prawitz, D. 1965. **Natural Deduction – A Proof-Theoretical Study**. Stockholm: Almqvist & Wiksell. *Stockholm Studies in Philosophy*, 3.

Stabler Jr., Edward P. 1993. Parsing as non-Horn deduction. **Artificial Intelligence**, **63**(1-2): 225-264.

Warren, David Scott & Friedman, Joyce. 1982. Using semantics on non-context-free parsing of Montague grammar. **American Journal of Computational Linguistics**, **8**(3-4): 123-138.

8. Apêndices

8.1. Semântica de L_{0E} em DCG

```
% L0e_Sem1.ari
%      (Dowty, Wall & Peters 1981: 23)
% Semantics of L0e (DCG parser)

:- reconsult('L0e_Att.ari').

% Formation rules

s(T) --> n(E), vp([E,T]).
s(T) --> neg([T1,T]), s(T1).
s(T) --> s(T1), conj([T1,T2,T]), s(T2).

vp(SemVal) --> vi(SemVal).
vp([E1,T]) --> vt([E2,E1,T]), n(E2).

% Basic expressions

conj(SemVal) --> [Word], {word(Word,conj),
      semantic_value(Word,SemVal)}.

word(and,conj).
word(or,conj).

n(SemVal) --> [Word], {word(Word,n),
      semantic_value(Word,SemVal)}.

word(sadie,n).
word(liz,n).
word(hank,n).

vi(SemVal) --> [Word], {word(Word,vi),
      semantic_value(Word,SemVal)}.

word(snores,vi).
word(sleeps,vi).
word(is_boring,vi).

vt(SemVal) --> [Word], {word(Word,vt),
      semantic_value(Word,SemVal)}.

word(loves,vt).
word(hates,vt).
word(is_taller_than,vt).

neg(SemVal) --> [Word], {word(Word,neg),
      semantic_value(Word,SemVal)}.

word(it_is_not_the_case_that,neg).
```

8.2. Definições alternativas para o predicado "type/1"

8.2.1. Com complexidade localizada no próprio predicado "type/1"

```
% type1.ari
% Three-level finite type definition
% (complexity based on type/1)
% Dowty, Wall & Peters 1981: 89

% the first type produced is 'e'
% the last one is '[[[t,t],[t,t]],[[t,t],[t,t]]]'

type(X) :- basic(X).
type(X) :- complex(X).
type(X) :- complex1(X).
type([X,Y]) :- basic(X), complex1(Y).
type([X,Y]) :- complex1(X), basic(Y).
type([X,Y]) :- complex(X), complex1(Y).
type([X,Y]) :- complex1(X), complex(Y).
type([X,Y]) :- complex1(X), complex1(Y).

complex1([X,Y]) :- basic(X), complex(Y).
complex1([X,Y]) :- complex(X), basic(Y).
complex1([X,Y]) :- complex(X), complex(Y).

complex([X,Y]) :- basic(X), basic(Y).

basic(e).
basic(t).
```

8.2.2. Com complexidade localizada no predicado auxiliar "complex/1"

```
% type2.ari
% Three-level finite type definition
% (complexity based on complex/1)
% Dowty, Wall & Peters 1981: 89

% the first type produced is 'e'
% the last one is '[[[t,t],[t,t]],[[t,t],[t,t]]]'

type(X) :- basic(X).
type(X) :- complex(X).

complex(X) :- auxiliar(X).
complex(X) :- auxiliar1(X).
```

```
complex([X,Y]) :- basic(X), auxiliar1(Y).
complex([X,Y]) :- auxiliar1(X), basic(Y).
complex([X,Y]) :- auxiliar1(X), auxiliar1(Y).

auxiliar1([X,Y]) :- basic(X), auxiliar(Y).
auxiliar1([X,Y]) :- auxiliar(X), basic(Y).
auxiliar1([X,Y]) :- auxiliar(X), auxiliar(Y).

auxiliar([X,Y]) :- basic(X), basic(Y).

basic(e).
basic(t).
```

