DANIEL MARTINESCHEN

ANALISADOR GRAMATICAL POR DESLOCAMENTO E REDUÇÃO PARA GRAMÁTICAS CATEGORIAIS

Trabalho de Graduação apresentado como requisito parcial à obtenção do grau de Bacharel em Ciência da Computação. Departamento de Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.
Orientador: Prof. Luiz Arthur Pagani, Departamento de Lingüística, UFPR

CURITIBA

Fevereiro de 2004

AGRADECIMENTOS

Agradeço primeiramente a quem me iniciou no campo da lingüística e do PLN: ao Prof. Borges, por ter me apresentado a Gramática Categorial e me aberto possibilidades para estudo sobre esse tema. Sem as suas aulas de lingüística em 2001 eu não teria seguido este caminho e provavelmente não teria escrito este trabalho. E ao prof. Michel Gagnon, com quem aprendi Prolog e que me incentivou a permanecer na área de PLN, apesar de não ter sido possível um trabalho conjunto.

Em segundo, mas não menos importante, agradeço ao meu orientador, Prof. Luiz Arthur, pela sua paciência, pelo incentivo e pelos trabalhos conjuntos, e principalmente por não ter perdido a fé no meu potencial, permanecendo como orientador até o final, mesmo com meu não cumprimento de alguns prazos.

Agradeço também a Renato Kajita, coordenador da minha equipe na Pastoral da Criança, por me liberar do estágio nos dias cruciais para que eu pudesse dar conta deste trabalho.

Agradeço grandemente também aos meus amigos e amigas de turma e do grupo PET, que sempre me incentivaram a manter meus estudos em PLN (apesar de muitos me acharem maluco por causa disso), e cuja convivência me ajudou a não enlouquecer com o ritmo frenético do último semestre do curso, recheado de disciplinas, estágios, projetos e trabalhos de graduação...

Agradeço por fim a meus familiares, que (apesar de também me acharem meio maluco) sempre me motivaram a manter os estudos em lingüística e que sempre me apoiaram em momentos de *stress*.

RESUMO

A Gramática Categorial (GC) é um programa de investigação lingüística, desenvolvido para análise de língua natural, que tem ganhado destaque nos últimos anos devido ao seu rigor formal e à indissociabilidade entre a sintaxe e semântica presentes nela. Diversas implementações em computador são feitas com o propósito de refinar a GC de modo a identificar possíveis falhas formais e para abranger um conjunto maior de fenômenos lingüísticos.

O trabalho apresentado no presente texto é fruto do estudo e implementação de um analisador gramatical por deslocamento e redução (*shift-reduce parser*) em Prolog, tomando como base a implementação apresentada por Covington (1994). O *parser* foi adaptado para trabalhar com uma versão da gramática categorial chamada GC Livre de Cohen (1967, *apud* Wood, 1993, p. 35), para dar conta de um pequeno fragmento da língua portuguesa.

Num primeiro momento é apresentado o que é um *parser*, sua relação com gramáticas e as abordagens de análise mais freqüentemente usadas. Depois é feita uma apresentação da GC Livre, sobre a qual se baseou o desenvolvimento do *parser*. Problemas decorrentes de implementação como a aplicação recursiva das regras categoriais unárias são discutidos, e possíveis soluções são propostas. Ao fim do trabalho são indicadas propostas de trabalhos futuros.

SUMÁRIO

LISTA DE FIGURAS

LISTA DE PROGRAMAS

CAPÍTULO 1

INTRODUÇÃO

Com a disseminação dos computadores, suas utilizações têm se expandido a praticamente todos os setores do conhecimento, e não é diferente com o campo da Lingüística. Ao estudar uma teoria gramatical, o interesse de escrever um programa de computador que permita analisar sentenças da língua que essa gramática modela é imediato. Implementado em computador, esse programa permite que se analise o comportamento da gramática, dando pistas de possíveis incorreções ou imprecisões na definição formal ou de que a técnica utilizada na implementação necessita de refinamentos ou de análise mais cuidadosa do problema.

Com o aumento do poder de processamento dos computadores e com o desenvolvimento de linguagens de programação de alto nível de abstração, a implementação de *analisadores gramaticais* ou *parsers* se tornou acessível a pessoas que não são da área da computação, notadamente lingüistas e lógicos, que estão preocupados com a eficácia, a precisão e o formalismo das teorias gramaticais que estudam e desenvolvem, e não com os detalhes de implementação. O exemplo mais popular dessas linguagens, usada tanto por lingüistas quanto por cientistas da computação, é a linguagem Prolog, que implementa um fragmento do cálculo de predicados de primeira ordem, e provê um meio simples de implementar *parsers*, acessível a pessoas que não têm treinamento formal em programação.

A Gramática Categorial, desenvolvida a partir o fim do século XIX com os trabalhos de Gottlob Frege sobre lógica matemática, propõe um tratamento das expressões das línguas naturais diferente das teorias tradicionais. As expressões de uma língua (no nosso caso, o Português) são encaradas como *funções* e *argumentos*, e as relações entre essas expressões não são mais determinadas por uma estrutura externa (como na Gramática de Estrutura Sintagmática), mas sim iguais às relações entre funções e argumentos usuais na matemática. A análise de expressões passa a ser, então, a manipulação algébrica de funções e argumentos, utilizando operações matemáticas como aplicação e composição de funções e lógico-matemáticas como elevação de um tipo para um predicado de ordem superior.

No século XX, com os trabalhos de Ajdukiewicz (1935), Bar-Hillel (1953), Bar-Hillel *et al.* (1960), Lambek (1958) e muitos outros, a GC foi amadurecendo, e atualmente é uma teoria cuja popularidade tem aumentado e cujas idéias têm repercussão em muitas outras teorias gramaticais. A pesquisa em GC tem se disseminado nos últimos anos, e uma variedade enorme de problemas e de línguas diferentes têm sido tratados, como se pode conferir em Wood (1993), uma boa introdução às GCs.

O trabalho apresentado no presente texto é fruto do estudo e implementação de um analisador gramatical por deslocamento e redução (*shift-reduce parser*, em inglês) em Prolog, tomando como base a implementação apresentada por Covington (1994), que utiliza a Gramática de Estrutura Sintagmática ou *Phrase Structure Grammar*. O *parser* foi adaptado para trabalhar com uma versão da gramática categorial chamada GC Livre de Cohen (1967, *apud* Wood, 1993, p. 35), para dar conta de um pequeno fragmento da língua portuguesa.

Primeiramente vamos fazer uma discussão sobre o que é um *parser*, a sua relação com as teorias gramaticais e as abordagens de análise mais freqüentemente utilizadas. Depois, fazemos uma apresentação da GC Livre, sobre a qual se baseou o desenvolvimento do *parser*. Problemas decorrentes de implementação, como a aplicação recursiva e irrestrita das regras categoriais unárias, são discutidos, e soluções para eles são apresentadas, juntamente com uma reflexão sobre seus efeitos colaterais na flexibilidade do sistema. No final do trabalho listamos uma série de trabalhos e extensões que representariam melhoras no *parser*, além de idéias para disseminação do seu uso e de chegar a um sistema robusto e utilizável.

CAPÍTULO 2

ANALISADORES GRAMATICAIS

Neste capítulo falaremos sobre Analisadores Gramaticais (*parsers*). Primeiramente daremos uma definição do que são e como são compostos. Num segundo momento, discutiremos a relação forte entre *parsers* e gramáticas, falando sobre a motivação de fazer implementações de *parsers* em computadores. Depois, apresentamos as abordagens de análise que são utilizadas, suas restrições e aplicações. Por último apresentamos o analisador por deslocamento e redução.

2.1 O que é um analisador gramatical?

Um analisador gramatical é um algoritmo computacional que, baseado nas regras de uma gramática, permite analisar uma expressão da língua modelada pela gramática e responde se essa expressão é ou não gramatical (ou, alternativamente, se ela pertence ou não à língua), e como produto adicional infere a estrutura sintática (e possivelmente a semântica também) dessa expressão. Um exemplo de representação de estrutura sintática é a árvore, como ilustrado na Figura 2.1.

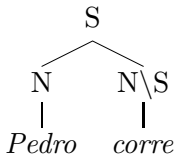

Figura 2.1: Representação em árvore da sentença *Pedro corre*

Um *parser* é diferente de um *reconhecedor* no sentido de que o último somente faz uma parte do trabalho, que é responder se uma expressão é ou não bem formada, sem inferir sua estrutura sintática. Neste trabalho trataremos apenas de *parsers*.

2.2 *Parsers* e gramáticas

Uma gramática é um sistema formal que define quais são as expressões de uma determinada língua. Dado um fragmento de uma língua, gramáticas são desenvolvidas para tratar de fenômenos lingüísticos de interesse, seja este específico para uma aplicação ou uma reflexão teórica sobre determinado problema. Como sistema formal, abstrato, uma gramática parece dar conta do fragmento de língua escolhido e dos fenômenos considerados.

Implementado em uma linguagem de programação de alto nível de abstração (no caso de línguas naturais a mais usada é a linguagem Prolog), um *parser* vai procurar reconhecer e inferir a estrutura gramatical de uma expressão seguindo as regras definidas na gramática. Se a gramática estiver mal definida, o *parser* não terá o comportamento esperado, dando respostas erradas. Assim, indícios de que a gramática deve ser revisada ou que sua implementação deve ser refeita aparecem. Além disso, um *parser* serve para obter análises que não são óbvias à

primeira vista, aumentando o entendimento sobre a gramática e os fenômenos que ela pretende explicar.

Esse caráter experimental da implementação de um *parser* é interessante no campo de lingüística e de PLN, pois permite a evolução das teorias lingüísticas e das técnicas de implementação. Porém, há aplicações para as quais um *parser* é parte integrante e básica, e não objeto de reflexão teórica. Em compiladores para linguagens de programação, por exemplo, a implementação de um *parser* para a gramática da linguagem em questão é indispensável, e deve ser eficiente e rápida.

2.3 Estratégias de análise

Falamos da relação entre *parsers* e gramáticas, e nos referimos ao fato de que um *parser* segue uma determinada estratégia para efetuar a análise gramatical. Nesta seção vamos falar com mais detalhe sobre elas.

Para construir a estrutura gramatical de uma expressão, existem duas estratégias: a abordagem descendente ou *top-down* e a ascendente ou *bottom-up*. Vamos falar com mais detalhe sobre cada uma delas. Vamos adotar a árvore como representação para entender o funcionamento dessas estratégias.

2.3.1 *Top-down*

Como dito acima, na seção 2.1, uma possível representação para a estrutura gramatical de uma expressão é a árvore. A árvore apresenta de forma clara a relação de dependência entre as subpartes da expressão, e permite visualizar a aplicação das regras gramaticais.

Na estratégia descendente ou *top-down* o *parser* vai construir a árvore começando do nó raiz, e vai aplicar as regras gramaticais em seqüência até conseguir achar uma *derivação* que permita dizer que a expressão avaliada é bem-formada. A seqüência de construção da árvore é da esquerda para a direita, razão pela qual os *parsers top-down* também são chamados de *left-right, top-down parsers* (analisadores descendentes processando da esquerda para a direita). Tomemos por exemplo a gramática de estrutura sintagmática mostrada na Figura 2.2.

$$S \rightarrow SN\ SV \tag{2.1}$$
$$SN \rightarrow Det\ N \tag{2.2}$$
$$SN \rightarrow N \tag{2.3}$$
$$SV \rightarrow V \tag{2.4}$$
$$SV \rightarrow V\ SN \tag{2.5}$$
$$Det \rightarrow o \tag{2.6}$$
$$N \rightarrow Pedro \tag{2.7}$$
$$N \rightarrow menino \tag{2.8}$$
$$V \rightarrow corre \tag{2.9}$$

Figura 2.2: Exemplo de Gramática de Estrutura Sintagmática

Uma seqüência de análise *top-down* para a sentença *Pedro corre* é ilustrada na figura 2.3. Nela, pode-se notar como a árvore é construída de cima para baixo, tomando como ponto de partida o nó S. As regras gramaticais são aplicadas considerando os nós da árvore na direção esquerda-direita, e são escolhidas de acordo com a ordem em que estão escritas na gramática.

