Representação semântica em analisadores gramaticais

Luiz Arthur Pagani (UFPr)

Resumo

Apresenta-se, no presente texto, uma reflexão sobre alguns analisadores gramaticais que oferecem não apenas a estrutura sintática da expressão analisada, mas principalmente uma representação da sua interpretação semântica. A questão central a ser discutida é o uso de variáveis da língua de programação (o Prolog, no caso) para replicar as variáveis do cálculo de predicados, escolhido para representar a interpretação semântica. Como conseqüência, propõe-se uma representação na qual as variáveis do cálculo de predicados recebem uma representação diferente das variáveis do Prolog e a redução-β precisa ser explicitamente definida.

1 Introdução

Ainda hoje, quando se fala em analisadores gramaticais (*parsers*), principalmente no Brasil (como em [11], por exemplo), normalmente o que se designa são apenas analisadores sintáticos, porque em quase todos o resultado do processamento é apenas uma representação da estrutura sintática da expressão analisada (como em [6] e [9]), apesar de não ser nova a preocupação em construir a representação semântica junto com a análise sintática (como em [12, ps. 91–114] e [5, ps. 196–256]). Também não é nova a preocupação com a formalização da interpretação semântica, o que torna ainda menos compreensível essa ausência nos analisadores gramaticais, dado que o modelamento computacional oferece um bom "campo de provas" para teorias formalizadas.

Portanto, o que se pretende com este trabalho é apresentar uma série de analisadores gramaticais bastante simples, implementados em Prolog, nos quais se introduz uma representação semântica para cada expressão analisada. A partir desta apresentação, serão discutidas questões relativas tanto ao formato da representação semântica quanto à implementação computacional. Em relação à representação semântica, mais especificamente, discutiremos a escolha de representar a interpretação de um nome próprio como *Pedro* através de `(pedro^S)^S` (introduzida no analisador devido à compatibilização com os quantificadores generalizados). Já sobre a implementação computacional, discutiremos o uso de variáveis do Prolog para representar as variáveis da interpretação semântica e a maneira como a redução-β é definida — o que causa uma unificação incomum de variáveis do ponto de vista semântico em alguns momentos do processamento (ainda que o resultado computacional final seja coerente).

2 Processadores sintáticos

No processamento sintático computacional, é comum encontrar uma primeira grande diferença entre reconhecedores e analisadores sintáticos ([8, p. 5], [12, p. 35n] e [9, p. 123]). Um reconhecedor é caracterizado como um algoritmo que recebe como entrada uma expressão lingüística e apresenta como saída apenas uma resposta afirmativa ou negativa, conforme a expressão seja, respectivamente, gramatical (de acordo com as regras estipuladas no algoritmo) ou não; um analisador gramatical, por outro lado, além de avaliar a gramaticalidade ou não da expressão, ainda apresenta uma análise sintática para a expressão avaliada (ou mais de uma, caso a expressão seja estruturalmente ambígua).

2.1 Reconhecedor sintático

Um exemplo simples de reconhecedor sintático pode ser apresentado, como no programa `rec_dcg.pl`, da seguinte maneira:[1]

```
% Regras sintagmáticas

s --> sn, sv.

sn --> det, n(c).
sn --> n(p).

sv --> v(td), sn.
sv --> v(i).

% Itens lexicais

det --> [todo].
det --> [um].

n(c) --> [buraco].
n(c) --> [menino].
n(p) --> [pedro].

v(i) --> [dormiu].
v(td) --> [saltou].
```

Um programa em Prolog como este, escrito na notação conhecida como DCG (*Definite Clause Grammar*), replica com bastante semelhança as regras de estrutura sintagmática com as quais os lingüistas já estão largamente acostumados a trabalhar. O programa acima, especificamente, implementa a seguinte gramática independente de contexto:

$S \rightarrow SN\ SV$	$Det \rightarrow \{todo, um\}$
$SN \rightarrow Det\ N_C$	$N_C \rightarrow \{buraco, menino\}$
$SN \rightarrow N_P$	$N_P \rightarrow \{Pedro\}$
$SV \rightarrow V_{TD}\ SN$	$V_I \rightarrow \{dormiu\}$
$SV \rightarrow V_I$	$V_{TD} \rightarrow \{saltou\}$

Depois de carregar este programa num interpretador de Prolog,[2] pode-se solicitar que ele nos informe sobre a gramaticalidade das expressões *Pedro dormiu* e *todo menino saltou um buraco* com as seguintes diretivas:

```
?- s([pedro, dormiu], []).
yes ? ;
no
?- s([todo, menino, saltou, um, buraco], []).
yes ? ;
no
```

[1] Mais do que apenas simples, este reconhecedor é sintaticamente precário: ele não lida, por exemplo, nem com nenhuma concordância, nem com qualquer mobilidade distribuicional, além de operar com poucas categorias. Como o objetivo do presente texto é debater a representação semântica, a exemplificação sintática é propositalmente precária, porque não haveria espaço aqui para apresentar questões sobre estruturação sintática e discutir soluções para elas.

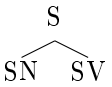
[2] Todos os programas apresentados no texto podem ser obtidos na minha página de internet `http://www.ufpr.br/~arthur`. Eles foram testados nos interpretadores GNU-Prolog e SWI-Prolog, que também podem ser ambos obtidos gratuitamente pela internet nas páginas `http://swi-prolog.org` e `http://www.gprolog.org`.

Como ambas as expressões são sentenças, e perguntamos ao interpretador efetivamente se elas eram sentenças, ele responde afirmativamente para as duas.[3] Do ponto de vista lingüístico, no entanto, apenas saber se uma expressão é ou não gramatical pode não ser muito revelador; algumas vezes é preciso saber também como a expressão se estrutura. Para isso, é necessário um analisador sintático, no qual se registre o histórico de sua construção; mas o reconhecedor acima pode ser facilmente adaptado para essa tarefa.

2.2 Analisador sintático

Para transformar um reconhecedor, escrito em DCG, num analisador, basta acrescentar um argumento ao predicado no qual definimos a construção da expressão, e registrar nesse argumento o histórico desta construção, usando um formato adequado para a sua representação.