8.2.3. Definição por DCG, com apresentação em notação comum

```
% type3.ari
% Two-level finite DCG type definition
% Dowty, Wall & Peters 1981: 89

% the first type presented is "e"
% the last one is "<<t,t>,<t,t>>"

type :- type(Type,[]), present(Type), nl.

present([]) :- !.
present([A|B]) :- write(A), present(B).

% type recognition

type(A) :- name(A,B), translate(B,C), write(C).

translate([],[]) :- !.
tranlate([A],B) :- name(B,[A]).
translate([A|B],[C|D]) :- name(C,[A]), translate(B,D).

% types format for DCG: [<,<,t,',',t,>,',',<,t,t,>,>]

type --> basic.
type --> complex.
type --> open, basic, comma, complex, close.
type --> open, complex, comma, basic, close.
type --> open, complex, comma, complex, close.

complex --> open, basic, comma, basic, close.

basic --> [e].
basic --> [t].

open --> [<].

comma --> [','].

close --> [>].
```

8.3. Programas alternativos para a sintaxe de L_{type}

8.3.1. Sintaxe de L_{type} em DCG

```
% lt_syn3.ari
% Syntax of Ltype with DCG parser
% Dowty, Wall & Peters 1981: 91-2

%%%%%%%%%%%%%%%%%%%
% Type definition %
%%%%%%%%%%%%%%%%%%%

:- reconsult('type.ari').

%%%%%%%%%%%%%%%%%%%%%
% Basic expressions %
%%%%%%%%%%%%%%%%%%%%%

be(Type,BasicExpression) -->
      con(Type,BasicExpression).

be(Type,BasicExpression) -->
      var(Type,BasicExpression).

con(Type,Constant) -->
      [Constant],
      {Constant =.. [c,Number,Type],
      integer(Number),
      type(Type)}.

var(Type,Variable) -->
      [Variable],
      {Variable =.. [c,Number,Type],
      integer(Number),
      type(Type)}.

%%%%%%%%%%%%%%%%%%%
% Formation rules %
%%%%%%%%%%%%%%%%%%%

me(t,[[some|[ME]]|Syncat]) -->
      [some],
      me(t,ME),
      syncat(Syncat).
me(t,[[all|[ME]]|Syncat]) -->
      [all],
      me(t,ME),
      syncat(Syncat).
me(t,[[not|[ME]]|Syncat]) -->
```

```
        [not],
        me(t,ME),
        syncat(Syncat).
me(t,[some,ME]) -->
        [some],
        me(t,ME).
me(t,[all,ME]) -->
        [all],
        me(t,ME).
me(t,[not,ME]) -->
        [not],
        me(t,ME).
me(Type,BE) -->
        be(Type,BE),
        !.
me(Type,[[BE1,BE2],ME]) -->
        be([Type2,[Type1,Type]],BE1),
        be(Type2,BE2),
        me(Type1,ME).
me(Type,[BE,ME]) -->
        be(Type1,Type],BE),
        me(Type1,ME).
me(t,[ME|Syncat]) -->
        me(t,ME),
        !,
        syncat(Syncat).

% Syncategorematic binary connectives
syncat([and,ME]) -->
        [and],
        me(t,ME).
syncat([or,ME]) -->
        [or],
        me(t,ME).
syncat([if,ME]) -->
        [if],
        me(t,ME).
syncat([iff,ME]) -->
        [iff],
        me(t,ME).

%%%%%%%%%%%%%%%%%%%
% Drive predicate %
%%%%%%%%%%%%%%%%%%%

me(Expression) :-
        me(Type,Structure,Expression,[]),
        write('Type: '),
        write(Type),
        nl,
        write('Structure: '),
        nl,
        write(Structure),
        nl.
```