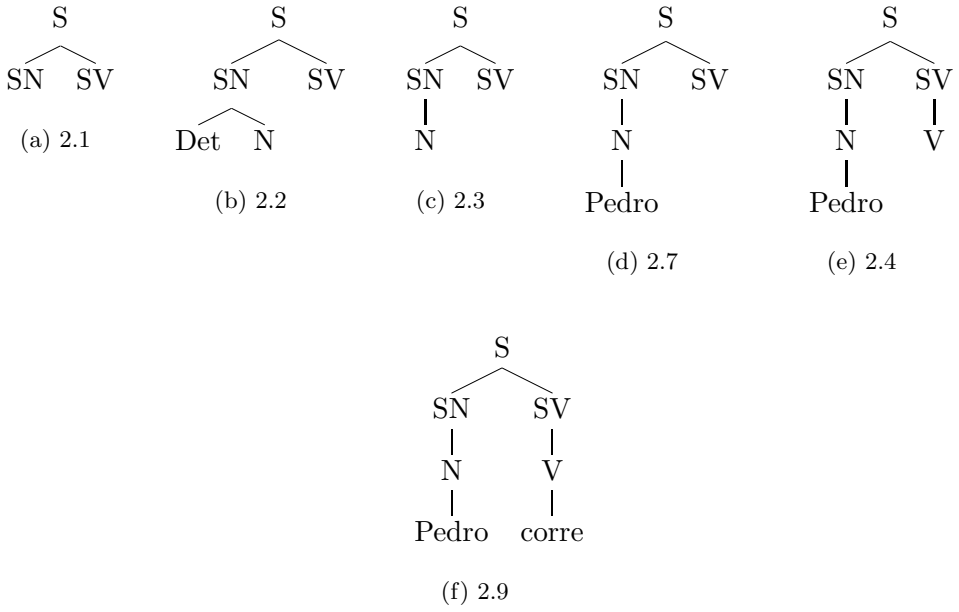

Figura 2.3: Análise *top-down* da sentença *Pedro corre* passo a passo

O *parser* aplica primeiro a regra 2.1, que diz como um S pode ser formado. Depois, vai escolher entre as regras 2.2 e 2.3 para expandir o não-terminal SN. A primeira regra disponível é a 2.2, que leva a um insucesso pois a primeira palavra da sentença não é um *Det* (determinante). A regra 2.3 é escolhida, e ao nó SN da árvore adicionamos um nó filho N que, aplicando a regra 2.7, corresponderá à palavra *Pedro*. Agora o *parser* parte para expandir os nós que ainda não foram expandidos. O próximo nó é o SV, filho do nó-raiz S. Novamente, há duas regras a escolher, e o *parser* pega a primeira que encontra, a regra 2.4. Ao expandir o nó V, pela regra 2.9, é lida a palavra *corre* e a sentença é então reconhecida.

O principal problema do *parser top-down* é que no caso de haver regras recursivas à esquerda (ex. $SN \rightarrow SN\ SP$) ele pode entrar em *loop* infinito. Se a gramática não for cuidadosamente construída, para retirar a recursividade à esquerda nas regras, o *parser* pode ter esse comportamento. Outra característica desse *parser* é que ele assume que na entrada deve ter uma sentença, e orienta toda a análise com o objetivo de encontrar uma sentença bem-formada.

2.3.2 *Bottom-up*

Uma estratégia alternativa à *top-down* é a abordagem *bottom-up* ou *ascendente*. Nessa abordagem, o *parser* faz a análise começando com as palavras da entrada, e vai construindo a árvore sintática "de baixo para cima", ou seja, faz a análise a partir das *folhas* da árvore, ao invés de começar pela raiz.

Usando a mesma gramática da figura 2.2, vamos analisar a sentença *Pedro corre* utilizando a abordagem *bottom-up*. Essa análise está ilustrada na figura 2.4. Um detalhe a ser observado é que nessa análise somente foram consultadas as regras que realmente se aplicavam à sentença, evitando tentativas desnecessárias.

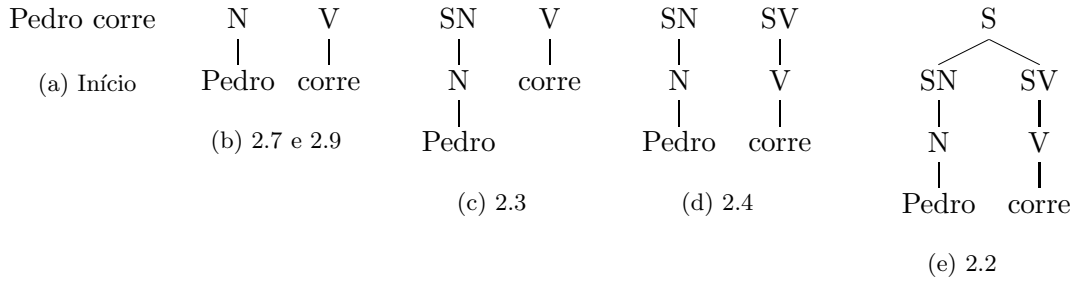

Figura 2.4: Análise *bottom-up* da sentença *Pedro corre* passo a passo

2.3.3 Vantagens e desvantagens das duas abordagens

Ambas as abordagens acima possuem vantagens e desvantagens, e a escolha por uma delas deve considerar as suas características e limitações. A principal vantagem da abordagem *top-down* é ser facilmente implementável, principalmente se for utilizado um recurso como a DCG (*Definite Clause Grammar*) do Prolog. Associada a uma técnica de previsão, a abordagem *top-down* pode ser bastante eficiente, evitando retrocessos na análise (Aho *et al.*, 1988, p. 181-195). Porém, ela pode levar o *parser* a executar infinitamente, se as regras gramaticais tiverem recursão à esquerda.

A abordagem *bottom-up* tem a vantagem de não entrar em *loop* infinito com regras gramaticais recursivas à esquerda. Além disso, as regras aplicadas são escolhidas de acordo com a expressão analisada, evitando tentativas que podem levar a insucessos, e a árvore sintática é construída de maneira precisa. Se há mais de uma regra que reduz uma determinada subexpressão, o *parser* é capaz de apresentar todas as possibilidades, porém não as tenta cegamente como na estratégia *top-down*. O que pode levar um *parser bottom-up* a refazer uma análise inteira é a possível ambigüidade do léxico, ou seja, a possibilidade de a cada palavra poderem ser associadas várias categorias. Outra característica da abordagem *bottom-up* é que o *parser* não fica limitado a procurar somente por sentenças, mas procura descobrir se a expressão é conexa, ou seja, se ela se reduz a uma única categoria, não importa qual seja.

Há ainda estratégias de análise mistas, que integram técnicas de ambas as abordagens, como o *left-corner parsing*. Maiores detalhes podem ser encontrados em Aho *et al.* (1988, cap. 4).

2.4 Analisador por deslocamento e redução

Nesta seção apresentamos uma técnica utilizada na construção de *parsers*, a análise por deslocamento e redução ou *shift-reduce parsing*. Essa técnica é apresentada na literatura (Aho *et al.*, 1988, p. 195-257) como a implementação padrão para analisadores *bottom-up*. Ela provê

um algoritmo para a aplicação das regras gramaticais de forma a evitar os problemas de *loops* infinitos que ocorrem com a estratégia *top-down*.

A técnica do deslocamento e redução é caracterizada pelo uso de uma estrutura de dados auxiliar, a *pilha*, e por duas operações: o deslocamento (*shift*) e a redução (*reduce*). Na operação de deslocamento uma palavra da expressão lida é inserida na pilha. A operação de redução analisa as (sub-)expressões que estão no topo da pilha e procura aplicar alguma regra gramatical, *reduzindo* a expressão que está na pilha à categoria que está do lado esquerdo da regra aplicada.

O *parser* por deslocamento e redução, então, lê a entrada palavra por palavra, empilhando cada uma e tentando aplicar as regras gramaticais. Ele procede alternando deslocamentos e reduções até que a entrada inteira seja lida e não possam mais ser feitas reduções. Se, ao final da análise, a entrada tiver sido lida inteira e houver somente uma categoria no topo da pilha, então o *parser* reporta sucesso. Quando a entrada tiver sido completamente lida e não houver reduções a serem feitas, mas ainda sobrar mais de uma categoria no topo da pilha, o *parser* reporta um erro, indicando que a expressão não é bem-formada.

Um *parser* por deslocamento e redução, por adotar a abordagem *bottom-up*, não enfrenta problemas com regras recursivas à esquerda como na análise *top-down*. Porém surgem outros problemas, que se tornam evidentes quando passamos à implementação. O *parser* pode não ter meios para decidir entre deslocar uma nova expressão e reduzir o topo da pilha (conflito *deslocamento-redução*) ou entre duas regras a aplicar quando efetua uma redução (conflito *redução-redução*).

Para resolver os conflitos, é necessário escolher, em cada conflito, qual das duas operações será executada. No desenvolvimento de *parsers* para compiladores, em que as gramáticas utilizadas são livres de contexto (*context-free grammars*) e a linguagem de programação utilizada é a linguagem C, é utilizada uma tabela de predição, que indica a ação a tomar em cada situação do *parser*, considerando o elemento do topo da pilha e a próxima palavra da entrada. Essa técnica, também chamada de análise LR(k) ou LR(k) *parsing* (k é a quantidade de palavras de *lookahead*, ou seja, palavras da entrada que não foram lidas e são consideradas para decidir o rumo da análise), é descrita com detalhes em Aho *et al.* (1988, seção 4.7).

Na Seção 4.1 discutiremos como esses conflitos são tratados na implementação do *parser* em linguagem Prolog.

CAPÍTULO 3

GRAMÁTICA CATEGORIAL

Neste capítulo é apresentada a GC, tomando como base principalmente os textos de Borges Neto (1999) e Wood (1993). Apresentaremos primeiramente o sistema AB de Ajdukiewicz (1935) e Bar-Hillel (1953) (*apud* Wood, 1993), que provê somente a regra de aplicação funcional e depois a GC Livre de Cohen (1967, *apud* Wood, 1993, p. 35), que aumenta o sistema AB com as regras de associatividade, composição e elevação de tipo, e sobre a qual foi feita a implementação do *parser*. Para apresentar exemplos de análises gramaticais utilizamos o diagrama de Prawitz (1965).

3.1 O que é Gramática Categorial

De acordo com Borges Neto (1999, cap. 1), a GC "é um programa de investigação lingüística que procura dar conta da sintaxe, da semântica e de aspectos como morfologia e fonologia das línguas naturais". A GC foi desenvolvida desde o princípio direcionada ao tratamento das línguas naturais, e é portanto tópico da área de Processamento da Língua Natural (PLN). A GC se destaca de outras teorias gramaticais (como a Gramática Gerativa ou a Gramática de Estrutura Sintagmática) por três características principais:

1. A forma como os constituintes das expressões são entendidos se baseia nos conceitos lógico-matemáticos de função e argumento. Segundo essa abordagem as expressões da língua se combinam segundo as regras que regem o comportamento de funções lógico-matemáticas: aplicação de função a argumentos, composição de funções, etc.

2. As informações sintáticas, semânticas e outras que se façam necessárias são concentradas diretamente nos itens lexicais, o que caracteriza a GC como uma teoria "essencialmente *lexicalista*"(Borges Neto, 1999, cap. 1, grifo do autor).

3. A estreita ligação entre sintaxe e semântica, o que geralmente é entendido como se houvesse um homomorfismo entre os dois níveis (ou seja, para cada representação semântica pode haver várias sintáticas, e vice-versa). Além disso, tanto sintática quanto semanticamente as relações entre os constituintes são estritamente composicionais, ou seja, seguem o *Princípio de Frege*, segundo o qual o significado de uma expressão é função dos significados de suas partes ligados por uma relação.

Além disso, a GC é originalmente interdisciplinar, com contribuições de áreas como matemática, lógica, filosofia, lingüística e computação (nem todas essas com relações óbvias entre si), o que a torna ainda mais atrativa do ponto de vista da interação entre áreas que ela incentiva.

3.1.1 O caráter funcional da GC

O lógico alemão Gottlob Frege, no fim do século XIX, foi o primeiro a estender as noções matemáticas de *função e argumento* para a lógica matemática. Sua proposta inicial (Frege,

1967) era analisar proposições da lógica em termos de funções e argumentos, em vez de sujeito e predicado. Dessa forma, as proposições da lógica podiam ser analisadas de inúmeras formas diferentes, decorrentes de diferentes decomposições dessas proposições em relações entre funções e argumentos.

Frege (1891 *apud* Wood, 1993) mostra como podemos dividir sentenças em geral em duas partes: uma completa ou *saturada* e outra incompleta ou *insaturada*. Uma frase como *Pedro corre* pode ser dividida nas partes *Pedro* e (___*corre*), sendo a segunda parte insaturada[1]. Essa expressão insaturada tem um lugar vazio, uma valência, e quando esse "buraco" é preenchido por um nome próprio a sentença obtém sentido completo. A essa expressão "insaturada" Frege dá o nome de *função*, e à expressão que a complementa de *argumento*. Essa mesma idéia está presente na GC, e ficará mais visível quando introduzirmos a noção de categoria e a regra de aplicação funcional.

Desse modo, Frege propõe uma nova maneira de interpretar expressões de uma língua, entendendo-as como relações entre funções e argumentos. Os lógicos Curry (1930; 1961 *apud* Wood, 1993, p.20) e Schönfinkel (1924 *apud* Wood, 1993, p.20) também fizeram trabalhos explorando a perspectiva funcional da linguagem, e contribuíram para o desenvolvimento das GCs.

3.1.2 A noção de categoria

Sabemos que, na matemática, funções são mapeamentos entre conjuntos, sendo o conjunto de origem chamado *domínio* e o de destino *imagem* ou *contra-domínio*. Se entendermos que um elemento que pertence a um conjunto C é do *tipo* X, podemos entender que funções são operações que levam (ou transformam) elementos de um tipo para outro. Assim, para que o conceito de função e argumento possa ser usado na análise das línguas naturais é necessário atribuir categorias às palavras da língua, de modo que algumas destas sejam argumentos (pertençam ao conjunto domínio de alguma função) e outras sejam funções (tomam expressões de algum tipo e devolvem expressões de algum outro tipo).