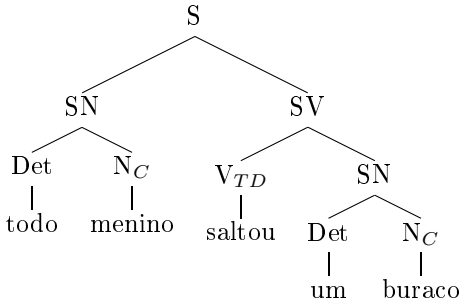
Normalmente, os lingüistas representam as estruturas sintagmáticas através de árvores, e isso pode ser feito em Prolog na forma de predicados e argumentos, colocando o nó superior como predicado e os seus ramos inferiores como argumentos. Assim, uma árvore como:



poderia então ser representada como `s(sn, sv)`; e com um termo mais complexo, como:

```
s(sn(det(todo), n(menino)), sv(vtd(saltou), sn(det(um), n(buraco))))
```

estaríamos representando a árvore para *todo menino saltou um buraco*:



O analisador gramatical, portanto, poderia ser reescrito como o programa `ana_dcg.pl` abaixo, a partir do reconhecedor acima.

```
% Regras sintagmáticas

s(s(SN, SV)) --> sn(SN), sv(SV).

sn(sn(Det, N)) --> det(Det), n(c, N).
sn(sn(N)) --> n(p, N).

sv(sv(V, SN)) --> v(td, V), sn(SN).
sv(sv(V)) --> v(i, V).

% Itens lexicais

det(det(todo)) --> [todo].
det(det(um)) --> [um].

n(c, n(buraco)) --> [buraco].
n(c, n(menino)) --> [menino].
n(p, np(pedro)) --> [pedro].
```

[3] O interpretador indica que uma solução foi encontrada (`yes`) e pergunta se desejamos outras soluções (`?`); o ponto e vírgula solicita as alternativas e `no` significa que não existe nenhuma outra solução.

```
v(i, vi(dormiu)) --> [dormiu].
v(td, vtd(saltou)) --> [saltou].
```

Agora, as análises sintáticas para as sentenças *Pedro dormiu* e *todo menino saltou um buraco* podem ser solicitadas através das diretivas:

```
?- s(Estrutura, [pedro, dormiu], []).
Estrutura = s(sn(np(pedro)),sv(vi(dormiu)))
yes
?- s(Estrutura, [todo, menino, saltou, um, buraco], []).
Estrutura = s(sn(det(todo),n(menino)),sv(vtd(saltou),sn(det(um),n(buraco)))) ? ;
no
```

Nas consultas acima, o interpretador nos informa a estrutura da expressão. No primeiro caso, como não há mais soluções alternativas, o interpretador não nos indaga sobre elas. No segundo caso, depois de apresentar uma resposta, ele nos pergunta se queremos obter outras respostas; como não há outras alternativas para se estruturar a sentença, o interpretador nos responde `no` (no caso de uma sentença ambígua, o interpretador nos apresentaria cada uma das possibilidades à medida que fôssemos teclando `;`).

Como veremos na seção a seguir, um analisador sintático também pode ser facilmente adaptado para registrar a interpretação semântica; basta escolher uma forma para se representá-la e codificá-la em Prolog acrescentando mais um argumento para isso.

3 Analisadores gramaticais

Como estamos interessados em uma questão muito específica da representação semântica, na qual a estrutura sintática não tem relevância (já que não estaremos discutindo nada sobre ambigüidade, nem mesmo a estrutural), vamos nos ater apenas na construção da representação semântica, desconsiderando a estrutura sintática.

3.1 Termo-λ como representação

Tem sido bastante recorrente na Semântica Formal representarmos as interpretações semânticas de uma expressão através do cálculo de predicados acrescido do cálculo-λ. Em diversos manuais introdutórios (como [4, 2, 7, 3]), encontramos explicações do seguinte tipo.

Para representar o significado de uma sentença como *Pedro dormiu*, recorremos à idéia aristotélica de que ela seria verdadeira quando dita numa situação em que Pedro estivesse dormindo, e seria falsa quando não. Isso pode ser representado através da teoria de conjuntos, dizendo que *dormiu* denota o conjunto dos indivíduos que estavam dormindo (D) e que *Pedro* denota o indivíduo que é o Pedro (p); assim, a interpretação semântica desta sentença pode ser representada composicionalmente pela fórmula do cálculo de predicados ($D\ p$),[4] que diz exatamente isso.

Uma implementação deste tipo de representação poderia ser feita através da regra

```
s(V@N) --> n(N), v(V).
```

(O símbolo `@` é normalmente usado para representar a aplicação funcional, de forma que `V@N` é o equivalente em Prolog de ($V\ N$).)

Mas isso ainda não é suficiente para lidar com sintagmas nominais mais complexos, como os quantificadores generalizados. Para estes, a Semântica Formal postula que eles sejam funções que tomam a denotação do sintagma verbal (no caso do sujeito) ou do verbo (no caso do objeto direto) — ao contrário dos nomes próprios, que são argumentos para o sintagma verbal (sujeito) e para o verbo (objeto direto). Assim, o

[4] Prefiro a notação *(Função Argumento)*, mais utilizada na Gramática Categorial, ao invés da tradicional *Função(Argumento)*.

significado de *todo menino* seria $\lambda Q.\forall x.(M\ x) \rightarrow (Q\ x)$, e o de *um buraco* seria $\lambda S.\exists y.(B\ y) \wedge (S\ y)$; definindo M e B como os significados, respectivamente, de *menino* e *buraco*, o significado de *todo* precisaria ser $\lambda P.\lambda Q.\forall x.(P\ x) \rightarrow (Q\ x)$; e o de *um*, $\lambda R.\lambda S.\exists y.(S\ y) \wedge (R\ y)$ — ambos por abstração.[5]

No entanto, com os quantificadores generalizados, a regra para a construção da representação semântica da sentença precisa estipular que este colabora com uma função enquanto o verbo intransitivo entraria com o argumento; assim, o significado de *todo menino dormiu* seria $(\lambda Q.\forall x.(M\ x) \rightarrow (Q\ x)\ D)$, que por redução-$\beta$ é equivalente a $\forall x.(M\ x) \rightarrow (D\ x)$.

Para implementar isso em Prolog, precisaríamos da DCG abaixo (`qg_red.pl`), acrescida de um predicado para executar a redução:

```
:- op(500, yfx, @).
:- op(550, xfx, =>).
:- op(600, xfy, ^).

s(S) --> sn(SN), v(i, V), {reduz(SN@V, S)}.
s(V@N) --> n(p, N), v(i, V).

sn(SN) --> det(Det), n(c, N), {reduz(Det@N, SN)}.

det(P^Q^qualquer(X, P@X => Q@X)) --> [todo].

n(c, m) --> [menino].
n(p, p) --> [pedro].

v(i, d) --> [dormiu].

reduz((Argumento^Expressão)@Argumento, Expressão).
```

Nesta implementação, foi preciso definir como operadores os símbolos `@` (para a aplicação funcional), `=>` (para representar a implicação) e `^` (para o operador-λ); além disso, a quantificação universal está representada pelo predicado `qualquer/2`,[6] cujo primeiro argumento deve conter a variável a ser quantificada, e no segundo registra-se o escopo da quantificação. A definição do predicado `reduz/2` foi baseada na definição do predicado `reduce/3` de [12, p. 96]; ele precisa ser executado nas definições da sentença e do quantificador generalizado, para que tenhamos sempre a forma mais reduzida da expressão.

