REPRESENTAÇÃO SEMÂNTICA EM ANALISADORES GRAMATICAIS

Luiz Arthur Pagani (UFPr)

1 Introdução

Ainda hoje, analisadores gramaticais (*parsers*) são, na verdade, analisadores sintáticos, porque neles o resultado do processamento é apenas uma representação da estrutura sintática da expressão analisada (como em Dougherty (1994), Matthews (1998) e Othero & Menuzzi (2005)), apesar de não ser nova a preocupação de também construir uma interpretação semântica (como em Pereira & Shieber (1987: 91-114) e Covington (1994: 196-256)). Tampouco 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 alguns analisadores, implementados em Prolog, que produzem uma representação semântica para cada expressão analisada. A partir desta apresentação, serão discutidas questões relativas tanto à representação semântica quanto à implementação computacional. Em relação à representação semântica, discutiremos a escolha de representar um nome próprio como *Pedro* através de `(pedro^S)^S` (pela compatibilização com os quantificadores generalizados). Já sobre a implementação computacional, discutiremos o uso de variáveis do Prolog como variáveis da interpretação semântica e a maneira como a redução-β é definida – o que causa uma unificação de variáveis incomum do ponto de vista semântico durante o processamento (ainda que o resultado final seja coerente).

2 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. Observemos, então, o funcionamento de dois analisadores desse tipo: o de Pereira & Shieber e o de Covington.

2.1 Pereira & Shieber

Um dos analisadores mais antigos capaz de lidar com a interpretação é o de Pereira & Shieber (1987: 102-103):

```
:- op(500, xfy, &).
:- op(510, xfy, =>).
s(S) --> np(VP^S), vp(VP).
np(NP) --> det(N1^NP), n(N1).
np((E^S)^S) --> pn(E).
vp(X^S) --> tv(X^IV), np(IV^S).
vp(IV) --> iv(IV).
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)).
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)).
```

Este programa foi transcrito quase integralmente; retiramos dele apenas a alternativa de acrescentar no SN uma oração relativa. No resto, o programa tem a mesma estrutura do original; dele, três características merecem ser observadas.

A primeira delas, que também é a causa das outras duas, é que um determinante 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 execução parcial da aplicação e da redução: ao invés de uma implicação entre duas aplicações (`P@X` e `Q@X`), ocorre uma implicação entre dois termos (`S1` e `S2`) nos quais as aplicações e suas respectivas reduções acontecem à esquerda do operador-λ (`X^S1` e `X^S2`). A explicação (Pereira & Shieber 1987: 100) é a de que as fórmulas para os quantificadores generalizados precisam 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)`; portanto, a abstração de `Y^program(Y)` nos leva a `(X^P)^(X^Q)^all(X,P=>Q)` como representação de *every*.

A segunda característica é a interpretação dos nomes próprios. Devido à execução parcial da redução e da aplicação, a representação do significado do nome próprio Shrdlu acaba ficando como `(shrdlu^P)^P`, ao invés da mais simples `P^P@shrdlu` (*λP.(P shrdlu)*). 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 (Pereira & Shieber 1987: 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`.

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, o funcionamento do programa é o esperado; mas como veremos depois, há motivo para desconfiarmos dessa codificação através de "expressões bizarras".)

Finalmente, a terceira característica está relacionada aos verbos transitivos diretos: *wrote* é representado como `X^Y^wrote(X,Y)`, ao invés de `Y^X^wrote(X,Y)`, que seria o esperado para *λy.λx.wrote(x,y)*. Aqui, a inversão na ordem das variáveis é afetada pela execução parcial e pela facilidade para a unificação da regra 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)`.

Apesar disto, o programa constrói as representações corretas (que os autores chamam de forma lógica (*logical form*)). Em Pereira & Shieber (1987: 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 não apresenta 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 (Pereira & Shiber 1987: 104-114) explicam a construção de um analisador para esta ambigüidade de escopo, mas não podemos nos ocupar com isso agora, pois nos afastaria da questão discutida aqui. Vejamos então um outro analisador que constrói a representação semântica da expressão analisada.