As categorias são, portanto, de dois tipos: *básicas* (ou *fundamentais* ou *argumentais*) e *funtoras* (ou *derivadas*). As do primeiro tipo são as que são sempre argumentos em qualquer expressão; as do segundo tipo são funções, embora também possam exercer o papel de argumento[2]. O conjunto de categorias é definido recursivamente como segue. Partimos de um conjunto de categorias básicas que contém as categorias S e N (respectivamente a categoria das *sentenças* e a categoria dos *nomes*).

Para construirmos categorias complexas ou funtoras utilizamos o conectivo "|". Dadas duas categorias quaisquer X e Y, podemos construir a categoria $X|Y$, que representa uma função que toma como argumento uma expressão de categoria Y e retorna uma de categoria X. Aplicando essa regra recursivamente, podemos obter *infinitas* categorias: $X|Y$, $Y|(X|Z)$, $(X|Y)|(Z|W)$, etc (usando parênteses para evidenciar o conectivo principal quando necessário). O conectivo "|" pode ser entendido como uma "barra de fração", fração na qual o numerador é o resultado e o denominador é o argumento da função, por exemplo, em $\frac{X}{Y}$. Essa notação "vertical", porém,

[1] Frege indica que uma expressão é insaturada colocando um "__"(sublinhado ou *underscore*) na posição em que deve estar a expressão que a completa.

[2] Onde se lê *expressão funtora* e *expressão argumental* entenda-se *expressão de categoria funcional* e *expressão de categoria básica ou argumental*, respectivamente.

é inadequada por ocupar muito espaço no texto, portanto é feita uma "rotação de 90 graus" na fração e ela é grafada na forma "horizontal".

Com o conectivo "|" teremos um sistema *unidirecional*, que não leva em conta a ordem das palavras nas sentenças. Essa abordagem foi adotada por Ajdukiewicz (1935) e se justificava por uma transformação das sentenças da língua para a notação polonesa, na qual os funtores sempre precedem seus argumentos. Por exemplo, a expressão matemática "5+3×4" na notação polonesa ficaria "+(5)(×(3)(4))" (parênteses adicionados para evidenciar a abrangência dos funtores).

Porém, essa abordagem unidirecional, baseada no uso da notação polonesa, não é adequada para o tratamento das línguas naturais, pois não consegue captar as diversas ordens de constituintes que aparecem nas estruturas sintáticas das expressões, como demonstrado por Bar-Hillel (Bar-Hillel (1953), Bar-Hillel *et al.* (1960) *apud* (Wood, 1993, p. 22)). Ele propôs o que ele chama de Gramática Categorial Bidirecional, na qual os conectivos formadores de categorias funtoras indicam a direção em que o argumento deve ser tomado. Esses conectivos são / e \, que podem ser vistos como "inclinações" da barra de fração. O primeiro indica que o argumento deve ser procurado à direita, enquanto que o segundo diz que o mesmo deve ser procurado à esquerda da expressão funtora.

Assim, podemos definir formalmente o conjunto *infinito* de categorias CAT, mostrado na Definição 3.1.1.

Definição 3.1.1 (Categorias) *Seja* CATBAS $= \{N, S\}$ *o conjunto de categorias básicas. O conjunto de categorias* CAT *é definido recursivamente como:*

- *se* $X \in$ CATBAS*, então* $X \in$ CAT*;*
- *se* $X, Y \in$ CAT*, então* X/Y *e* $X\backslash Y \in$ CAT*.*

Alguns autores distinguem, no conjunto de categorias básicas, nomes próprios (N) de nomes comuns (Nc); apesar disso, trabalharemos somente com as categorias S e N, já que não é objetivo deste estudo determinar o conjunto das categorias básicas para o Português (o que deve ser feito por um lingüista).

Com a definição acima já podemos atribuir categorias às palavras de um fragmento do português. Vamos atribuir a categoria N ao nome *Pedro* e a categoria $N\backslash S$ (uma função procurando um N à sua esquerda para devolver uma expressão de tipo S) à palavra *corre*. Assim, podemos olhar a sentença *Pedro corre* em função das categorias das palavras que a compõem:

Pedro	corre
N	$N\backslash S$

A notação utilizada acima para grafar as categorias é a chamada notação de Lambek, chamada por Wood de "resultado no topo"(Wood, 1993, p. 14). Existe ainda a notação de Steedman (ou "resultado primeiro", id.), na qual a categoria resultante sempre fica à esquerda e o que muda é somente a direção do conectivo. Por exemplo, a categoria $N\backslash S$, na notação de Lambek, é grafada $S\backslash N$ na notação de Steedman. Neste trabalho será usada somente a notação de Lambek, por ser mais clara e por ser mais freqüentemente usada em trabalhos sobre GC.

3.1.3 Regra de aplicação funcional

Para podermos fazer análises das sentenças falta definirmos as regras gramaticais, que vão reger a combinação das expressões categorizadas. A regra que vamos introduzir aqui, a mais básica delas, é a de *aplicação funcional*. Uma metáfora freqüentemente utilizada para descrever a aplicação funcional é a idéia de "cancelamento" ou "simplificação" do denominador de uma fração, como em $\frac{X}{Y} \bullet Y = X$. Essa regra é mais formalmente descrita na definição 3.1.2.

Definição 3.1.2 (Aplicação Funcional) *Sejam* $X, Y \in \mathrm{CAT}$

- $X/Y : f \cdot Y : a \rightarrow X : f(a)$ *(aplicação à direita)*
- $Y : a \cdot Y\backslash X : f \rightarrow X : f(a)$ *(aplicação à esquerda)*

Na definição acima e nas definições subseqüentes incluímos a representação semântica correspondentes às categorias: a expressão funtora de tipo X/Y denota a função f, enquanto que a expressão de tipo X denota um argumento a. A aplicação de f a a é notada $f(a)$.

Com a regra de aplicação acima já podemos analisar uma sentença simples do português, mostrada na Figura 3.1.

$$\dfrac{\mathrm{N}\dfrac{\text{Pedro}}{p}_{\text{lex}} \quad \mathrm{N\backslash S}\dfrac{\text{corre}}{C}_{\text{lex}}}{\mathrm{S}\dfrac{\text{Pedro corre}}{C(p)}\ \text{AplE}}$$

Figura 3.1: Análise de *Pedro corre* utilizando somente aplicação funcional.

A regra mostrada na figura acima é a de aplicação à esquerda, na qual a categoria de *Pedro*, N, serve de argumento para a categoria de *corre*, $N\backslash S$. O resultado é a expressão correspondente à sentença original *Pedro corre* com categoria resultante S (sentença)

3.2 A GC livre

A GC apresentada até aqui é chamada de *modelo AB*, em homenagem a Ajdukiewicz e Bar-Hillel, no qual a única regra gramatical é a de aplicação funcional. Abaixo apresentamos as regras de composição funcional, associatividade (*associativity*) e de elevação de tipo (*type raising*), que, junto com a regra de aplicação, constituem a chamada GC Livre (*Free Categorial Grammar*), de Cohen (1967, *apud* Wood, 1993, p. 35).

3.2.1 Associatividade ou permutação

A segunda regra da GC Livre é a de associatividade (ou *permutação*). Segundo essa regra, é possível alterar a ordem na qual um funtor procura o seu argumento. Por exemplo, um verbo transitivo direto como *ama*, que tem categoria $(N\backslash S)/N$, pode se combinar primeiro com o sujeito (à esquerda) para depois se combinar com o objeto direto (à direita), ou se combinar primeiro com o objeto, resultando em um verbo intransitivo, e depois se combinar com o sujeito

para formar uma sentença. Para isso é necessário que se mude a ordem em que *ama* toma seus argumentos, ou seja, a categoria de *ama* passa a ser $N\backslash(S/N)$. A regra é descrita formalmente na Definição 3.2.1, como apresentado por Wood (1993, p. 37).

Definição 3.2.1 (Associatividade) *Sejam* $X, Y, Z \in \mathrm{CAT}$

- $X\backslash(Y/Z) : \lambda v_x.\lambda v_z.f(v_z)(v_x) \rightarrow (X\backslash Y)/Z : \lambda v_z.\lambda v_x.f(v_z)(v_x)$ *(associatividade para a esquerda)*
- $(X\backslash Y)/Z : \lambda v_z.\lambda v_x.f(v_z)(v_x) \rightarrow X\backslash(Y/Z) : \lambda v_x.\lambda v_z.f(v_z)(v_x)$ *(associatividade para a direita)*

Aqui utilizamos o cálculo lambda para a representação semântica das expressões[3]. Na definição acima fica clara a indissociabilidade entre os níveis sintático e semântico: a mudança na ordem dos argumentos na categoria sintática é realizada também na representação semântica. Nota-se porém que a ordem dos argumentos dentro da função não é alterada, somente a ordem em que são preenchidos.

Consideremos como exemplo a sentença *João ama Maria*. *João* e *Maria* são palavras de categoria N e *ama* tem categoria $(N\backslash S)/N$. Poderíamos fazer uma análise usando somente a regra de aplicação, como na Figura 3.2.

$$\dfrac{\mathrm{N}\dfrac{\text{João}}{j}_{\text{lex}} \quad N\backslash S\dfrac{(N\backslash S)/N\dfrac{\text{ama}}{A}_{\text{lex}} \quad \mathrm{N}\dfrac{\text{Maria}}{m}_{\text{lex}}}{\text{ama Maria}}\,\text{AplD}}{\mathrm{S}\dfrac{\text{João ama Maria}}{A(m)(j)}\,\text{AplE}}$$

Figura 3.2: Análise de *João ama Maria* utilizando somente aplicação funcional.

Utilizando a regra de associatividade temos a possibilidade de fazer uma análise alternativa da mesma sentença, permitindo uma outra combinação das expressões. Veja essa análise na Figura 3.3.

$$\dfrac{\dfrac{\mathrm{N}\dfrac{\text{Joao}}{j}_{\text{lex}} \quad N\backslash(S/N)\dfrac{(N\backslash S)/N\dfrac{\text{ama}}{A}_{\text{lex}}}{\text{ama}}\,\text{PerD}}{\mathrm{S/N}\dfrac{\text{João ama}}{\lambda v_z.A(v_z)(j)}\,\text{AplE} \quad \mathrm{N}\dfrac{\text{Maria}}{m}_{\text{lex}}}}{\mathrm{S}\dfrac{\text{João ama Maria}}{A(m)(j)}\,\text{AplD}}$$

Figura 3.3: Análise de *João ama Maria* utilizando associatividade.

[3]Para uma introdução mais completa ao cálculo lambda consultar Cresswell (1988), Dowty *et al.* (1981) ou Hindley e Seldin (1986).

Na análise acima, o verbo *ama* com a categoria permutada pode se combinar primeiro com o sujeito, *João*, para depois se combinar com *Maria*. Esta regra aumenta o poder combinatório da álgebra das expressões da língua em estudo.

3.2.2 Composição

A terceira regra gramatical a ser introduzida é a de composição funcional. Na matemática, se temos uma função f que toma um argumento do mesmo tipo que o retornado por uma outra função g, podemos construir uma nova função h tal que $h(x) = f(g(x))$, que vai tomar um argumento do tipo que g precisa e retornar um valor do tipo que f retorna.

A regra categorial para a composição se baseia exatamente nessa idéia, e é apresentada formalmente na Definição 3.2.2.

Definição 3.2.2 (Composição) *Sejam* $X, Y, Z \in \mathrm{CAT}$

- $X/Y : f \cdot Y/Z : g \rightarrow X/Z : \lambda v_z.f(g(v_z))$ *(composição para a direita)*
- $Z\backslash Y : g \cdot Y\backslash X : f \rightarrow Z\backslash X : \lambda v_z.f(g(v_z))$ *(composição para a esquerda)*

Para exemplificar o uso da composição vamos considerar a mesma sentença do exemplo anterior, porém atribuindo a categoria $S/(N\backslash S)$ à palavra *João*.[4] Observe que a análise apresentada na Figura 3.4 é totalmente *incremental*, ou seja, ela é processada na ordem em que as palavras são lidas — da esquerda para a direita.

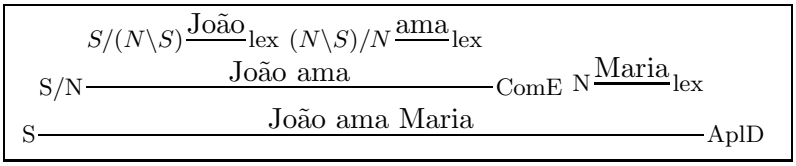

Figura 3.4: Análise de *João ama Maria* utilizando somente composição funcional.

3.2.3 Elevação de tipo ou promoção

A quarta e última regra da GC Livre é a elevação de tipo ou *promoção* (*type raising* ou *type lifting*), que "permite que transformemos qualquer categoria atômica X em um funtor procurando um funtor que procura por um X"(Wood, 1993, p.42). Mais do que isso, a regra de elevação de tipo faz com que uma expressão de uma categoria arbitrária se torne uma função (ou conjunto) de segunda ordem, indicando um "conjunto de conjuntos". Por exemplo, tomemos o nome *Pedro*, com categoria N obtida do léxico. Se aplicarmos a regra de elevação, poderemos obter a categoria $S/(N\backslash S)$, que pode ser interpretada como "o conjunto de predicados que se aplicam a Pedro" ou "o conjunto dos conjuntos a que o indivíduo Pedro pertence". Vejamos mais formalmente essa idéia na Definição 3.2.3.