Rodando este programa para as sentenças *Pedro dormiu* e *todo menino dormiu*, através das perguntas abaixo, obtemos as respostas desejadas: `qualquer(A,m@A=>d@A)` e `d@p`, que são respectivamente as representações em Prolog para $\forall x.(M\ x) \rightarrow (D\ x)$ e $(D\ p)$.

```
?- s(Interpretação, [todo, menino, dormiu], []).
Interpretação = qualquer(A,m@A=>d@A) ? ;
no
?- s(Interpretação, [pedro, dormiu], []).
Interpretação = d@p
yes
```

Como se pode perceber, da forma simplificada como foi apresentado, o analisador lida com o nome próprio e com o quantificador generalizado de maneiras diferentes: na posição de sujeito, o primeiro é argumento da denotação do sintagma verbal, enquanto o segundo é função que toma o sintagma verbal como argumento. Contudo, uma maneira simples de unificar os dois, recorrendo à antiga solução encontrada por Montague, é através da promoção de tipo (*type raising*) do nome próprio; assim, *Pedro* passa a denotar $\lambda P.(P\ p)$ (em Prolog, `P^P@p`). Isso leva à seguinte reconfiguração do analisador, no programa `qg.pl`:

[5] Para detalhes sobre o cálculo-λ, ver [3, p. 50].

[6] Como o foco aqui é a construção de uma representação semântica, além da questão da ambigüidade do escopo entre quantificadores, não vamos poder também considerar a questão da ambigüidade das leituras distribuídas ou coletivas; assim, escolhemos arbitrariamente a interpretação distribuída, representada pela expressão *qualquer*, como significado da quantificação universal.

```
:- op(500, yfx, @).
:- op(550, xfx, =>).
:- op(600, xfy, ^).

s(S) --> sn(V^S), v(i, V).

sn(SN) --> det(N^SN), n(c, N).
sn(P^P@N) --> n(p, N).

det(P^Q^qualquer(X, P@X => Q@X)) --> [todo].

n(c, m) --> [menino].
n(p, p) --> [pedro].

v(i, d) --> [dormiu].
```

Nesta nova versão, além da promoção do significado dos nomes próprios, a redução foi embutida nas regras sintagmáticas através da técnica de execução parcial [12, p. 98].[7] Como estas alterações não afetam os resultados acima, valem aquelas mesmas diretivas.

Agora que revisamos alguns conceitos básicos para se compreender o funcionamento de analisadores gramaticais que se ocupam da interpretação semântica, vamos observar o funcionamento de dois deles: o de Pereira & Shieber e o de Covington.

3.2 Pereira & Shieber

Um dos analisadores gramaticais mais antigos com capacidade de lidar com a representação semântica é o de [12, ps. 102–103] (que chamaremos de `pershi.pl`):

```
:- op(500, xfy, &).
:- op(510, xfy, =>).

s(S) --> np(VP^S), vp(VP).

np(NP) --> det(N2^NP), n(N1), optrel(N1^N2).
np((E^S)^S) --> pn(E).

vp(X^S) --> tv(X^IV), np(IV^S).
vp(IV) --> iv(IV).

optrel((X^S1)^(X^(S1 & S2))) --> [that], vp(X^S2).
optrel(N^N) --> [].

det(LF) --> [D], {det(D, LF)}.
det(every, (X^S1)^(X^S2)^all(X,S1 => S2)).
det(a, (X^S1)^(X^S2)^exists(X, S1 & S2)).

n(LF) --> [N], {n(N, LF)}.
n(program, X^program(X)).
n(student, X^student(X)).
```

[7] Basicamente, a execução parcial consiste em redistribuir a definição de um predicado nas definições dos outros predicados nos quais ele era usado. No caso de `reduz((Argumento^Expressão)@Argumento, Expressão)` e `s(S) --> sn(SN), v(i, V), {reduz(SN@V, S)}`, o procedimento de unificação faz com que `SN = Argumento^Expressão`, `V = Argumento` e `S = Expressão`; portanto, decorre dessa unificação que `SN = V^S`. Assim, aquelas duas cláusulas podem ser reduzidas a uma única: "`s(S) --> sn(V^S), v(i, V).`"

```
pn(E) --> [PN], {pn(PN, E)}.
pn(terry, terry).
pn(shrdlu, shrdlu).

tv(LF) --> [TV], {tv(TV, LF)}.
tv(wrote, X^Y^wrote(X, Y)).

iv(LF) --> [IV], {iv(IV, LF)}.
iv(halts, X^halts(X)).
```

Como o programa foi transcrito integralmente, podemos ver que, para os sintagmas nominais, além das possibilidades de serem compostos por um nome próprio ou por um determinante seguido de um nome comum, há ainda a alternativa de acrescentar-lhes uma oração relativa; isso é feito através do predicado `optrel`, que pode não acrescentar nenhum material segmental ao sintagma nominal (`optrel(N^N) --> [].`) ou continuá-lo acrescentando o complementizador (*that*, introduzido sincategorialmente) seguido de um sintagma verbal (`optrel((X^S1)^(X^(S1 & S2))) --> [that], vp(X^S2).`). Ainda que essa opção possa não ser a melhor do ponto de vista lingüístico, novamente não podemos nos deter nela, devido ao foco semântico do presente trabalho.

No resto, o programa tem quase a mesma estrutura comentada anteriormente. Três diferenças, porém, merecem ser observadas.

A primeira delas, que também é a causa das outras duas, é que um quantificador generalizado pode ser representado como `(X^S1)^(X^S2)^all(X,S1=>S2)` (o universal) e não como `P^Q^qualquer(X, P@X => Q@X)`; isso se deve à decisão dos autores de também aplicar a execução parcial na aplicação funcional: assim, ao invés de uma implicação entre duas aplicações funcionais (`P@X` e `Q@X`), eles podem fazer uma implicação entre dois termos (`S1` e `S2`) nos quais as aplicações funcionais e suas respectivas reduções acontecem à esquerda do operador-λ (`X^S1` e `X^S2`). A explicação [12, p. 100] é a de que a forma das fórmulas para os quantificadores generalizados precisa ser `Q^all(X, program(X) & Q)`, sendo que "`Q` é a aplicação de `R` a `X`, de modo que se efetue `reduz(R, X, Q)`". A unificação faz `R = X^Q`, e a fórmula pode ser reescrita como `(X^Q)^all(X, program(X) => Q)`; usando o mesmo raciocínio, a abstração de `Y^program(Y)` desta fórmula nos leva a `(X^P)^(X^Q)^all(X, P => Q)` como representação do significado do determinante *every*.

A segunda diferença é a maneira como a interpretação dos nomes próprios é feita. Devido à escolha da execução parcial da redução-β e da aplicação funcional, a representação do significado do nome próprio *Shrdlu*[8] acaba ficando como `(shrdlu^P)^P`, ao invés da mais simples `P^P@p`, de *Pedro*. Como um verbo como *halts* recebe a representação semântica `X^halts(X)`, a unificação para `s(S) --> np(VP^S), vp(VP).` faz `VP^S = (shrdly^P)^P` e `VP = X^halts(X)`; destas primeiras duas equivalências, decorre que:

- `(X^halts(X))^S = (shrdlu^P)^P`,
- `X = shrdlu`,
- `P = halts(X)` (e, portanto, `P = halts(shrdlu)`),
- `S = P` e, finalmente,
- `S = halts(shrdlu)`.