2.2 Covington

Um segundo analisador gramatical que constrói uma representação semântica é o de Covington (1994: 207-210):

```
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 Pereira & Shieber, Covington não nos oferece uma listagem completa; assim, o programa 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 codificação 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, 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 nas definições dos predicados `some/3` e `all/3`.) 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 nos verbos transitivos diretos: no programa de Covington, o significado de *chased* é `Y^X^chase(X,Y)` (com as variáveis na mesma ordem que *λy.λx.chase(x, y)*), ao contrário do que acontecia no programa de Pereira & Shieber, em que *wrote* era representado como `X^Y^wrote(X,Y)`. Ainda que também não caracterize nenhuma diferença de estratégia, a solução de Covington é mais coerente com o termo-λ correspondente. Para isso, no entanto, foi preciso "espalhar" mais a solução de Pereira & Shieber: a regra para o 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 léxico).

Em relação à representação 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 apresenta uma

solução – ele apenas sugere, como exercício (Covington 1994: 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 *every dog chased a cat*, este analisador 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 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 equivalentes, como se pode constatar fazendo `S=Sem` e `VP=X^Scope`.)

3 Analisador com histórico derivacional

Segundo Pereira & Shieber (1987: 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 a escolha não causa danos, apesar de não ser "logicamente correta", e efetivamente esta solução tem sido adotada mesmo em analisadores mais recentes, como em Blackburn & Bos (2005: 73).[1] 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 inadequada em relação às etapas de sua construção; depois disso, apresentaremos um analisador que executa a derivação respeitando o princípio da composicionalidade.

3.1 Adaptando os analisadores para a derivação

O programa abaixo é, basicamente, o analisador de Covnigton adaptado para apresentar seu histórico derivacional. Para um lingüista, a apresentação da derivação ajuda a compreender as etapas da 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).
categoria(sv).
categoria(det).
categoria(n).
categoria(v).
apresenta(N, [X, Y]) :-
   tab(N),
   write(X),
```

[1] Infelizmente, não foi possível incluir uma avaliação do analisador 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).

```
   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` apenas facilitam a inicialização e a apresentação da análise (formam uma interface), e por isso não serão comentados.
Esta versão consiste apenas na adaptação do argumento para a representação semântica, de forma que nele se documente sua derivação. Assim, a representação de *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 *Pedro*, percebemos que ela construiria a representação `[(p^P)^P,[p,[]]]`; ela nos diz que a denotação agora é `(p^P)^P`, obtida a partir de `[p,[]]`.
Isto pode ser facilmente constatado rodando as diretivas abaixo:

```
?- analisa([pedro]).
sn:
(pedro^A)^A
6
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,

com um recuo, ele nos informa a sua denotação `((pedro^A)^A)`; e na terceira, com 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 *dormiu*.
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
```

Coerentemente, o analisador nos informa que a expressão é uma sentença (`s`), cujo significado é `dormir(pedro)`. Mas também diz que este significado é obtido composicionalmente de `(pedro^dormir(pedro))^dormir(pedro)` e de `pedro^dormir(pedro)`; estas representações corresponderiam a *λ(λpedro:dormir(pedro)):dormir(pedro)* e *λpedro:dormir(pedro)*, respectivamente – que, como admitiram Pereira & Shieber, não constituem termos-λ bem formados.[2]

3.2 Construindo uma derivação mais coerente

No programa a seguir, finalmente, implementa-se a sugestão de Pereira & Shieber, empregando as variáveis do Prolog apenas para guiar a introdução das variáveis das representações; estas variáveis da metalinguagem da representação terão o formato `x(N)`, para qualquer número inteiro `N` (as do Prolog, portanto, são variáveis da metalinguagem para a construção da representação; ambas 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, ^).
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),
```

[2] 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.

```
   !.
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).
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)}.
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].
reduz_lista(Lista, Reduzido) :-
   Lista = [Termo|_],
   reduz_termo(Termo, Resultado),
   reduz_lista([Resultado|Lista], Reduzido),
   !.
reduz_lista(Resultado, Resultado).
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).
reduz_termo(Var^Termo, Var^Resultado) :-
   reduz_termo(Termo, Resultado).
reduz_termo(Termo1 /\ Termo2, Termo1 /\ Resultado) :-
   reduz_termo(Termo2, Resultado).
reduz_termo(Termo1 /\ Termo2, Resultado /\ Termo2) :-
   reduz_termo(Termo1, Resultado).
reduz_termo(Termo1 => Termo2, Termo1 => Resultado) :-
   reduz_termo(Termo2, Resultado).
```

```
reduz_termo(Termo1 => Termo2, Resultado => Termo2) :-
   reduz_termo(Termo1, Resultado).
reduz_termo(algum(Var, Termo), algum(Var, Resultado)) :-
   reduz_termo(Termo, Resultado).
reduz_termo(qualquer(Var, Termo), qualquer(Var, Resultado)) :-
   reduz_termo(Termo, Resultado).
substitui(x(M), x(N), Termo2, Termo3) :-
   ( x(M) \= x(N)
   -> Termo3 = x(M)
   ; Termo3 = Termo2
   ).
substitui(Termo3, _, _, Termo3) :-
   atom(Termo3).
substitui(Termo1a@Termo1b, Var, Termo2, Termo3a@Termo3b) :-
   substitui(Termo1a, Var, Termo2, Termo3a),
   substitui(Termo1b, Var, Termo2, Termo3b).
substitui(Var1^Termo1, Var, Termo2, Termo3) :-
   ( Var1 = Var
   -> Termo3 = Var1^Termo1
   ; substitui(Termo1, Var, Termo2, Resultado),
     Termo3 = Var1^Resultado
   ).
substitui(Termo1a /\ Termo1b, Var, Termo2, Termo3a /\ Termo3b) :-
   substitui(Termo1a, Var, Termo2, Termo3a),
   substitui(Termo1b, Var, Termo2, Termo3b).
substitui(Termo1a => Termo1b, Var, Termo2, Termo3a => Termo3b) :-
   substitui(Termo1a, Var, Termo2, Termo3a),
   substitui(Termo1b, Var, Termo2, Termo3b).
substitui(algum(Var1, Termo1), Var, Termo2, Termo3) :-
   ( Var1 = Var
   -> Termo3 = algum(Var1, Termo1)
   ; substitui(Termo1, Var, Termo2, Resultado),
     Termo3 = algum(Var1, Resultado)
   ).
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 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[3] 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, `F(a1, a2)` passa a ser `F@a1@a2`.[4]
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 ? ;
```