[4] Omitimos por enquanto a representação semântica de $S/(N\backslash S)$, que será esclarecida na seção 3.2.3 com a regra de elevação de tipo.

Definição 3.2.3 (Elevação de tipo) *Seja* $X \in \mathrm{CAT}$.

- $X : a \rightarrow Y/(X\backslash Y) : \lambda P.P(a)$ *(elevação para a direita)*
- $X : a \rightarrow (Y/X)\backslash Y : \lambda P.P(a)$ *(elevação para a esquerda)*

Na definição acima podemos perceber os efeitos da elevação de tipo na representação semântica da expressão. Uma expressão de categoria X, que denota o indivíduo representado pela constante a, passa a ter a nova categoria $Y/(X\backslash Y)$, denotando a função $\lambda P.P(a)$, que representa o conjunto de predicados P tais que $P(a)$ é verdadeiro.

Vamos exemplificar com a sentença *Pedro corre*, já utilizada anteriormente neste texto. Na análise apresentada na Figura 3.5, o nome *Pedro* terá sua categoria elevada e a regra de aplicação funcional será utilizada para que esta se combine com a categoria do verbo *corre*. Note que a interpretação semântica da expressão continua a mesma, $C(p)$, depois das devidas reduções relativas ao cálculo lambda.

$$\dfrac{\dfrac{\mathrm{N}\dfrac{\text{Pedro}}{p}\,\text{lex}}{S/(N\backslash S)\dfrac{\text{Pedro}}{\lambda P.P(p)}\,\text{Elev}} \quad N\backslash S\dfrac{\text{corre}}{C}\,\text{lex}}{\mathrm{S}\dfrac{\text{Pedro corre}}{C(p)}\,\text{AplD}}$$

Figura 3.5: Análise de *Pedro corre* utilizando elevação de tipo.

Com isto terminamos a apresentação da GC Livre, e já temos uma noção básica de seu funcionamento. Descrições mais detalhadas da GC, juntamente com perspectivas históricas e relatos de desenvolvimentos e trabalhos em andamento podem ser encontrados em Wood (1993) e Borges Neto (1999). Referências para os trabalhos de Frege, Ajdukiewicz, Bar-Hillel e outros são encontradas nessas obras.

CAPÍTULO 4

ANALISADOR POR DESLOCAMENTO E REDUÇÃO PARA GCS

Neste capítulo apresentamos a implementação de um *parser* por deslocamento e redução para gramáticas categoriais em Prolog. O interpretador Prolog utilizado é o `swi-prolog`, da Universidade de Amsterdam na Holanda. Ele pode ser obtido a partir do endereço *http://www.swi-prolog.org*.

Após uma introdução, apresentamos a técnica chamada DCG, utilizada na escrita do *parser*. Os predicados do programa são explicados em detalhe, e resultados experimentais são apresentados. Os problemas encontrados durante a implementação e as soluções adotadas são apresentados ao final do capítulo.

4.1 Introdução

Como dito na Seção 2.2, uma das motivações para implementar um *parser* é testar se a teoria gramatical em estudo realmente dá conta dos fenômenos que pretende explicar, e propor melhoramentos nela ou nas técnicas de implementação. Com as gramáticas categoriais não é diferente: como as GCs propõem uma abordagem matemática das línguas naturais, é interessante examinar o seu comportamento, implementando um *parser* que rode em uma máquina de inferência, como um interpretador Prolog. Com seus diversos recursos, como unificação, ferramentas como a DCG e facilidade de programação, Prolog é a linguagem de programação "padrão" utilizada por lingüistas para implementar gramáticas, e foi a adotada neste trabalho.

Este trabalho se baseou em uma adaptação feita por Pagani (2003) no analisador por deslocamento e redução apresentado por Covington (1994, p. 155–159), originalmente desenvolvido para a gramática de estrutura sintagmática. Além da adaptação para funcionar com a gramática categorial, algumas alterações foram feitas para corrigir alguns problemas e para representar melhor alguns conceitos lingüísticos.

Na implementação em Prolog, os conflitos do *parser* por deslocamento e redução são resolvidos de uma maneira diferente do que quando usamos uma linguagem de programação imperativa como a linguagem C. A solução depende diretamente de como são implementados os predicados que fazem o deslocamento e a redução, e da ordem em que são escritas as regras gramaticais. Essa questão será discutida com detalhe na Seção 4.3.2.

4.2 O *parser* original

O *parser* original Covington (1994) apresentava um comportamento indesejado quando se solicitava o *backtrack* para obter as demais análises possíveis: as reduções vão sendo desfeitas, apresentando soluções incorretas, como observado em Pagani (2003). A primeira alteração feita no programa original é mostrada no Programa 4.1: o segundo argumento (que corresponde à pilha) da segunda cláusula do predicado `shift_reduce` foi modificado para ser uma lista contendo o próprio resultado. O código do *parser* original está listado no Apêndice A.

```
desloc_reduz(S, Pilha, Resultado) :-
  deslocamento(Pilha, S, NovaPilha, S1),
  redução(NovaPilha, PilhaReduzida),
  desloc_reduz(S1, PilhaReduzida, Resultado).

desloc_reduz([], [Resultado], Resultado).
```

Programa 4.1: Alteração no predicado `shift_reduce`

Essa mudança, além de resolver o problema de desfazer as reduções, implementa um conceito lingüístico interessante, o de conexidade. Segundo esse conceito, uma expressão é bem-formada se suas subpartes estiverem inter-relacionadas umas com as outras; usando a metáfora da árvore, isso equivale a dizer que todos os nós terminais precisam estar ligados a uma única raiz. Com essa alteração não é necessário especificar para o *parser* quais nós podem ser raiz de uma árvore: se, no resultado da análise de uma expressão, restar somente um elemento na pilha — não importa de que categoria seja —, essa expressão é bem-formada (Pagani, 2003).

A segunda modificação foi unir a inserção lexical com o deslocamento, eliminando a regra correspondente a ela e alterando o predicado `shift` para fazer ele mesmo a busca pelo item no léxico. Com essa alteração, conseguimos uma distinção mais precisa dos níveis de informação processada pelo *parser*, o que é lingüisticamente interessante: na implementação original a pilha recebia tanto expressões quanto categorias, enquanto que com essa mudança a pilha só recebe categorias[1]. A nova versão do predicado `shift`, chamado de `an_lex`, é listado no Programa 4.7.

E a terceira alteração, que é mais um exercício do que uma mudança mais significativa, foi implementar o *parser* utilizando o recurso da DCG, que descrevemos na próxima seção.

4.2.1 DCG – *Definite Clause Grammar*

A DCG (Gramática de Cláusula Definida — *Definite Clause Grammar*) é um recurso presente na maioria dos interpretadores Prolog e que permite a transcrição de gramáticas de estrutura sintagmática diretamente para código Prolog. Esse recurso foi apresentado pela primeira vez em Pereira e Warren (1980), e popularizado em Pereira e Shieber (1987). No Programa 4.2 transcrevemos a gramática da figura 2.2 (p. 4) para Prolog usando a DCG.

[1]Na verdade, nesta implementação a pilha não recebe categorias, mas sim uma estrutura de dados mais complexa que representa uma expressão. Essa expressão será descrita abaixo, na Seção 4.3.1.

```
s --> sn, sv.
sn --> det, n.
sn --> n.
sv --> v.
sv --> v, sn.
det --> [o].
n --> [pedro].
n --> [menino].
v --> [corre].
```

Programa 4.2: Gramática da Figura 2.2 em DCG

Os itens lexicais em DCG são representados entre colchetes. Um código em DCG para poder ser usado, precisa ser pré-processado e traduzido para predicados do Prolog, que utilizarão o recurso de diferença de listas. No Programa 4.3 mostramos como o código DCG do Programa 4.2 é traduzido para predicados Prolog.

```
s(L0, L) :- sn(L0,L1), sv(L1,L).
sn(L0, L) :- det(L0,L1), n(L1,L).
sn(L0, L) :- n(L0, L)
sv(L0, L) :- v(L0, L1).
sv(L0, L) :- v(L0, L1), n(L1, L).
det([o|L], L).
n([pedro|L], L).
n([menino|L], L).
v([corre|L],L).
```

Programa 4.3: Programa 4.2 traduzido de DCG para predicados do Prolog

Para analisar a sentença *Pedro corre* basta escrever a seguinte consulta:

```
?- s([pedro, corre], []).
```

Como estamos tratando de analisadores gramaticais, precisamos apresentar a estrutura gramatical da expressão avaliada. Uma forma de fazer isso é acrescentar um argumento aos predicados utilizados, e neste argumento (que será um funtor da linguagem Prolog) construir a estrutura gramatical. Um funtor que corresponderia à regra $S \rightarrow SN\ SN$ seria, por exemplo, `s(sn(...),sv(...))`. Assim, o Programa 4.2 pode ter as regras aumentadas de um argumento, como mostrado no Programa 4.4.

Agora, para analisar a sentença *Pedro corre*, a consulta deve ser:

```
?- s(X, [pedro, corre], []).
```

e a variável `X` será instanciada com o termo `s(sn(n(pedro)),sv(v(corre)))`, que representa a árvore mostrada na Figura 2.3(f) à página 5.

Um último recurso da DCG é a possibilidade de inserir dentro de uma especificação DCG chamadas a predicados do Prolog que não recebam como argumento as listas incluídas pela

```
s(s(SN,SV)) --> sn(SN), sv(SV).
sn(sn(Det,N)) --> det(Det), n(N).
sn(sn(N)) --> n(N).
sv(sv(V)) --> v(V).
sv(sv(V,SN)) --> v(V), sn(SN).
det(det(o)) --> [o].
n(n(pedro)) --> [pedro].
n(n(menino)) --> [menino].
v(v(corre)) --> [corre].
```

Programa 4.4: Programa 4.2 com predicados aumentados de um argumento

DCG. Para tanto, basta escrever o predicado na posição desejada e colocá-lo entre chaves ("{" e "}").

4.2.2 Questões sobre o uso da DCG em implementação de *parsers*

Como vimos, a DCG é um recurso que permite transcrever diretamente gramáticas de estrutura sintagmática para o Prolog. Como a gramática é transformada diretamente em predicados do Prolog, as regras são aplicadas de acordo com a ordem que o interpretador segue. O interpretador Prolog processa os predicados seguindo a abordagem *top-down* e da esquerda para a direita. Logo, quando transcrevemos uma gramática para DCG temos um *parser top-down* de imediato, sem precisar construir predicados adicionais. Porém esse *parser* vai sofrer os mesmos problemas característicos de busca em profundidade, descritos na seção 2.3.1, com regras recursivas à esquerda.

Essa é uma das motivações para implementar o *parser bottom-up* com a DCG: fugir dos problemas característicos da abordagem *top-down* e ainda aproveitar a facilidade de uso da DCG, que agiliza a codificação de predicados (não necessariamente relacionados a regras gramaticais) que processam seqüencialmente listas de elementos.

4.3 Implementação

Nesta seção descrevemos os predicados do *parser* modificado para funcionar com a gramática categorial. Descrevemos primeiramente a estrutura de dados utilizada para representar as expressões da pilha, depois passamos a uma análise detalhada de cada predicado do *parser* e, por fim, explicamos como é feita a apresentação das análises.

4.3.1 Estrutura de dados

Na Seção 2.4 vimos que o *parser* por deslocamento e redução utiliza como estrutura auxiliar uma *pilha*, que em Prolog é implementada como uma lista. Uma das alterações no *parser* original foi juntar em uma única operação a análise lexical e o deslocamento, para fazer com que a pilha

tratasse somente com categorias (que é o mais natural, já que o *parser* não deve se preocupar com as palavras, mas sim com seus tipos sintáticos e as respectivas combinações).

A estrutura de dados adotada para representar os elementos da pilha é a chamada Estrutura de Características Padronizada (*typed-feature structure*) ou ECP, que nada mais é do que um termo do Prolog com o seguinte formato:

```
ecp(Palavras, Categoria, Denotacao, Regra, Constituintes)
```

onde:

Palavras: Lista com as palavras que compõem a expressão.

Categoria: Categoria da expressão resultante do uso da regra indicada em **Regra** aplicada aos argumentos em **Constituintes**.

Denotação: Representação semântica, expressa na linguagem do cálculo-lambda.

Regra: A regra, aplicada à lista de constituintes, para obter a categoria da expressão

Constituintes: Lista contendo ECPs que representam as expressões que serviram de argumento para a regra **Regra**.

Além das regras gramaticais apresentadas no Capítulo 3, o *parser* registra também a operação de inserção lexical. Para cada regra e para a inserção lexical são usadas constantes do Prolog, como mostrado abaixo:

Inserção lexical: `i_lex`.

Aplicação: `apl_e` (esquerda) e `apl_d` (direita).

Composição: `comp_e` (esquerda) e `comp_d` (direita).

Associatividade: `per_e` (esquerda) e `per_d` (direita).