Pode parecer complicado, mas como reconhecem os próprios autores: [12, p. 101]:

> a forma lógica de *Shrdlu* parece contra-intuitiva, porque não corresponde à codificação de nenhuma expressão-λ. A posição que deveria ser ocupada por uma variável é ocupada pela constante `shrdlu`.

[8] *Shrdlu* é o nome de um programa para processamento de língua natural, escrito no final dos anos 60 por Terry Winograd, para sua tese de doutorado; este é o famoso programa que manipulava um mundo de blocos, e não é difícil nos depararmos com exemplos como *put the block in the box on the table*, que era um tipo de expressão bastante empregada no Shrdlu mas que apresentava um bom grau de ambigüidade. Mais informações podem ser obtidas na página do próprio Terry Wynograd sobre o Shrdlu em `http://hci.stanford.edu/~winograd/shrdlu/`.

Isto porque

> a execução parcial da aplicação pode resultar em expressões bizarras, justamente por causa da execução parcial. Apenas uma parte da tarefa da redução-β é executada, o restante acaba sendo realizado durante o processamento, quando as variáveis envolvidas são instanciadas. Portanto, não devemos nos preocupar tanto com o fato de que a codificação das expressões-λ que estamos usando apresentem algumas propriedades que o cálculo abstratamente não apresente.

(Efetivamente, como veremos já, o funcionamento do programa é realmente o esperado; mas como veremos um pouco mais adiante, pode haver algum motivo para nos fazer desconfiar dessa codificação através de "expressões bizarras".)

Finalmente, a terceira diferença está relacionada à representação dos verbos transitivos diretos: um verbo como *wrote* é representado através da fórmula `X^Y^wrote(X, Y)`, ao invés de `Y^X^wrote(X, Y)`, que é o que seria de se esperar a partir da expressão $\lambda y.\lambda x.wrote(x, y)$. Novamente, a inversão das variáveis na ordem de combinação é afetada pela execução parcial e pela facilidade para a unificação da regra do sintagma verbal para o verbo transitivo direto: `vp(X^S) --> tv(X^IV), np(IV^S).` Pela unificação, `X^IV = X^Y^wrote(X, Y)`; supondo que o objeto direto seja *Terry*, também é preciso que `IV^S = (terry^P)^P`. Assim, `IV = Y^wrote(X, Y) = terry^P`, `S = P`; daí, então, `Y = terry`, `P = wrote(X, Y)` (conseqüentemente `P = wrote(X, terry)`) e `S = wrote(X, terry)`. Portanto, o significado do SV (`X^S`) é instanciado como `X^wrote(X, terry)`.

De qualquer maneira, da forma como está, o programa é capaz de construir adequadamente as representações (que os autores chamam de forma lógica (*logical form*), seguindo uma certa tradição gerativista) para o significado das expressões analisadas. Em [12, p. 103], são apresentados três exemplos:

```
?- s(LF, [terry, wrote, shrdlu], []).
LF = wrote(terry, shrdlu) ? ;
no
?- s(LF, [every, program, halts], []).
LF = all(A, program(A)=>halts(A)) ? ;
no
?- s(LF, [every, student, wrote, a, program], []).
LF = all(A, student(A)=>exists(B, program(B)&wrote(A, B))) ? ;
no
```

Não é preciso muita perspicácia para perceber que este analisador gramatical ainda não é capaz de apresentar as duas interpretações de uma sentença ambígua como *every student wrote a program* (uma na qual o quantificador universal tem escopo sobre o existencial — que é a que o programa apresenta — e outra na qual o existencial tem escopo sobre o universal); na seqüência, os autores [12, ps. 104–114] explicam a construção de um analisador para dar conta desta ambigüidade de escopo, mas não podemos nos ocupar com isso agora, pois nos afastaria da questão discutida aqui. Vejamos então um outro analisador gramatical que constrói a representação semântica da expressão analisada.

3.3 Covington

Um segundo analisador gramatical com capacidade de construir uma representação interpretativa é o de [5, ps. 207–210] (vamos chamá-lo de `cov.pl`):

```
s(Sem) --> np((X^Scope)^Sem), vp(X^Scope).

np((Sem^Scope)^Scope) --> proper_noun(Sem).
np(Sem) --> determiner((X^Restrictor)^Sem), n(X^Restrictor).

vp(Sem)  --> intransitive_verb(Sem).
vp(X^Pred) --> transitive_verb(Y^X^Scope), np((Y^Scope)^Pred).
```

```
n(Sem) --> common_noun(Sem).
n(X^(P, Q)) --> adjective(X^P), n(X^Q).

proper_noun(max) --> [max].
proper_noun(fido) --> [fido].

determiner((X^Restrictor)^(X^Scope)^all(X, Restrictor, Scope)) --> [every].
determiner((X^Restrictor)^(X^Scope)^some(X, Restrictor, Scope)) --> [a]; [some].

common_noun(X^cat(X)) --> [cat].
common_noun(X^dog(X)) --> [dog].
common_noun(X^professor(X)) --> [professor].

adjective(X^big(X)) --> [big].
adjective(X^brown(X)) --> [brown].
adjective(X^little(X)) --> [little].
adjective(X^green(X)) --> [green].

intransitive_verb(X^bark(X)) --> [barked].
intransitive_verb(X^meow(X)) --> [meowed].

transitive_verb(Y^X^chase(X, Y)) --> [chased].
transitive_verb(Y^X^see(X, Y)) --> [saw].
```

Ao contrário de [12], Covington não nos oferece uma listagem completa de um programa que faça a interpretação semântica; assim, o programa listado acima é uma reconstrução a partir das explicações apresentadas pelo autor.

Mas se esta reconstituição está correta, o programa de Covington não é muito diferente do de Pereira & Shieber. A primeira diferença mais evidente é a forma de codificar as fórmulas do escopo dos quantificadores: enquanto estes representam explicitamente os conectivos (`&`, para a quantificação existencial, e `=>`, para a universal), aquele prefere deixar essa informação implícita; assim, sentenças como *a cat meowed* e *every dog barked* podem ser representadas, respectivamente, como `some(A, cat(A), meow(A))` e `all(A, dog(A), bark(A))`. (A diferença entre elas vai aparecer posteriormente, nas definições dos predicados `some/3` e `all/3`: quando forem invocados, as definições desses predicados lidam com as exigências de se encontrar pelo menos uma maneira de satisfazer o primeiro e de que o segundo seja satisfeito de todas as maneiras possíveis; mas isso também não nos interessa aqui.) No entanto, essa não chega a ser uma diferença relevante para caracterizar uma estratégia distinta de lidar com a questão.

Uma diferença mais substancial pode ser vista na representação dos verbos transitivos diretos: no programa de Covington, o significado de *chased* é `Y^X^chase(X, Y)` (com as variáveis na mesma ordem da expressão lambda correspondente $\lambda y.\lambda x.chase(x, y)$), ao contrário do que acontecia com o programa de Pereira & Shieber, em que *wrote* era representado como `X^Y^wrote(X, Y)`. Ainda que também não fique caracterizada nenhuma diferença de estratégia para resolver o mesmo problema, a solução de Covington é mais coerente com a interpretação normal dos quantificadores generalizados representada através de um termo-λ. Para isso, no entanto, foi preciso "espalhar" ainda mais a solução de Pereira & Shieber: a regra para o sintagma verbal com verbo transitivo direto passa a ser `vp(X^Pred) --> transitive_verb(Y^X^Scope), np((Y^Scope)^Pred)`, de forma que a representação do verbo transitivo é manipulada para acessar a variável adequada (ao invés dela ser invertida no item lexical, de forma a ficar mais acessível).