[3] O par de acumulador é uma técnica de programação 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 O'Keefe (1990: 22).
[4] Como o operador `@` é definido com associatividade à esquerda (`yfx`), `f@a1@a2` é equivalente a `(f@a1)@a2`.

```
no
```

Quando ela é um SN (cuja análise é obtida com `sn(1,X,Sem,[pedro],[])`), o seu significado é representado internamente como `[x(1)^x(1)@p,[p,[]]]`; já quando ela 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 fica numa lista, na qual se registra seus constituintes, de forma que o termo-λ mais simples que representa o significado da expressão sempre ocupa a primeira posição, todas as combinações de significados 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 *Pedro dormiu*, o resultado da redução seria:

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

Em relação às outras implementações, 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 redução a ser feita, a segunda cláusula 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 de um termo, a única diferença nesse padrão é a redução de `(Var^Termo)@Arg` – *(λVar.Termo Arg)* – em que se substitui a variável `Var` pelo termo `Arg` no termo `Termo`. A substituição é feita por 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, podemos ver abaixo a análise de *todo menino saltou um buraco*.[5]

```
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))
         b
yes ;
no
```

Nela, vê-se recorrentemente o termo mais simples que representa o significado da expressão e, logo abaixo, os termos mais complexos que foram reduzidos até chegarem àquele primeiro (antecedidos por <==). Depois, os subtermos da composição são apresentados na mesma indentação. Assim, observando os termos indentados mais à direita no exemplo, 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))`.

Ao contrário dos analisadores anteriores, todas as etapas de construção apresentam termos-λ coerentes com o que Pereira & Shieber qualificaram como "apresentados abstratamente no cálculo".

---

[5] Como esperado, sentenças mais simples, como *Pedro dormiu* e *todo menino dormiu*, e expressões de outros tipos também são analisadas, exatamente como no exemplo da análise de *Pedro*, acima.

4 Conclusão

Com a apresentação de antigos analisadores que executam a interpretação semântica, e de sua adaptação para aplicativos que demonstrem sua composicionalidade, espera-se ter explicitado uma diferença essencial que distingue uma boa parte dos trabalhos em Lingüística Computacional; a saber, 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 da representação final. No analisador proposto aqui, esta identificação não se sustenta, porque a preocupação é a de saber como esta interpretação é construída a partir da interpretação das subexpressões de uma expressão, de acordo com o princípio da composicionalidade.

No trajeto da sua elaboração, além da inadequação em relação à composicionalidade, mostrou-se também que a 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 dos termos-λ, que precisou ser "torcida" para aceitar termos que não seriam bem-formados.

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

Convém lembrar ainda que a interpretação construída nos analisadores apresentados aqui é bastante ingênua, já que não lida com ambigüidades de escopo e de leituras distribuídas e coletivas, além de desprezar a semântica temporal, por exemplo. No entanto, acredita-se que o analisador 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 Pereira & Shieber (1987) e Blackburn & Bos (2005); no entanto, qualquer fenômeno interpretativo descrito explicitamente através de operações bem definidas se adaptam facilmente ao analisador proposto.

Referências

Patrick Blackburn and Johan Bos. Representation and Inference for Natural Language. CSLI, Stanford, 2005.

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

Ray C. Dougherty. Natural Language Computing. Lawrence Erlbaum, Hillsdale, 1994.

Clive Matthews. An Introduction to Natural Language Processing through Prolog. Longman, London, 1998.

Richard A. O'Keefe. The Craft of Prolog. The MIT Press, Cambridge, 1990.

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

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