Promoção: `pro_e` (esquerda) e `pro_d` (direita)

Por exemplo, a palavra *corre* seria representada pela seguinte ECP:

```
ecp([corre], n\s, [lmbd, x12, ['C', x12]], i_lex, []).
```

Termos lambda

Vimos acima que o terceiro elemento da ECP é a representação semântica da expressão, expressa na linguagem do cálculo lambda. Para este trabalho utilizamos um fragmento do cálculo lambda, cuja representação em Prolog descrevemos nesta Seção. Um *termo-lambda* pode ser descrito pela gramática da Figura 4.1 (*const* representa uma constante e *var* uma variável).

Para representar os termos-lambda em Prolog utilizamos listas, e cada componente do termo-lambda é representado como segue. Constantes do cálculo lambda são representadas por constantes do Prolog; o ligador de variáveis (λ) é representado por `lmbd`; os conectivos "e" e "se ... então" ($\wedge$ e $\rightarrow$) são representados por "`/\`" e "`=>`", respectivamente; e os quantificadores $\forall$ e $\exists$ são representados por "`todo`" e "`algum`", respectivamente.

$$\begin{aligned}
\textbf{termo} &\Rightarrow var \\
\textbf{termo} &\Rightarrow const \\
\textbf{termo} &\Rightarrow termo \wedge termo \\
\textbf{termo} &\Rightarrow termo \rightarrow termo \\
\textbf{termo} &\Rightarrow termo(termo) \\
\textbf{termo} &\Rightarrow \lambda var[termo] \\
\textbf{termo} &\Rightarrow \exists var[termo] \\
\textbf{termo} &\Rightarrow \forall var[termo]
\end{aligned}$$

Figura 4.1: Gramática dos termos-lambda

Deixamos a descrição das variáveis por último para poder justificar o modo como implementamos a redução-β. Para representar as variáveis do cálculo lambda em Prolog utilizamos constantes, como `v10, v20`, especificadas diretamente no léxico, e não variáveis do Prolog. Adotamos essa opção para que o recurso de unificação do Prolog não fizesse ligação indevida de variáveis, e para que tivéssemos inteiro controle sobre as operações efetuadas pela redução-β. Porém, essa solução apresenta problemas, pois se uma mesma palavra ocorrer duas vezes na mesma expressão o predicado que efetua a redução-β pode fazer substituições incorretas. Essa questão será discutida com mais detalhe na seção 4.4.

4.3.2 O predicado `conexo`

Para realizar análises utilizamos o predicado `conexo/4`, listado no Programa 4.5[2].

```
conexo(Pilha, Resultado) -->
  an_lex(Pilha, NovaPilha),
  {conecta(NovaPilha, PilhaConexa)},
  conexo(PilhaConexa, Resultado).

conexo([Resultado], Resultado) --> [].
```

Programa 4.5: Predicado `conexo/4`

Os argumentos de `conexo/4` são:

- PILHA: a pilha do *parser*;
- RESULTADO: variável que vai conter a ECP representando a expressão analisada;
- o terceiro e o quarto argumentos são as listas da DCG.

[2]O terceiro e quarto argumentos não aparecem na definição, pois representam a lista de palavras e a lista restante, que são criadas automaticamente quando usamos DCG.

A primeira cláusula chama os predicados correspondentes ao deslocamento e à redução, `an_lex` e `conecta`, respectivamente. Note que os nomes dos predicados foram alterados do original (`shift` e `reduce`, respectivamente) para que refletissem as atividades que desempenham, ou seja: `an_lex` faz a análise lexical e `conecta` verifica se a expressão é conexa.

A diferença no modo de chamar os predicados deve-se ao uso da DCG para implementá-los: o predicado `conecta` tem somente dois argumentos, e para impedir que na tradução de DCG para Prolog sejam adicionados os dois argumentos padrão (lista de palavras e resto) ele é colocado entre chaves ("{" e "}"). O predicado `an_lex`, que tem quatro argumentos, precisa ter somente os dois primeiros indicados.

A segunda cláusula é a condição de parada: se a pilha contiver somente uma expressão (precisamente a análise da expressão) e não restar mais nenhuma expressão na entrada, a análise foi bem-sucedida e o *parser* retorna o resultado.

Para tornar o uso do *parser* mais cômodo foi criado o predicado `conexo/1`, listado no Programa 4.6. Ele vai chamar o predicado `conexo/4`, mostrado acima, vai imprimir o resultado na tela (uma ECP, apresentada na Seção 4.3.1) e vai falhar, forçando o *backtrack* de `conexo/4`, a fim de obter todas as análises possíveis. Outros predicados foram criados para apresentar os resultados de forma mais legível, e são apresentados na Seção 4.3.7.

```
conexo(Expressao) :-
  conexo([], Resultado, Expressao, []),
  nl,
  write(Resultado),
  nl,
  fail.
```

Programa 4.6: Predicado `conexo/1`

Conflitos

Na Seção 2.4, falamos sobre os conflitos que um *parser* por deslocamento e redução pode encontrar durante as análises. Na implementação em Prolog apresentada neste trabalho, os conflitos do tipo *deslocamento-redução* não ocorrem, devido à maneira como foi escrito o predicado `conexo/4`: esse predicado executa um deslocamento (para cada palavra da entrada) e depois tenta aplicar todas as reduções possíveis nas expressões que estão na pilha. Somente quando não conseguir mais reduzir a pilha é que o *parser* lê a próxima palavra da entrada. Dessa maneira, o *parser* nunca precisa decidir entre ler a próxima palavra e aplicar uma regra, pois essas ações são feitas separadamente e não interferem uma com a outra.

Já os conflitos do tipo *redução-redução* são resolvidos de uma forma diferente: os predicados referentes às regras são chamados na ordem em que estão escritos no programa Prolog. O *parser* chama os predicados seqüencialmente, e, se algum deles retornar sucesso (i.e. a regra é aplicável), um *ponto de escolha* é adicionado a partir dele, para que o *parser* possa efetuar o *backtrack*.

Na mesma seção citada acima, falamos sobre tabelas de previsão que são criadas para resolver os conflitos. Como a GC provê uma álgebra combinatória livre, com flexibilidade de

combinação das categorias em uma análise, é importante que o *parser* teste todas as possibilidades de aplicação de regras para conseguir todas as análises possíveis. Desse modo, o uso de um mecanismo de previsão (comumente chamado de *oráculo*) poderia limitar as possibilidades de análise.

Nesta implementação do *parser* as ações de deslocamento e de redução estão, portanto, desacopladas ou independentes uma da outra, solucionando de maneira quase artificial os conflitos. No Capítulo 5 discutimos a possibilidade de uma nova implementação, diferente da deste trabalho, em que possam ocorrer os conflitos discutidos acima.

4.3.3 Deslocamento

O predicado que faz o deslocamento e a inserção lexical é `an_lex/4`, listado no Programa 4.7.

```
an_lex(Pilha, [ecp([Pal], Cat, Denot, i_lex, []) | Pilha]) -->
  [Pal],
  {palavra(Pal, Cat, Denot)}.
```

Programa 4.7: Predicado `an_lex/4`

O primeiro argumento de `an_lex/4` é a pilha do *parser*, repassada pelo predicado `conexo/4`. O segundo argumento é a pilha resultante, em cujo topo está a ECP representando a palavra lida da entrada. A linha `[Pal]` lê a próxima palavra da entrada, e a linha seguinte consulta o léxico para obter a categoria e a denotação da palavra. Neste ponto será adicionado um ponto de escolha, se tivermos atribuído mais de uma categoria ao item lexical.

Como vimos no início do capítulo, a inserção lexical foi unida com a operação de deslocamento para que a pilha operasse com elementos homogêneos: logo que a palavra é lida é empilhada uma ECP que a representa, e a partir de então o *parser* opera somente com ECPs.

4.3.4 Redução

O predicado que faz a redução é `conecta/2`, listado no Programa 4.8.

```
conecta -->
  regra,
  conecta.
conecta --> [].
```

Programa 4.8: Predicado `conecta/2`

Escrito completamente em DCG, o predicado, na primeira cláusula, tenta aplicar alguma regra gramatical, chamando o predicado `regra`, e se chama recursivamente no final, para tentar novas reduções na pilha. A segunda cláusula é a condição de parada do predicado, que é satisfeita quando não houver mais nenhuma redução a ser feita.

4.3.5 Regras gramaticais

No Capítulo 3 apresentamos a GC livre, com as regras de aplicação, associatividade, composição e elevação de tipo. Para cada regra há uma cláusula do predicado `regra`, tanto para a variante direita quanto para a esquerda. Nesta seção as descrevemos uma a uma.

Aplicação

A cláusula referente à aplicação funcional para a direita é listada no Programa 4.9.

```
regra([Arg2, Arg1 | Resto], [Resultado | Resto]) :-
  Arg1 =..  [ecp, Expr1, X/Y, Denot1, _, _],
  Arg2 =..  [ecp, Expr2, Y, Denot2, _, _],
  append(Expr1, Expr2, Expr),
  reduz([Denot1, Denot2], Normal),
  Resultado =.. [ecp, Expr, X, Normal, apl_d, [Arg1, Arg2]].
```

Programa 4.9: Cláusula de `regra/2` para aplicação para a direita

A primeira coisa a notar é a ordem em que as variáveis `Arg1` e `Arg2` se unificam com os elementos do topo da pilha. Isso se deve ao fato de que as expressões na pilha ficam em ordem inversa à ordem da expressão de entrada[3].

As duas primeiras linhas decompõem os dois elementos do topo da pilha (que são ECPs) usando o operador `=..` ("Univ", que serve para construir ou decompor termos), para extrair as respectivas expressões, categorias e denotações dessas ECPs,[4] e para simultaneamente testar se a aplicação funcional pode ser utilizada. Para que possamos fazer a aplicação funcional da categoria de `Arg1` à de `Arg2` é preciso que a primeira seja da forma X/Y e a segunda Y. X e Y são variáveis do Prolog, e o recurso de unificação do Prolog faz boa parte do trabalho aqui. A unificação impede o uso da regra em basicamente duas situações:

- se a Pilha contiver somente um elemento ela não será casará com o primeiro argumento da cláusula;
- se as respectivas categorias dos elementos do topo da pilha não satisfizerem as condições supracitadas a regra não será usada.

Tendo passado pelo teste acima, as expressões são concatenadas para formar uma nova expressão (instanciada em `Expr`), e passamos para a redução-β das denotações, que é feita pelo predicado `reduz/2`. A variável `Normal` contém a nova denotação, após as devidas reduções. Por fim, a ECP da expressão resultante é construída na variável `Resultado` usando o operador `=..`, da seguinte forma: a *expressão* resultante é a concatenação das duas expressões argumento; sua *categoria* é X (resultante da aplicação de X/Y a Y); sua *denotação* é o resultado da aplicação da denotação de `Arg1` na de `Arg2`; a *regra* é `apl_d` e a lista de *constituintes* contém as duas ECPs que estavam no topo da pilha.

[3] Para a sentença *Pedro corre*, por exemplo, a pilha ficaria (simplificando) `[corre,pedro]`.

[4] Os dois componentes restantes da ECP (regra e lista de constituintes) não são considerados por não influenciarem no uso da regra de aplicação funcional.

```
regra([Arg2, Arg1 | Resto], [Result | Resto]) :-
  Arg1 =..  [ecp, Expr1, Y, Denot1, _, _],
  Arg2 =..  [ecp, Expr2, Y\X, Denot2, _, _],
  append(Expr1, Expr2, Expr),
  reduz([Denot2, Denot1], Normal),
  Result =..  [ecp, Expr, X, Normal, apl_e, [Arg1, Arg2]].
```

Programa 4.10: Cláusula de `regra/2` para aplicação à esquerda

A cláusula da aplicação funcional para a esquerda funciona de modo análogo à cláusula da aplicação à direita, com a diferença de que as categorias têm de ser Y e $Y\backslash X$ para `Arg1` e `Arg2`, respectivamente, a representação semântica resultante será a aplicação da denotação da segunda à da primeira, e a regra será `apl_e`. O código correspondente é listado no Programa 4.10.

Associatividade

Vamos descrever aqui somente a cláusula correspondente à regra de associatividade (ou permutação) para a direita, pois a sua variante para a esquerda é análoga. O código é listado no Programa 4.11.

```
regra([Topo|Resto], [Res|Resto]) :-
  Topo =.. [ecp, Expr, X\(Y/Z),[lmbd,Vx,[lmbd,Vz,Termo]],Regra,_],
  Regra \= per_e,
  Res =.. [ecp, Expr, (X\Y)/Z, [lmbd,Vz,[lbmd,Vx,Termo]],
           per_d, [Topo]].
```

Programa 4.11: Cláusula de `regra/2` para associatividade à direita

Aqui consideramos somente o elemento mais ao topo (*topmost*, em inglês) da pilha. Para que possamos trocar a ordem dos argumentos de uma categoria duas condições devem ser satisfeitas: primeiro, a categoria deve ser da forma $X\backslash(Y/Z)$ (na sua variante para a esquerda, $(X\backslash Y)/Z$), ou seja, deve ser um funtor de dois argumentos. Segundo, a regra usada para obter a ECP em `Topo` não pode ser a associatividade à esquerda; por isso, essa regra (instanciada na variável `Regra`) é comparada com `per_e` (na variante para a esquerda, com `per_d`). Desta maneira, conseguimos limitar a aplicação sucessiva da regra de associatividade a uma mesma expressão, o que levaria o *parser* a rodar infinitamente[5].