Em relação à representação semântica do nome próprio, aqui ele também sofre uma promoção de tipo quando passa pela regra de formação do sintagma nominal, exatamente como no anterior.

E, também como antes, essa versão não capta ambigüidades de escopo. Mas, ao contrário dos autores anteriores, apesar de reconhecer a existência da ambigüidade de escopo, Covington não chega a apresentar uma solução — ele apenas sugere, como exercício [5, p. 214], que o leitor implemente um predicado `raise_quantifier/2`, que atuaria colocando o quantificador interno numa posição mais externa.

De qualquer maneira, para construir a representação semântica de uma sentença como *every dog chased a cat*, este analisador gramatical de Covington também funciona adequadamente:

```
?- s(Sem, [every, dog, chased, a, cat], []).
Sem = all(A, dog(A), some(B, cat(B), chase(A, B))) ? ;
no
```

(Outra diferença, que talvez nem merecesse ser observada, aparece na regra de formação da sentença: para Covington, ela é `s(Sem) --> np((X^Scope)^Sem), vp(X^Scope)`, enquanto que, para Pereira & Shieber, ela é `s(S) --> np(VP^S), vp(VP)`. Essas duas regras, porém, são completamente equivalentes, como se pode constatar fazendo `S = Sem` e `VP = X^Scope`.)

4 Analisador com histórico derivacional

Segundo Pereira & Shieber [12, ps. 95–96n]:

> De um ponto de vista estritamente lógico, o uso de variáveis do Prolog para codificar variáveis da LPO [lógica de primeira ordem] é incorreto, configurando um caso de confusão entre variáveis do objeto (as da forma lógica) e variáveis da metalinguagem (as do Prolog, a metalinguagem usada aqui para descrever a relação entre as cadeias lingüísticas e as formas lógicas). Seria possível evitar esta confusão entre variáveis do objeto e da metalinguagem através de uma descrição um pouco mais complexa. No entanto, quando este abuso de notação é adequadamente compreendido, é provável que ele não cause nenhum problema e ainda traga benefícios substanciais em relação à simplicidade do programa.

Esta opinião faz parecer que uma escolha como esta não causa grandes danos, apesar de não ser "logicamente correta", e efetivamente esta solução tem sido adotada mesmo em analisadores mais recentes, como [1, p. 73].[9] O que se pretende mostrar nesta seção é que, implementando as variáveis da representação semântica através de variáveis do Prolog, a adaptação mais evidente para estes analisadores resulta na construção de uma derivação estrutural completamente inadequada para representar as etapas de construção da interpretação; depois disso, apresentaremos um analisador que executa a derivação respeitando adequadamente o princípio da representação composicional.

4.1 Adaptando os analisadores para a derivação

O programa abaixo (`deriv_sem.pl`) é, basicamente, o analisador gramatical de Covnigton adaptado para não apenas dar a representação semântica da expressão analisada, mas também para apresentar o seu histórico derivacional. De um ponto de vista mais lingüisticamente motivado, a apresentação do histórico derivacional ajudaria a compreender as etapas envolvidas na construção composicional do significado da expressão.

```
analisa(Expressão) :-
        categoria(Categoria),
        (   (Categoria = n; Categoria = v)
        ->  Regra =.. [Categoria, _, Sem]
        ;   Regra =.. [Categoria, Sem]
        ),
        phrase(Regra, Expressão, []),
        write(Categoria), write(':'), nl,
        apresenta(3, Sem).

categoria(s).
categoria(sn).
```

[9] Infelizmente, não foi possível incluir uma avaliação do analisador gramatical proposto neste livro, apesar de estar inteiramente relacionado ao tema tratado aqui, por dois motivos: primeiro, porque não foi possível ter acesso a esta referência a tempo hábil e, segundo, porque estaríamos nos afastando da proposta inicial (o que também nos faria extrapolar o espaço disponível).

```
categoria(sv).
categoria(det).
categoria(n).
categoria(v).

apresenta(N, [X, Y]) :-
        tab(N),
        write(X),
        nl, nl,
        M is N + 5,
        apresenta_lista(M, Y).

apresenta_lista(_, []).
apresenta_lista(N, [X|Y]) :-
        apresenta(N, X),
        apresenta_lista(N, Y).

s([S, [SN_Comp, SV_Comp]]) --> sn(SN_Comp), sv(SV_Comp),
        {SN_Comp = [SN, _], SV_Comp = [SV, _], SN = SV^S}.

sn([N_Promov, [N_Comp]]) --> n(próprio, N_Comp),
        {N_Comp = [N, _], N_Promov = (N^P)^P}.
sn([SN, [Det_Comp, N_Comp]]) --> det(Det_Comp), n(comum, N_Comp),
        {Det_Comp = [Det, _], N_Comp = [N, _], Det = N^SN}.

sv(V) --> v(intransitivo, V).
sv([X^SV, [V_Comp, SN_Comp]]) --> v(transitivo, V_Comp), sn(SN_Comp),
        {SN_Comp = [(Y^V)^SV|_], V_Comp = [Y^X^V|_]}.

n(Tipo, [Denotação, []]) --> [Expressão],
        {n(Tipo, Expressão, Denotação)}.

n(comum, buraco, X^buraco(X)).
n(comum, menino, X^menino(X)).
n(próprio, pedro, pedro).

det([Denotação, []]) --> [Expressão], {det(Denotação, Expressão)}.

det((Arg^Func1)^(Arg^Func2)^qualquer(Arg, Func1, Func2), todo).
det((Arg^Func1)^(Arg^Func2)^algum(Arg, Func1, Func2), um).

v(Tipo, [Denotação, []]) --> [Expressão],
        {v(Tipo, Expressão, Denotação)}.

v(intransitivo, dormiu, X^dormir(X)).
v(transitivo, saltou, Y^X^saltar(X, Y)).
```

Os predicados `analisa/1`, `categoria/1`, `apresenta/2` e `apresenta_lista/2` servem apenas para facilitar a inicialização e a apresentação da análise gramatical (formam uma espécie de interface rudimentar), e por isso não serão comentados aqui.

Esta versão consiste apenas na adaptação do argumento para a representação semântica, de forma que nele se documente a derivação da representação semântica. Assim, a representação do nome próprio *Pedro* é `[p, []]`, indicando que ele denota o indivíduo `p` e sua derivação não depende de mais nada (a lista vazia); a representação de *dormiu* fica como `[X^dormir(X), []]`, também nos informando que a denotação

`X^dormir(X)` não foi construída a partir de mais nada. Observando a regra de estruturação do sintagma nominal, para o mesmo nome próprio *Pedro*, percebemos que ela construiria a representação `[(p^P)^P, [p, []]]`; esta representação nos diz que a denotação agora é `(p^P)^P` e ela foi obtida a partir da denotação mais simples `[p, []]`.