8.3.2. Sintaxe de L_{type} com divisão de expressão sincategoremática

```
% lt_syn4.ari
% Syntax of Ltype spliting syncategorematic expressions
% Dowty, Wall & Peters 1981: 91-2

%%%%%%%%%%%%%%%%%%%%%%
% A. Type definition %
%%%%%%%%%%%%%%%%%%%%%%

:- reconsult('type.ari').

%%%%%%%%%%%%%%%%%%%%%%%%
% B. Basic expressions %
%%%%%%%%%%%%%%%%%%%%%%%%

be(Constant,Type) :-
      Constant =.. [c,Number,Type],
      integer(Number),
      type(Type).
be(Variable,Type) :-
      Variable =.. [v,Number,Type],
      integer(Number),
      type(Type).


%%%%%%%%%%%%%%%%%%%%%%
% C. Formation rules %
%%%%%%%%%%%%%%%%%%%%%%

% Meaningful expressions

me([BasicExpression],Type,[BasicExpression]) :-
      be(BasicExpression,Type),
      !.
me(Expression,Type,Structure) :-
      split(Expression,Expression1,Expression2),
      parse(Expression1,Expression2,Type,Structure).

% Split/3 definition

:- reconsult('split.ari').

% Parser

parse([some,Var],Expression,t,[some,Var|Structure]) :-
      Var =.. [v,Number,Type],
      integer(Number),
      type(Type),
      me(Expression,t,Structure).
parse([all,Var],Expression,t,[all,Var|Structure]) :-
      Var =.. [v,Number,Type],
      integer(Number),
      type(Type),
      me(Expression,t,Structure).
```

```
parse([not],Expression,t,[not|Structure]) :-
      me(Expression,t,Structure).
parse(Expression,Syncat,t,[Structure|SyncatStructure]) :-
      me(Expression,t,Structure),
      syncat(Syncat,SyncatStructure).
parse(Expression1,Expression2,Type,[Structure1,Structure2])
      :-
      me(Expression1,Type1,Structure1),
      me(Expression2,Type2,Structure2),
      cancel(Type1,Type2,Type).

% Syncategorematic expressions

syncat([and|Expression],[and,Structure]) :-
      me(Expression,t,Structure).
syncat([or|Expression],[or,Structure]) :-
      me(Expression,t,Structure).
syncat([if|Expression],[if,Structure]) :-
      me(Expression,t,Structure).
syncat([iff|Expression],[iff,Structure]) :-
      me(Expression,t,Structure).


% Type cacellation

cancel([A,B],A,B).
```

8.3.3. Sintaxe de L_{type} com analisador por deslocamento-e-redução

```
% lt_syn5.ari
% Syntax of Ltype
% Dowty, Wall & Peters 1981: 91-2
%     Shift-reduce parser
%     Covington 1994: 159

%%%%%%%%%%%%%%%%%%%%
% Type definition %
%%%%%%%%%%%%%%%%%%%%

:- reconsult('type.ari').

%%%%%%%%%%%%%%%%%%%%%%
% Basic expressions %
%%%%%%%%%%%%%%%%%%%%%%

% Constants

be(c,Number,Type) :-
      integer(Number),
      type(Type).

% Variables
```

```
be(v,Number,Type) :-
      integer(Number),
      type(Type).

%%%%%%%%%%%%%%%%%%%
% Formation rules %
%%%%%%%%%%%%%%%%%%%

% 9. existential quantification

rule([[t,ME],Var,some|Rest],[[t,[some,Var,ME]]|Rest]) :-
      Var =.. [v,Number,Type],
      be(v,Number,Type).

% 8. universal quantification

rule([[t,ME],Var,all|Rest],[[t,[all,Var,ME]]|Rest]) :-
      Var =.. [v,Number,Type],
      be(v,Number,Type).

% 7. equivalence

rule([[t,ME2],iff,[t,ME1]|Rest],[[t,[ME1,iff,ME2]]|Rest]).

% 6. implication

rule([[t,ME2],if,[t,ME1]|Rest],[[t,[ME1,if,ME2]]|Rest]).

% 5. disjunction

rule([[t,ME2],or,[t,ME1]|Rest],[[t,[ME1,or,ME2]]|Rest]).

% 4. conjunction

rule([[t,ME2],and,[t,ME1]|Rest],[[t,[ME1,and,ME2]]|Rest]).

% 3. negation

rule([[t,ME],not|Rest],[[t,[not,ME]]|Rest]).

% 2. type cancellation

rule([[A,ME2],[[A,B],ME1]|Rest],[[B,[ME1,ME2]]|Rest]).

% 1. basic expression

rule([BE|Rest],[[Type,BE]|Rest]) :-
      BE =.. [CV,Number,Type],
      be(CV,Number,Type).

%%%%%%%%%%
% Parser %
%%%%%%%%%%

% Drive clause

parse(Expression) :-
```

```
        parse(Expression,[],[Type,Structure]),
        write('Type: '),
        write(Type),
        nl,
        write('Structure: '),
        nl,
        write(Structure),
        nl.

% Shift-reduce parser

parse(Input,Stack,Result) :-
        shift(Input,Stack,NewStack,NewInput),
        reduce(NewStack,ReducedStack),
        parse(NewInput,ReducedStack,Result).
parse([],[[Type,Structure]],[Type,Structure]).

shift([First|Rest],Stack,[First|Stack],Rest).

reduce(Stack,ReducedStack) :-
        rule(Stack,NewStack),
        reduce(NewStack,ReducedStack).
reduce(ReducedStack,ReducedStack).
```