A categoria resultante é $(X\backslash Y)/Z$ (ou $X\backslash(Y/Z)$, na variante para a esquerda), e a ordem em que os argumentos são tomados é trocada também na denotação: antes a variável V_x era ligada antes que V_z; na representação resultante essa ordem é invertida, apesar de as variáveis não terem seus lugares trocados no termo-lambda. Na ECP resultante, o elemento que indica a regra usada é instanciado com `per_d` (ou `per_e`), e a lista de constituintes contém somente a ECP em `Topo`.

[5]Na prática, até estourar a memória disponível.

Novamente, aqui a unificação faz o trabalho de selecionar a expressão cuja categoria corresponde às restrições de forma supracitadas e em que a regra pode ser utilizada.

O código da cláusula correspondente à associatividade para a esquerda é listado no Programa 4.12.

```
regra([Topo|Resto], [Res|Resto]) :-
  Topo =.. [ecp, Expr,(X\Y)/Z,[lmbd,Vz,[lmbd,Vx,Termo]],Regra,_],
  Regra \= per_d,
  Res =.. [ecp, Expr, X\(Y/Z), [lmbd,Vx,[lmbd,Vz,Termo]],
           per_e,[Topo]].
```

Programa 4.12: Cláusula de `regra/2` para associatividade à esquerda

Composição

A cláusula correspondente à regra de composição para a direita é listada no Programa 4.13.

```
regra([Arg2, Arg1 | Resto], [Res | Resto]) :-
  Arg1 =.. [ecp, Expr1, X/Y, Denot, _, _],
  Arg2 =.. [ecp, Expr2, Y/Z, [lmbd, Vz, Termo], _, _],
  append(Expr1, Expr2, Expr),
  reduz([Denot, Termo], Normal),
  Res =.. [ecp, Expr, X/Z, [lmbd,Vz,Normal], com_d, [Arg1,Arg2]].
```

Programa 4.13: Cláusula de `regra/2` para composição para a direita

Aqui, outra vez a unificação seleciona as categorias às quais a regra se aplica. Para que essa regra seja usada, as categorias de `Arg1` e de `Arg2` (duas variáveis contendo ECPs) devem ser da forma X/Y e Y/Z, respectivamente. Se enxergarmos a denotação de `Arg1` como uma função f e a de `Arg2` como a função g, a denotação da expressão resultante da composição seria uma função h tal que $h(x) = f(g(x))$. Para representar a composição em termos-lambda, a parte interna da denotação de `Arg2`, instanciada na variável `Termo`, é substituída na denotação de `Arg1`, em `Denot`, e a variável V_z permanece ligada. Essa substituição é feita pelo predicado `reduz/2`.

Para exemplificar vamos mostrar uma análise da expressão *Pedro ama*, de categoria S/N. A categoria de *Pedro* foi elevada de N para $S/(N\backslash S)$, para utilizar a composição (omitida a regra de inserção lexical para clareza). Os passos da redução-β são mostrados para enfatizar como as denotações são encadeadas para formar a função composta, como pode-se ver na Figura 4.2.

$$\dfrac{\dfrac{\text{Pedro}}{\lambda P.P(p)}\,\text{Elev}\;\dfrac{\text{ama}}{\lambda v_z.\lambda v_x.A(v_z)(v_x)}\,\text{lex}}{\dfrac{\text{Pedro ama}}{\begin{array}{c}\lambda v_z[\lambda P[P(p)](\lambda v_x[A(v_z)(v_x)])] \\ \lambda v_z[\lambda v_x[A(v_z)(v_x)](p)] \\ \lambda v_z[A(v_z)(p)]\end{array}}\,\text{ComD}}$$

Figura 4.2: Análise de *Pedro corre* mostrando as reduções-β passo a passo

A variante para a esquerda é análoga, e a cláusula correspondente é listada no Programa 4.14.

```
regra([Arg2, Arg1 | Resto], [Res | Resto]) :-
  Arg1 =.. [ecp, Expr1, Z\Y, [lmbd, Vz, Termo], _, _],
  Arg2 =.. [ecp, Expr2, Y\X, Denot, _, _],
  append(Expr1, Expr2, Expr),
  reduz([Denot, Termo], Normal),
  Res =.. [ecp, Expr, Z\X, [lmbd,Vz,Normal], com_d, [Arg1,Arg2]].
```

Programa 4.14: Cláusula de `regra/2` para composição para a esquerda

Promoção

A cláusula que implementa a regra de promoção para a direita é listada no Programa 4.15.

```
regra([Arg2, Arg1 | Resto], [Arg2, Res | Resto]) :-
  Arg1 =.. [ecp, Exp, X, Denot, _, _],
  Arg2 =.. [ecp, _, Cat, _, _, _],
  (Cat = (X\Y)/_; Cat = X\(Y/_)),
  Res =.. [ecp, Exp, Y/(X\Y), [lmbd, 'P' ,['P', Denot]],
           pro_d,[Arg1]].
```

Programa 4.15: Cláusula de `regra/2` para promoção para a direita

Note que, apesar de a regra de promoção ser unária, a cláusula considera os dois elementos do topo da pilha. Isso é feito para poder restringir o seu uso e impedir que o *parser* entre em *loop*, da seguinte forma: a categoria de `Arg1`, X, somente será promovida para $Y/(X\backslash Y)$ se a categoria de `Arg2`, a ECP da expressão adjacente, permitir uma posterior composição (for da forma $(X\backslash Y)/Z$) ou uma composição precedida por uma permutação (for da forma $X\backslash(Y/Z)$).[6]

A representação semântica, que denotava um indivíduo — por exemplo, no caso do nome *Pedro* —, passa então a denotar um conjunto de conjuntos, precisamente o conjunto dos conjuntos dos quais o indivíduo Pedro faz parte. Obviamente não são somente categorias básicas que podem ser alçadas, mas o resultado é o mesmo: a representação semântica resultante denota uma função ou predicado de ordem superior, que é introduzido com a variável lambda ligada `P`.

Para a promoção para a esquerda o procedimento é análogo, com a diferença de que o elemento do topo da lista a ter a categoria elevada é `Arg2` e a categoria de `Arg1` deve ser da forma $(Z\backslash Y)/X$ ou $Z\backslash(Y/X)$. A representação semântica resultante é a mesma. No Programa 4.16 listamos o código da cláusula correspondente.

As restrições introduzidas na regra de promoção têm um efeito colateral, que é impedir que a categoria seja elevada tendo em vista o uso da regra de *aplicação* posteriormente. Uma solução para isso é discutida na Seção 4.4.

[6] Z é uma categoria qualquer.

```
regra([Arg2, Arg1 | Resto], [Result, Arg1 | Resto]) :-
  Arg1 =.. [ecp, _, Cat, _, _, _],
  Arg2 =.. [ecp, Exp, X, Denot, _, _],
  (Cat = (_\Y)/X; Cat = _\(Y/X)),
  Result =.. [ecp, Exp, (Y/X)\Y, [lmbd, 'P',['P', Denot]],
              pro_e, [Arg2]].
```

Programa 4.16: Cláusula de `regra/2` para promoção para a esquerda

4.3.6 Redução-β

Nesta seção descrevemos os predicados que efetuam a redução-β, realizada quando é feita uma substituição de uma variável ligada em um termo-lambda. Para isso foram escritos dois predicados, `reduz/2` e `substitui/4`.

Predicado `reduz/2`

A primeira e segunda cláusulas, mostradas no Programa 4.17, reduzem, respectivamente, termos-lambda da forma $termo1 \wedge termo2$ e $termo1 \rightarrow termo2$. Para fazer isso, os dois "sub-termos" são reduzidos e retornados no resultado, ainda relacionados pelo conectivo. O *cut* (`!`), introduzido no fim de cada cláusula, tem o objetivo de evitar que sejam tentadas as outras cláusulas de redução-β quando o *parser* fizer o *backtrack* para obter as análises alternativas.

```
reduz([Expressao1, /\, Expressao2], [Resultado1,/\,Resultado2]) :-
  reduz(Expressao1, Resultado1),
  reduz(Expressao2, Resultado2), !.
reduz([Expressao1, =>, Expressao2], [Resultado1,=>,Resultado2]) :-
  reduz(Expressao1, Resultado1),
  reduz(Expressao2, Resultado2), !.
```

Programa 4.17: Primeiras duas cláusulas de `reduz/2`

O Programa 4.18 lista a terceira e quarta cláusulas, que tratam dos quantificadores $\exists$ e $\forall$, representados em Prolog por `algum` e `todo`, respectivamente. O processo é análogo ao das cláusulas anteriores: a "estrutura" do termo-lambda é conservada no resultado, somente a expressão interna é reduzida para fazer substituições de variáveis. Novamente, é adicionado *cut* ao final de cada cláusula para apagar pontos de escolha e impedir *backtrack* nelas. Note que nos dois casos a variável não está ligada, e não deve ser aplicada nenhuma substituição.

```
reduz([algum, Var, Expressao], [algum,Var,Resultado]) :-
  reduz(Expressao, Resultado), !.
reduz([todo, Var, Expressao], [todo,Var,Resultado]) :-
  reduz(Expressao, Resultado), !.
```

Programa 4.18: Terceira e quarta cláusulas de `reduz/2`

A quinta cláusula, listada no Programa 4.19, funciona de modo análogo às quatro já apresentadas: o "sub-termo"-lambda interno é reduzido, para efetuar substituições de variáveis, e a "estrutura" do termo-lambda no resultado é conservada, ou seja, a variável-lambda instanciada em `Var` continua ligada.

```
reduz([lmbd, Var, Expressao], [lmbd,Var,Resultado]):-
  reduz(Expressao, Resultado), !.
```

Programa 4.19: Quinta cláusula de `reduz/2`

A sexta cláusula é a que efetivamente elimina o ligador de variáveis λ e chama o predicado `substitui/4`, que vai trocar na expressão `Expressao` todas as ocorrências da variável-lambda em `Var` pelo termo instanciado em `Argumento`. O resultado parcial, instanciado na variável `Parcial`, deve ser então reduzido, e o resultado final é instanciado na variável `Resultado`.

A última cláusula, mostrada no Programa 4.20, é a condição de parada do predicado: quando nenhuma redução-β for aplicável, o termo-lambda já está na forma reduzida.

```
reduz([[lmbd, Var, Expressao], Argumento], Resultado):-
  substitui(Expressao, Var, Argumento, Parcial),
  reduz(Parcial, Resultado), !.

reduz(Resultado, Resultado).
```

Programa 4.20: Sexta cláusula de `reduz/2`

Predicado `substitui/4`

O predicado `substitui/4`, utilizado pelo predicado `reduz/2`, efetua a substituição de uma variável de um termo-lambda por um outro termo-lambda. As quatro cláusulas são listadas no Programa 4.21.

```
substitui([], _, _, []).

substitui([Var | Resto], Var, Arg, [Arg|Result]) :-
  atomic(Var),
  substitui(Resto, Var, Arg, Result), !.

substitui([Prim | Resto], Var, Arg, [Prim|Result]) :-
  atomic(Prim),
  Prim \= Var,
  substitui(Resto, Var, Arg, Result), !.

substitui([Lista|Resto], Var, Arg, [ResultLista | ResultResto]) :-
  substitui(Lista, Var, Arg,  ResultLista),
  substitui(Resto, Var, Arg, ResultResto).
```

Programa 4.21: Predicado `substitui/4`

A primeira cláusula já testa se a expressão em que será feita a substituição está vazia. Se estiver, não há o que substituir, e o resultado é a própria lista vazia.

Na segunda cláusula o primeiro elemento da expressão é a própria variável a ser substituída. No termo-lambda resultante o termo instanciado em `Arg` é posto no lugar da variável `Var`, e o predicado se chama recursivamente para substituir outras ocorrências de `Var` no resto da expressão. O teste `atomic(Var)` verifica se `Var` é mesmo um átomo do Prolog. Um *cut* é adicionado para evitar *backtracks* indesejados.

A terceira cláusula é semelhante à segunda, com a diferença de que o primeiro elemento do termo-lambda em que será feita a substituição não é a variável `Var`. Por exemplo, pode ser um quantificador ou o ligador de variáveis λ. O primeiro elemento é copiado no termo-lambda resultado e `substitui` é chamado recursivamente para procurar ocorrências de `Var` em `Resto`.

A última cláusula trata do caso em que o primeiro elemento do termo-lambda em que será feita a substituição não é atômico, ou seja, é um sub-termo mais complexo. Para isso, chamamos recursivamente `substitui` para esse sub-termo (resultado instanciado em `ResultLista`) e depois para o restante do termo-lambda (resultado em `ResultResto`). Os resultados parciais são concatenados para gerar o termo-lambda final.

4.3.7 Apresentação das análises

Nesta seção mostramos como o *parser* apresenta as análises. A listagem do código-fonte de todos os predicados descritos nesta seção pode ser encontrado no Apêndice A. A primeira forma de apresentação, já vista acima, é simplesmente imprimir a ECP resultante na tela. O problema com essa apresentação é que ela se torna ilegível com análises mais complexas. Veja por exemplo a ECP da análise de *Pedro ama*, apresentada anteriormente neste capítulo:

```
ecp([pedro, ama], S/N, [lmbd,y_10,[[A, y_10],p]], com_d,[ecp([
   pedro],S/(N\S),[lmbd, P, [P, p]], pro_d, [ecp([pedro], N, p,
   i_lex, [])]), ecp([ama],(N\S)/N,[lmbd, y_10, [lmbd, x_10, [[A,
   y_10], x_10]]], i_lex, [])])
```

Programa 4.22: Predicado `substitui/4`

Para tornar os resultados mais legíveis foram implementadas duas formas de apresentação: lista indentada e representação ao estilo de Prawitz.