Isto pode ser facilmente constatado rodando as diretivas abaixo:

```
?- analisa([pedro]).
sn:
   (pedro^A)^A
      pedro
yes ? ;
n:
   pedro
yes ? ;
no
?- analisa([dormiu]).
sv:
   A^dormir(A)
yes ? ;
v:
   A^dormir(A)
yes ? ;
no
```

Da forma como a interface foi definida, o analisador gramatical encontra uma primeira representação para *Pedro* e a apresenta de uma forma indentada: na primeira linha, ele nos informa que é um SN; na segunda linha, com um recuo, ele nos informa a sua denotação (`(pedro^A)^A`); e na terceira linha, com ainda mais recuo, vemos a informação de que a denotação inicial (através da qual a outra foi composta) é `pedro`. Se, depois disso, solicitarmos que o interpretador busque uma solução alternativa, ele ainda vai nos informar que *Pedro* também é um N, denotando `pedro`. O mesmo ocorre com *correu*.

Mas se pedirmos para analisar *Pedro dormiu*, o resultado é bastante estranho:

```
?- analisa([pedro, dormiu]).
s:
   dormir(pedro)
      (pedro^dormir(pedro))^dormir(pedro)
           pedro
      pedro^dormir(pedro)
yes ? ;
no
```

Agora, o analisador nos informa que a expressão é uma sentença (`s`) e, como esperávamos, que seu significado é `dormir(pedro)`. Mas ele também nos informa que seu significado é obtido composicionalmente de `(pedro^dormir(pedro))^dormir(pedro)` e de `pedro^dormir(pedro)`; traduzindo de volta para a notação dos termos-λ, estas representações corresponderiam a $\lambda(\lambda pedro.dormir(pedro)).dormir(pedro)$ e $\lambda pedro.dormir(pedro)$, respectivamente — que, como admitiram Pereira & Shieber, não constituem termos-λ bem formados.[10]

4.2 Construindo uma derivação mais coerente

No programa a seguir (`lambda.pl`), finalmente, implementa-se a sugestão de Pereira & Shieber, empregando as variáveis do Prolog não como variáveis da interpretação, mas apenas para guiar a introdução destas últimas na composição das representações; estas variáveis da metalinguagem da representação semântica terão o

[10] A coisa ainda ficaria mais complicada para *Pedro saltou um buraco* ou *todo menino saltou um buraco*; mas como a complicação é essencialmente a mesma, nos contentamos com o exemplo mais simples apresentado acima.

formato `x(N)`, para qualquer número inteiro `N` (as do Prolog, portanto, são variáveis da metalinguagem para a construção da representação; ambos os tipo de variáveis são metavariáveis, mas de metalinguagens diferentes); além disso, como é exigido pelas teorias de prova, toda variável introduzida na análise não pode ter sido usada antes.

```
:- op(500, yfx, @).
:- op(550, xfx, /\).
:- op(550, xfx, =>).
:- op(600, xfy, ^).

%%%%%%%%%%%%
% Interface %
%%%%%%%%%%%%

analisa(Expressão) :-
        categoria(Categoria),
        (   (Categoria = n; Categoria = v)
        ->  Regra =.. [Categoria, _, Sem]
        ;   Regra =.. [Categoria, 1, _, Sem]
        ),
        phrase(Regra, Expressão, []),
        nl,
        apresenta(0, Sem).

categoria(s).
categoria(sn).
categoria(sv).
categoria(det).
categoria(n).
categoria(v).

apresenta(N, [X, []]) :-
     tab(N),
     write(X),
     nl,
     !.
apresenta(N, [X|Y]) :-
     tab(N),
     X \= [_|_],
     write(X),
     apresenta_aux(N, Y),
     !.
apresenta(N, X) :-
     apresenta_lista(N, X).

apresenta_aux(_, []) :-
     !.
apresenta_aux(N, [X|Y]) :-
     X \= [_|_],
     nl,
     tab(N),
     write('<== '),
     write(X),
     apresenta_aux(N, Y).
```

```
apresenta_aux(N, [X|Y]) :-
     nl,
     M is N + 7,
     apresenta(M, X),
     apresenta_lista(M, Y).

apresenta_lista(_, []).
apresenta_lista(N, [X|Y]) :-
     nl,
     apresenta(N, X),
     apresenta_lista(N, Y).

%%%%%%%%%%%%%
% Gramática %
%%%%%%%%%%%%%

% Regras sintagmáticas

s(X1, X3, S) --> sn(X1, X2, SN), sv(X2, X3, SV),
     {SN = [SN1|_], SV = [SV1|_], reduz_lista([SN1@SV1, [SN, SV]], S)}.

sn(X1, X2, [x(X1)^x(X1)@N1, N]) --> n(p, N), {N = [N1|_], X2 is X1 + 1}.
sn(X1, X2, SN) --> det(X1, X2, Det), n(c, N),
     {Det = [Det1|_], N = [N1|_], reduz_lista([Det1@N1, [Det, N]], SN)}.

sv(X, X, V) --> v(i, V).
sv(X1, X4, SV) --> v(t, V), sn(X1, X2, SN),
     {V = [V1|_], SN = [SN1|_],
      X3 is X2 + 1, X4 is X3 + 1,
      reduz_lista([x(X2)^SN1@(x(X3)^V1@x(X3)@x(X2)), [V, SN]], SV)}.

% Itens Lexicais

det(X1, X4, [x(X1)^x(X2)^algum(x(X3), x(X1)@x(X3) /\ x(X2)@x(X3)), []]) --> [um],
     {X2 is X1 + 1, X3 is X2 + 1, X4 is X3 + 1}.
det(X1, X4, [x(X1)^x(X2)^qualquer(x(X3), x(X1)@x(X3) => x(X2)@x(X3)), []]) --> [todo],
     {X2 is X1 + 1, X3 is X2 + 1, X4 is X3 + 1}.

n(c, [b, []]) --> [buraco].
n(c, [m, []]) --> [menino].
n(p, [p, []]) --> [pedro].

v(i, [d, []]) --> [dormiu].
v(t, [s, []]) --> [saltou].

%%%%%%%%%%%%%%%%
% Redução-beta %
%%%%%%%%%%%%%%%%

% reduz_lista(Lista, Reduzido)
% ****************************************
% * Reduzido é o resultado da redução-beta *
% * recorrente do primeiro Termo da Lista  *
% ****************************************
```

```
reduz_lista(Lista, Reduzido) :-
     Lista = [Termo|_],
     reduz_termo(Termo, Resultado),
     reduz_lista([Resultado|Lista], Reduzido),
     !.
reduz_lista(Resultado, Resultado).

% reduz_termo(Termo1, Termo2)
% ************************************************
% * Termo2 é o resultado da redução-beta do Termo1 *
% ************************************************
% Aplicação
reduz_termo(Termo1@Termo2, Termo1@Resultado) :-
     reduz_termo(Termo2, Resultado).
reduz_termo(Termo1@Termo2, Resultado@Termo2) :-
     reduz_termo(Termo1, Resultado).
reduz_termo((Var^Termo)@Arg, Resultado) :-
     substitui(Termo, Var, Arg, Resultado).
% Abstração
reduz_termo(Var^Termo, Var^Resultado) :-
     reduz_termo(Termo, Resultado).
% Conjunção
reduz_termo(Termo1 /\ Termo2, Termo1 /\ Resultado) :-
     reduz_termo(Termo2, Resultado).
reduz_termo(Termo1 /\ Termo2, Resultado /\ Termo2) :-
     reduz_termo(Termo1, Resultado).
% Implicação
reduz_termo(Termo1 => Termo2, Termo1 => Resultado) :-
     reduz_termo(Termo2, Resultado).
reduz_termo(Termo1 => Termo2, Resultado => Termo2) :-
     reduz_termo(Termo1, Resultado).
% Quantificação existencial
reduz_termo(algum(Var, Termo), algum(Var, Resultado)) :-
     reduz_termo(Termo, Resultado).
% Quantificação universal
reduz_termo(qualquer(Var, Termo), qualquer(Var, Resultado)) :-
     reduz_termo(Termo, Resultado).

% substitui(Termo1, Var, Termo2, Termo3)
% ************************************************************
% * Termo3 é o resultado de substituir Var por Termo2 no Termo1 *
% ************************************************************
% Variável
substitui(x(M), x(N), Termo2, Termo3) :-
     (   x(M) \= x(N)
     ->  Termo3 = x(M)
     ;   Termo3 = Termo2
     ).
% Constante
substitui(Termo3, _, _, Termo3) :-
     atom(Termo3).
% Aplicação
substitui(Termo1a@Termo1b, Var, Termo2, Termo3a@Termo3b) :-
     substitui(Termo1a, Var, Termo2, Termo3a),
```

```
     substitui(Termo1b, Var, Termo2, Termo3b).
% Abstração
substitui(Var1^Termo1, Var, Termo2, Termo3) :-
     (  Var1 = Var
     -> Termo3 = Var1^Termo1
     ;  substitui(Termo1, Var, Termo2, Resultado),
         Termo3 = Var1^Resultado
     ).
% Conjunção
substitui(Termo1a /\ Termo1b, Var, Termo2, Termo3a /\ Termo3b) :-
     substitui(Termo1a, Var, Termo2, Termo3a),
     substitui(Termo1b, Var, Termo2, Termo3b).
% Implicação
substitui(Termo1a => Termo1b, Var, Termo2, Termo3a => Termo3b) :-
     substitui(Termo1a, Var, Termo2, Termo3a),
     substitui(Termo1b, Var, Termo2, Termo3b).
% Quantificação existencial
substitui(algum(Var1, Termo1), Var, Termo2, Termo3) :-
     (  Var1 = Var
     -> Termo3 = algum(Var1, Termo1)
     ;  substitui(Termo1, Var, Termo2, Resultado),
         Termo3 = algum(Var1, Resultado)
     ).
% Quantificação universal
substitui(qualquer(Var1, Termo1), Var, Termo2, Termo3) :-
     (  Var1 = Var
     -> Termo3 = qualquer(Var1, Termo1)
     ;  substitui(Termo1, Var, Termo2, Resultado),
         Termo3 = qualquer(Var1, Resultado)
     ).
```