Lista indentada

O formato de lista indentada é bastante utilizado, e seu princípio é simples: para cada componente da ECP imprime-se o seu nome indentado para a direita e ao lado é impresso o seu valor. A cada sub-ECP que estiver na lista de constituintes é avançado um passo de indentação. A lista indentada correspondente à ECP acima é:

Expressão: [pedro, ama]

Categoria: s/n

Denotação: [lmbd, y10, [[A, y10], p]]

Operação: Composição para a direita

Argumentos:

Expressão: [pedro]

Categoria: s/(n\s)

Denotação: [lmbd, P, [P, p]]

Operação: Promoção para a direita

Argumento:

Expressão: [pedro]

Categoria: n

Denotação: p

Expressão: [ama]

Categoria: (n\s)/n

Denotação: [lmbd, y10, [lmbd, x10, [[A, y10], x10]]]

O predicado desenvolvido foi o predicado `apresenta/1`, que chama o predicado `indenta/2` para imprimir a representação em forma de lista indentada e, após cada análise, faz o *backtrack* para obter análises alternativas (exatamente como o predicado `conexo/1`). Os resultados são apresentados em texto simples, no próprio terminal do Prolog, e futuramente será gerada saída em LaTeX. É possível gravar a saída em um arquivo com o predicado `grava/1`, que toma como argumento uma lista contendo a expressão a ser analisada e grava as análises no arquivo "apresenta.ind".

Derivação no estilo de Prawitz

Uma maneira de apresentar as análises muito utilizada na literatura é o diagrama de Prawitz, apresentado pela primeira vez em Prawitz (1965). Essa forma de apresentação de análises é mais adequada tipograficamente do que a árvore e realça os conceitos de composicionalidade e de conexidade. Utilizando a notação de Prawitz na sua versão original uma análise de *Pedro corre* ficaria como na Figura 4.3.

$$\dfrac{\dfrac{\text{Pedro}}{N : p}\text{lex} \quad \dfrac{\text{corre}}{N\backslash S : C}\text{lex}}{\dfrac{\text{Pedro corre}}{S : C(p)}\text{AplE}}$$

Figura 4.3: Diagrama de Prawitz na sua forma original

Uma versão levemente modificada foi utilizada neste trabalho, com as informações da análise de cada expressão organizadas de maneira diferente em torno da barra, de modo a tornar sua

leitura mais clara: a categoria foi deslocada para a esquerda da barra, ficando na parte inferior da barra a denotação, acima a expressão e à direita a regra gramatical.

Inicialmente foi construído o predicado `apresenta/1`, que apresentava as derivações ao estilo de Prawitz em texto simples, como no exemplo abaixo.

```
    Pedro        corre
  n------lex n\s-------lex
      p            C
       Pedro corre
s-------------------------aplE
           C(p)
```

Contudo, quando há análises mais complexas a saída em texto simples fica ilegível, pois o tamanho do terminal do Prolog é limitado, além de não permitir formatação ou redimensionamento do texto, e a representação dessas análises é quebrada em diversas linhas, dificultando sua leitura. Uma alternativa é gerar a saída em código-fonte para o processador de textos LaTeX.

O pacote `semantic`, disponível na maioria das distribuições do LaTeX, provê o comando `\inference` para formatar inferências lógicas. Sua sintaxe é:

`\inference[`*esquerda*`]{`*premissas*`}{`*conclusão*`}[`*direita*`]`

O predicado `apresLatex/1` foi escrito para formatar as informações constantes de uma ECP utilizando o comando `\inference` acima. O predicado `gravaTeX/1` é semelhante ao predicado `grava/1` descrito acima, e grava a saída em código LaTeX no arquivo "apresenta.tex".

4.4 Problemas encontrados

No desenvolvimento do *parser* nos deparamos com alguns problemas, relacionados principalmente ao caráter livre da álgebra combinatória da GC, que se relacionam ao fato de o *parser* a entrar em *loop* infinito.

As regras gramaticais da GC permitem uma manipulação algébrica quase ilimitada das expressões da língua para fazer uma análise. O uso associado das regras binárias e unárias permitem diversas análises diferentes, muitas inclusive que não são notadas à primeira vista. As regras unárias (associatividade e elevação de tipo) permitem que transformemos as categorias das expressões de modo a serem possíveis análises totalmente incrementais e se mostram ferramentas importantes na análise de fragmentos maiores da língua em questão.

As regras unárias, porém, não têm nenhuma limitação de uso *a priori*, podendo ser aplicadas indefinidamente a uma categoria sem contribuir com análises pertinentes. Isso se configura num grande problema quando o *parser* é implementado em computador, pois as regras podem levar a *loops* infinitos. Para solucionar esse problema foram introduzidas limitações para o uso das duas regras, que discutimos abaixo.

4.4.1 Associatividade

Para impedir que a regra de associatividade seja aplicada indefinidamente em um categoria, introduzimos um teste (já apresentado na Seção 4.3.5) para verificar se a associatividade foi

aplicada imediatamente antes na categoria. Na cláusula da associatividade à direita, a regra anterior é comparada com `per_e`, e na variante à esquerda, com `per_d`. Com essa restrição a aplicação sucessiva da associatividade é evitada, e o *parser* não entra em *loop* infinito quando usa essa regra.

4.4.2 Elevação de tipo

O problema com a regra de elevação de tipo ou promoção é mais complicado ainda do que com a de associatividade, pois a elevação pode ser aplicada a qualquer categoria — e não somente a categorias funtoras que contenham dois conectivos. Em (Wood, 1993, p. 42) é apresentada uma discussão sobre o fato de as regras de composição e de elevação de tipo complementarem uma à outra, "a regra unária provendo a categoria necessária para que a regra binária seja aplicável"[7]. Seguindo essa perspectiva, limitamos o uso da regra de composição para os casos em que ela permita uma composição subseqüente ou uma composição antecedida por uma aplicação da regra de associatividade. Na cláusula da promoção para a direita esse teste é feito pela linha:

`(Cat = (X\Y)/_; Cat = X\(Y/_))`,

em que `Cat` é a categoria da expressão adjacente à qual se pretende aplicar a regra de elevação de tipo. Para a cláusula da variante para a esquerda a linha é:

`(Cat = (_\ Y)/X; Cat = _\(Y/X))`.

Essa solução, porém, acaba tendo um efeito colateral: como a regra de elevação só é aplicada quando a categoria da expressão adjacente permitir uma composição subseqüente (ou intercalada pela associatividade), a regra não é utilizada para o caso em que uma *aplicação* subseqüente é possível. Para isso, um terceiro teste é acrescentado a cada uma das linhas acima, para verificar se a categoria da expressão adjacente permite que seja feita uma aplicação. Para a promoção para a direita o teste adicionado é `Cat =` $X\backslash Y$, sendo `Cat` a categoria da expressão à direita da expressão cuja categoria estamos elevando. Para a promoção para a esquerda, o teste é `Cat = Y\X`, neste caso `Cat` é a categoria da expressão à esquerda da expressão de categoria a ser elevada. As linhas das cláusulas correspondentes ficam, então, alteradas para:

```
(Cat = X\Y; Cat = (X\Y)/_; Cat = X\(Y/_))

(Cat = Y\X; (_\Y)/X; Cat = _\(Y/X))
```

Programa 4.23: Alterações nas regras de elevação de tipo

4.5 Resultados experimentais

Nesta seção apresentamos algumas análises obtidas pelo *parser*, implementadas as restrições de uso das regras unárias.

[7] "the unary rule providing the category necessary for the binary rule to be applicable".

4.5.1 *Pedro corre*

Nas Figuras 4.4 e 4.5 estão as duas análises obtidas para a sentença *Pedro corre*. Dois resultados são obtidos, um somente com aplicação funcional e outro com uso da elevação de tipo do nome *Pedro*. Essa segunda análise só é obtida se fizermos a mudança apontada na Seção 4.4.2 acima.

$$\dfrac{N\dfrac{\text{pedro}}{p}\text{Lex}\quad N\backslash S\dfrac{\text{corre}}{\lambda x_{12}[C(x_{12})]}\text{Lex}}{S\dfrac{\text{pedro corre}}{C(p)}\text{AplE}}$$

Figura 4.4: Análise de *Pedro corre* usando aplicação funcional

$$\dfrac{\dfrac{N\dfrac{\text{pedro}}{p}\text{Lex}}{S/(N\backslash S)\dfrac{\text{pedro}}{\lambda P[P(p)]}\text{ProD}}\quad N\backslash S\dfrac{\text{corre}}{\lambda x_{12}[C(x_{12})]}\text{Lex}}{S\dfrac{\text{pedro corre}}{C(p)}\text{AplD}}$$

Figura 4.5: Análise de *Pedro corre* usando elevação de tipo

4.5.2 *Pedro ama Maria*

Para a sentença *Pedro ama Maria* o *parser* conseguiu encontrar dez análises diferentes, todas com a mesma representação semântica: $A(m)(p)$. Mostramos nas Figuras 4.6, 4.7 e 4.8 as três primeiras análises por motivos de espaço. Na análise da Figura 4.6 o verbo *ama* se combina primeiro com *Pedro*, para depois se combinar com *Maria*. Na Figura 4.7, o nome *Pedro* tem sua categoria elevada, e é aplicada à categoria de *ama*; logo depois, o nome *Maria* pode ser combinado à expressão para formar a sentença. Por fim, na Figura 4.8 a seqüência é diferente: primeiro o verbo *ama* se combina com o nome *Maria*, que teve a categoria previamente elevada, para depois se combinar com o nome *Pedro*.

$$\dfrac{\dfrac{N\dfrac{\text{Pedro}}{p}\text{Lex}\quad N\backslash(S/N)\dfrac{(N\backslash S)/N\dfrac{\text{ama}}{\lambda y_{10}[\lambda x_{10}[A(y_{10})(x_{10})]]}\text{Lex}}{\lambda x_{10}[\lambda y_{10}[A(y_{10})(x_{10})]]}\text{PerE}}{S/N\dfrac{\text{Pedro ama}}{\lambda y_{10}[A(y_{10})(p)]}\text{AplE}}\quad N\dfrac{\text{Maria}}{m}\text{Lex}}{S\dfrac{\text{Pedro ama Maria}}{A(m)(p)}\text{AplD}}$$

Figura 4.6: Análise de *Pedro ama Maria* com aplicação e associatividade

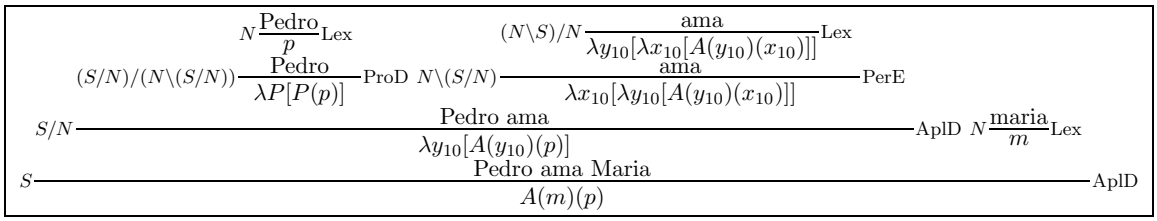

Figura 4.7: Análise de *Pedro ama Maria* utilizando elevação de tipo

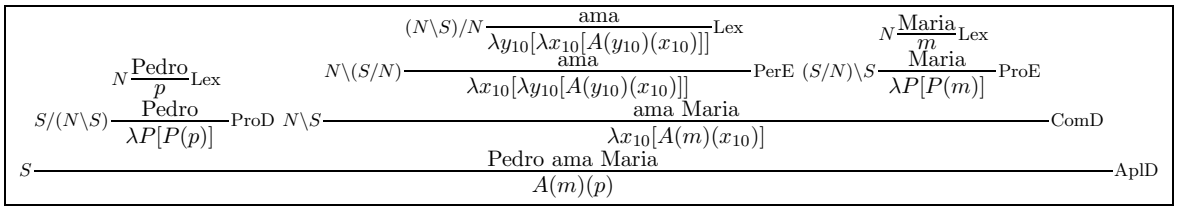

Figura 4.8: Análise de *Pedro ama Maria* combinando *ama* primeiro com Maria e depois com *Pedro*

4.5.3 *Pedro ama*

Nas Figuras 4.10, 4.11 e 4.9 mostramos as análises retornadas para a expressão *Pedro ama*, de categoria S/N. Note que a sentença está incompleta, pois estamos considerando o verbo *ama* somente na sua versão transitiva direta. Desse modo, o *parser* não reconhece *Pedro ama* como sendo de categoria S, como era de se esperar.