Como antes, os predicados responsáveis pela interface não serão comentados.

Para os predicados que definem as construções gramaticais, a mudança é bem pequena: a definição dos itens lexicais é exatamente igual à anterior, apenas nas regras sintagmáticas é que se acrescenta um par de acumulador[11] no qual se executa o registro dos identificadores das variáveis usada na derivação (representadas no programa por `X1`, `X2`...). A notação para os predicados da representação semântica também mudou: como $F(a)$ é a aplicação da função F ao argumento a, emprega-se o mesmo símbolo `@`, escrevendo então `F@a`; assim, uma fórmula como `f(a1, a1)` passa a ser `f@a1@a2`.[12]

Pode-se observar a introdução das variáveis da metalinguagem representacional na análise da expressão *Pedro*, como abaixo.

```
?- analisa([pedro]).
x(1)^x(1)@p
       p
yes ? ;
p
yes ? ;
no
```

Quando ela é considerada como um SN (cuja análise é obtida a partir de `sn(1, X, Sem, [pedro], [])`), o seu significado é representado internamente no programa como `[x(1)^x(1)@p, [p, []]]`; já quando ela

[11] O par de acumulador é uma técnica de programação em Prolog para implementar contadores, por exemplo. Na primeira variável do par, o predicado recebe o valor inicial; na segunda, registra-se o valor a ser passado para o próximo predicado. Para mais detalhes, ver [10, p. 22].

[12] Como o operador `@` é definido com associatividade à direita (`yfx`), `f@a1@a2` é equivalente a `(f@a1)@a2`.

é um N (cuja análise inicia com `n(1, X, Sem, [pedro], [])`), o significado é apenas `[p, []]`. A interface mostra esses resultados através de apresentação indentada, também como antes.

Como a representação semântica é feita através de uma lista, em que se registra os constituintes da composição, de foma que o termo-λ mais simples que representa o significado da expressão sempre ocupa a primeira posição, todas as operações de composição do significado precisam acessar esse primeiro elemento da lista. As novas representações compostas contém o novo termo construído e todo o histórico da construção das suas partes. O significado de *Pedro dormiu*, por exemplo, antes da redução-β, fica como:

```
[(x(1)^x(1)@p)@d, [[x(1)^x(1)@p, [p, []]], [d, []]]].
```

Os termos reduzidos, por sua vez, são acrescentados no topo da lista. Para o mesmo exemplo anterior, o resultado da redução seria:

```
[d@p, (x(1)^x(1)@p)@d, [[x(1)^x(1)@p, [p, []]], [d, []]]].
```

Em relação às implementações anteriores, a principal diferença está na definição explícita e exaustiva da redução-β. O predicado `reduz_lista/2` separa o termo a ser reduzido (o primeiro da lista) e a redução é feita através do predicado `reduz_termo/2` (caso não haja nenhuma redução a ser feita, a segunda cláusula da definição de `reduz_lista/2` passa a própria lista adiante). A definição de `reduz_termo/2` é uma clássica indução na complexidade do termo (ou seja, há uma instrução de redução para cada regra de formação dos termos); ela consiste basicamente na redução recursiva dos subtermos do termo que está sendo reduzido, a única diferença nesse padrão é a redução de `(Var^Termo)@Arg` — $(\lambda Var.Termo\ \ Arg)$ — que exige a substituição da variável `Var` pelo termo `Arg` no termo `Termo`. A substituição é executada pelo predicado `substitui/4`, que também é uma indução na complexidade do termo a ser substituído, de forma que o primeiro argumento é o termo que sofrerá a substituição, o segundo é a variável que será substituída, o terceiro é o termo que ocupará o lugar da variável e o quarto registra o resultado da operação.

Como exemplo de funcionamento do programa, podemos ver abaixo a análise de *todo menino saltou um buraco.*[13]

```
analisa([todo, menino, saltou, um, buraco]).

qualquer(x(3), m@x(3)=>algum(x(6), b@x(6)/\s@x(6)@x(3)))
<== qualquer(x(3), m@x(3)=> (x(7)^algum(x(6), b@x(6)/\s@x(6)@x(7)))@x(3))
<== (x(2)^qualquer(x(3), m@x(3)=>x(2)@x(3)))@ (x(7)^algum(x(6), b@x(6)/\s@x(6)@x(7)))

       x(2)^qualquer(x(3), m@x(3)=>x(2)@x(3))
       <== (x(1)^x(2)^qualquer(x(3), x(1)@x(3)=>x(2)@x(3)))@m

              x(1)^x(2)^qualquer(x(3), x(1)@x(3)=>x(2)@x(3))

              m

       x(7)^algum(x(6), b@x(6)/\s@x(6)@x(7))
       <== x(7)^algum(x(6), b@x(6)/\ (x(8)^s@x(8)@x(7))@x(6))
       <== x(7)^ (x(5)^algum(x(6), b@x(6)/\x(5)@x(6)))@ (x(8)^s@x(8)@x(7))

              s

              x(5)^algum(x(6), b@x(6)/\x(5)@x(6))
              <== (x(4)^x(5)^algum(x(6), x(4)@x(6)/\x(5)@x(6)))@b

                     x(4)^x(5)^algum(x(6), x(4)@x(6)/\x(5)@x(6))
```

[13] Como era de se esperar, sentenças mais simples, como *Pedro dormiu* e *todo menino dormiu*, também são coerentemente analisadas; expressões de outros tipos também são analisadas, exatamente como no exemplo da análise de *Pedro*, acima.

```
                    b
yes ;
no
```

Nela, a interface apresenta recorrentemente o termo mais simples que representa o significado da expressão e, logo abaixo, os termos mais complexos que foram reduzidos até chegarem naquele primeiro mais simples (antecedidos pelo símbolo `<==`). Depois disso, os subtermos usados na composição daquele termo são apresentados com uma indentação. Assim, observando os termos indentados mais à direita no exemplo acima, podemos ver o significado de *um* (`x(4)^x(5)^algum(x(6), x(4)@x(6)/\x(5)@x(6))`) e de *buraco* (`b`) que são combinados como `(x(4)^x(5)^algum(x(6), x(4)@x(6)/\x(5)@x(6)))@b` (logo acima um pouco mais para a esquerda), que depois é reduzido para `x(5)^algum(x(6), b@x(6)/\x(5)@x(6))`.

Como é possível perceber, ao contrário do analisador antecedente, todas as etapas de construção do significado apresentam termos-λ coerentes com o que Pereira & Shieber qualificaram como "apresentados abstratamente no cálculo".

5 Conclusão

Com a apresentação de antigos analisadores gramaticais que executam a interpretação semântica, e de sua adaptação para aplicativos que demonstrem a composicionalidade da construção das interpretações, espera-se ter explicitado uma diferença essencial que distingue uma boa parte dos trabalhos em Lingüística Computacional; a saber, a diferença entre a preocupação com o produto da análise ou com o seu processo. Nos analisadores de Pereira & Shieber e de Covington, a identificação das variáveis da representação semântica e da sua construção só se justifica pelo fato de que, neles, a análise se ocupava exclusivamente com o seu produto; ou seja, apenas com a construção do termo final que representa a interpretação. No analisador proposto aqui, esta identificação não se sustenta porque a preocupação é a de não apenas tomar conhecimento da interpretação de uma expressão, mas principalmente saber como esta interpretação é construída a partir da interpretação das subexpressões daquela expressão, de acordo com o princípio da composicionalidade.

No trajeto da sua elaboração, além da inadequação em relação à composicionalidade, mostrou-se também que execução parcial da redução-β e da aplicação realmente criavam termos-λ "bizarros" e completamente equivocados para representar a interpretação semântica das subexpressões. Assim, o custo da suposta simplicidade implementacional é o sacrifício da sintaxe original dos termos-λ, que precisou ser torcida para aceitar termos que não seriam inicialmente bem-formados.

Para um lingüista que se interessa não apenas pelo produto lingüístico, mas principalmente pelo processo lingüístico, esse tipo de defeito é crucial. Assim, fica justificada a recusa dos "benefícios substanciais em relação à simplicidade do programa".

Como foi observado em diversos momentos do texto, a interpretação semântica construída nos analisadores apresentados aqui é bastante ingênua, já que não lida com ambigüidades de escopo e de leituras distribuídas e coletivas, para relembrar duas passagens anteriores, além de desprezar a semântica temporal, para citar algo que ainda não tinha sido mencionado. No entanto, acredita-se que o analisador final proposto aqui possa ser facilmente adaptado para executar as análises apropriadas destes e de outros fenômenos semânticos, já que a implementação é mais coerente com as escolhas do cálculo de predicados e do cálculo-λ para construir a representação semântica, como é praxe na Semântica Formal. Sendo assim, a próxima etapa mais evidente para o desenvolvimento deste analisador seria a inclusão de procedimentos para produzir as interpretações ambíguas devidas à relação entre os escopos dos quantificadores, o que permitiria inclusive uma comparação com as implementações em [12, 1]; no entanto, qualquer fenômeno interpretativo descrito explicitamente através de operações bem definidas podem ser facilmente adaptados no analisador proposto.

Referências

[1] Patrick Blackburn and Johan Bos. *Representation and Inference for Natural Language.* CSLI, Stanford, 2005.

[2] Ronnie Cann. *Formal Semantics.* Cambridge University Press, Cambridge, 1993.

[3] Bob Carpenter. *Type-Logical Semantics*. The MIT Press, Cambridge, 1997.

[4] Genaro Chierchia and Sally McConnel-Ginet. *Meaning and Grammar*. The MIT Press, Cambridge, 1990.

[5] Michael A. Covington. *Natural Language Processing for Prolog Programmers*. Prentice Hall, Englewood Clifs, 1994.

[6] Ray C. Dougherty. *Natural Language Computing*. Lawrence Erlbaum, Hillsdale, 1994.

[7] David R. Dowty, Robert E. Wall, and Stanley Peters. *Introduction to Montague Semantics*. Kluwer, Dordrecht, 1981.

[8] Gerard Gazdar and Chris Mellish. *Natural Language Processing in Prolog*. Addison-Wesley, Wokingham, 1989.

[9] Clive Matthews. *An Introduction to Natural Language Processing through Prolog*. Longman, London, 1998.

[10] Richard A. O'Keefe. *The Craft of Prolog*. The MIT Press, Cambridge, 1990.

[11] Gabriel A. Othero and Sérgio M. Menuzzi. *Lingüística Computacional: Teoria & Prática*. Parábola Editorial, São Paulo, 2005.

[12] Fernando C. N. Pereira and Stuart M. Shieber. *Prolog and Natural-Language Analysis*. CSLI, Stanford, 1987.