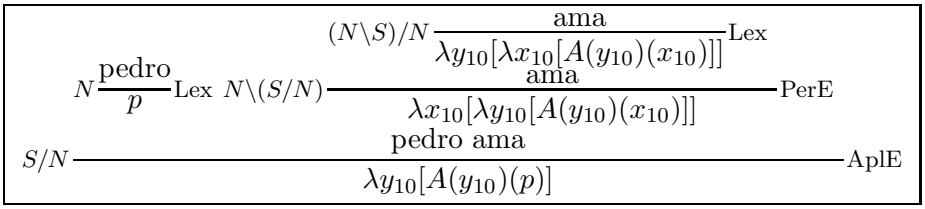

Figura 4.9: Análise de *Pedro ama* utilizando associatividade e aplicação

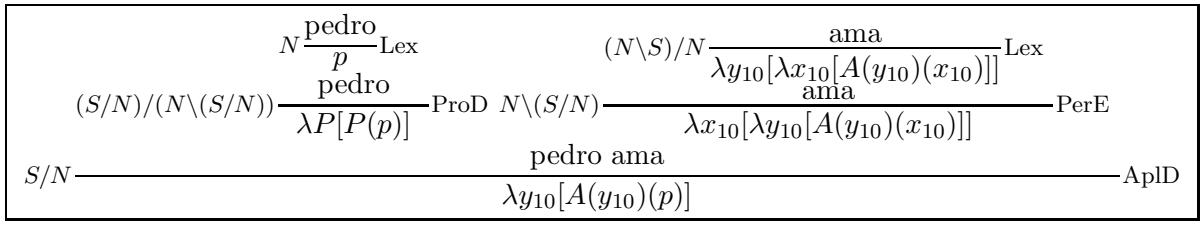

Figura 4.10: Análise de *Pedro ama* com elevação e tipo e associatividade

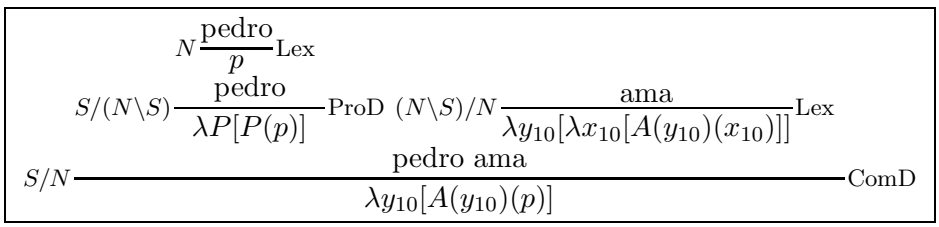

Figura 4.11: Análise de *Pedro ama* com elevação de tipo e composição

CAPÍTULO 5

TRABALHOS FUTUROS

O *parser* apresentado neste trabalho está em estágio experimental ainda, e ainda há bastante trabalho a ser feito para torná-lo utilizável em uma aplicação que necessite de um *parser* funcional. Nesta seção discutimos propostas de trabalhos futuros, listadas mais ou menos em ordem de magnitude das mudanças com relação à implementação atual, e são dadas indicações de possíveis soluções.

A primeira proposta é reestruturar os predicados do *parser* de modo a não evitar a ocorrência dos conflitos deslocamento-redução e redução-redução. Como discutido na Seção 4.3.2, a implementação apresentada neste trabalho contorna os conflitos de maneira artificial, evitando que durante uma análise o *parser* tenha que escolher entre deslocamento e redução. Também é interessante que nessa nova implementação os conflitos redução-redução não se resolvam simplesmente pela ordem dos predicados no programa Prolog, mas sim por outros fatores como, por exemplo, qual das regras gramaticais consegue reduzir mais expressões de uma vez.

Outra proposta, atualmente em estudo, é o uso de um parser tabular ou *chart parser* para evitar trabalho duplicado quando o *backtrack* é feito para obter possíveis análises alternativas. Uma implementação que consiga dar conta da flexibilidade da álgebra combinatória da GC é um trabalho interessante.

O desenvolvimento de uma interface gráfica para usuários (GUI) para facilitar a manipulação das regras gramaticais, a análise de sentenças e a apresentação dessas análises é uma extensão importante, pois permitiria a disseminação do *parser* para lingüistas e outras pessoas que não tenham conhecimento prévio de Prolog, e que poderiam contribuir no formalismo da GC. Há diversas possibilidades de implementação, com ferramentas como o XPCE (biblioteca gráfica do SWI-Prolog) ou o *toolkit* gráfico Tcl/Tk, entre outras.

A representação semântica, expressa na linguagem do cálculo lambda, necessita ser mais completa, com a implementação de predicados que realizem as reduções α e η, além de ser necessário um tratamento melhor para quantificadores ($\exists$ e $\forall$). A especificação das variáveis-lambda, por ser arbitrariamente especificada no léxico, é problemática nos casos em que uma mesma palavra ocorre mais de uma vez na expressão, causando repetição de variáveis e possíveis substituições incorretas. Esse problema ainda não fez o *parser* obter resultados errados devido ao tamanho reduzido e à simplicidade do léxico considerado, porém com um léxico maior e com itens lexicais mais complexos esse problema pode se tornar grave. Um sistema de tipos lógicos mais rigidamente formalizado também é um estudo interessante.

Como o objetivo deste trabalho é desenvolver um *parser* funcional que obtenha o maior número de análises pertinentes, a atenção não foi direcionada às questões de eficiência do código Prolog, apesar de não terem sido totalmente negligenciadas. Modificações para aumentar a eficiência do *parser*, de modo a torná-lo robusto a expressões mais complexas do que as atualmente tratadas e passíveis de muitas possibilidades de análises são tema de trabalho futuro a partir da implementação atual.

Um outro trabalho a se fazer é aumentar o fragmento lingüístico coberto pelo *parser*, pois, ao aumentar a quantidade de expressões possíveis de serem analisadas, podemos avaliar se a versão da GC utilizada é adequada ou necessita de extensões, seja nas categorias básicas, nos conectivos ou nas regras gramaticais, ou se uma outra estratégia de análise se faz necessária.

Por último, mais como um experimento e um exercício do que uma extensão mais concreta e dirigida, o desenvolvimento de um *parser* para a GC utilizando linguagens funcionais (Haskell, por exemplo) seria uma possibilidade interessante.

CAPÍTULO 6

CONCLUSÕES

O objetivo deste trabalho foi apresentar o estudo e a implementação de um analisador gramatical por deslocamento e redução em Prolog para Gramática Categorial, mais precisamente a GC livre de Cohen. Problemas decorrentes da implementação do *parser*, principalmente com relação à recursividade das regras categoriais unárias, foram apresentados, juntamente com as soluções adotadas e as respectivas limitações e conseqüências.

Dado o caráter experimental deste trabalho, uma lista razoável de trabalhos futuros e propostas de extensão do *parser* é apresentada no capítulo 5, indicando justamente os pontos em que o *parser* ainda deixa um pouco a desejar. O trabalho está ainda no seu início, e com os devidos refinamentos e extensões pode contribuir para a construção de um *parser* bastante funcional, além de (como indicado na lista de trabalhos futuros) uma interface gráfica que permita a disseminação do *parser* para mais pessoas.

APÊNDICE A

LISTAGEM DOS PROGRAMAS APRESENTADOS

Neste apêndice incluímos o código-fonte do *parser* original desenvolvido em Covington (1994), e a parte do código-fonte deste trabalho que trata da apresenação das análises. Somente são listados aqui os predicados que fazem a apresentação das análises, porque os predicados que fazem a análise e os predicados que efetuam a redução-β já são apresentados nas respectivas seções.

Programa A.1: *Parser* original de Covington

```
% Bottom-up shift-reduce parser
% (Covington 1994: 159)

% parse(+S,?Result)
% parses input string S, where Result
% is list of categories to which it reduces.
parse(S,Result) :- shift_reduce(S,[],Result).

% shift_reduce(+S,+Stack,?Result)
% parses input string S, where Stack is
% list of categories parsed so far.
shift_reduce(S,Stack,Result) :-
  shift(Stack,S,NewStack,S1), % fails if S = []
  reduce(NewStack,ReducedStack),
  shift_reduce(S1,ReducedStack,Result).
shift_reduce([],Result,Result).

% shift(+Stack,+S,-NewStack,-NewS)
% shifts first element from S onto Stack.
shift(X,[H|Y],[H|X],Y).

% reduce(+Stack,-ReducedStack)
% repeatedly reduces beginning of Stack
% to form fewer, larger constituents.
reduce(Stack,ReducedStack) :-
  brule(Stack,Stack2),
  reduce(Stack2,ReducedStack).
reduce(Stack,Stack).

% Phrase structure rules
```

```
brule([vp,np|X],[s|X]).
brule([n,d|X],[np|X]).
brule([np,v|X],[vp|X]).
brule([Word|X],[Cat|X]) :- word(Cat,Word).

% Lexicon
word(d,the).
word(n,dog).
word(n,dogs).
word(n,elephant).
word(n,elephants).
word(v,chase).
word(v,chases).
word(v,see).
word(v,sees).
```

Programa A.2: Código para apresentar os resultados no formato de lista indentada

```
% indenta.pl
% Apresentação indentada para análises com gramáticas categoriais

:- [gramcat].

grava(X) :-
  tell('apresenta.ind'),
  apresenta(X),
  told.

apresenta(X) :-
  write('-----------------------------------------------------'),
  conexo([], Resultado, X, []),
  nl,
  indenta(Resultado, 0),
  write('-----------------------------------------------------'),
  fail.
apresenta(_).

indenta(ecp(Palavra, Categoria, Denotação, i_lex, []), Tab) :-
  !,
  tab(Tab), write('Expressão: '), write(Palavra), nl,
  tab(Tab), write('Categoria: '), write(Categoria), nl,
  tab(Tab), write('Denotação: '), write(Denotação), nl, nl.
indenta(X, Tab) :-
```

```
  X =..   [ecp,
           Expressão,
           Categoria,
           Denotação,
           Operação,
           Argumentos],
  tab(Tab), write('Expressão: '), write(Expressão), nl,
  tab(Tab), write('Categoria: '), write(Categoria), nl,
  tab(Tab), write('Denotação: '), write(Denotação), nl,
  tab(Tab), write('Operação : '), escreve(Operação), nl,
  tab(Tab),
    (Argumentos = [_] -> write('Argumento: ');
       write('Argumentos: ')), nl, nl,
  Tab1 is Tab + 5,
  indenta_lista(Argumentos, Tab1).

indenta_lista([], _).
indenta_lista([Primeiro | Resto], Tab) :-
  indenta(Primeiro, Tab),
  indenta_lista(Resto, Tab).

escreve(X) :-
  traduz(X, Y),
  write(Y).

traduz(apl_d, 'Aplicação para a direita').
traduz(apl_e, 'Aplicação para a esquerda').
traduz(per_d, 'Permutação para a direita').
traduz(per_e, 'Permutação para a esquerda').
traduz(com_d, 'Composição para a direita').
traduz(com_e, 'Composição para a esquerda').
traduz(pro_d, 'Promoção para a direita').
traduz(pro_e, 'Promoção para a esquerda').
```

Programa A.3: Código para apresentar os resultados no estilo de Prawitz, versão texto simples

```
% prawitz.pl
% Autor: Luiz Arthur Pagani
% E-mail: arthur@ufpr.br
% Criação: 18.10/2002
% Última modificação: 29.03/2003
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Apresentação de derivações ao estilo de Prawitz %
```

```
% para gramáticas categoriais                          %
% (depois de cada expressão, pula de linha)           %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
:- [gramcat].

grava(Expressão) :-
        tell('resultado.pra'),
        apresenta(Expressão),
        told.

apresenta(Expressão) :-
        conexo([], Resultado, Expressão, []),
        nl,
        prawitz(Resultado, 0, Final),
        nl,
        linha(Final),
        fail.
apresenta(_).

% Palavra
prawitz(ecp([Palavra], Cat, Denotação, i_lex, []), Tab, Final) :-
  name(Palavra, PalavraLista),
  length(PalavraLista, TamanhoPalavra),
  tamanho(Cat, TamanhoCategoria),
  denot(Denotação, DenotaçãoLista),
  length(DenotaçãoLista, TamanhoDenotação),
  máximo(TamanhoDenotação, TamanhoPalavra, Máximo),
  Tab1 is Tab + TamanhoCategoria,
  Tab2 is Tab1 + ((Máximo - TamanhoPalavra)//2) + 1,
  tab(Tab2),
  write(Palavra),
  nl,
  tab(Tab),
  write(Cat),
  Linha is Máximo + 2,
  linha(Linha),
  write('lex'),
  nl,
  Tab3 is Tab1 + ((Máximo - TamanhoDenotação)//2) + 1,
  tab(Tab3),
  name(Termo, DenotaçãoLista),
  write(Termo),
```

```
  nl,
  Final is Tab1 + Linha + 3.

% Derivação unaria
prawitz(ecp(Expr, Cat, Denotação, Regra, [Arg]), Tab, Final) :-
  tamanho(Cat, TamanhoCategoria),
  arg(2, Arg, CategoriaArgumento),
  tamanho(CategoriaArgumento, TamanhoCategoriaArgumento),
  Tab0 is Tab + (TamanhoCategoria - TamanhoCategoriaArgumento),
  prawitz(Arg, Tab0, _),
  nl,
  expr(Expr, ExpressãoLista),
  length(ExpressãoLista, TamanhoExpressão),
  denot(Denotação, DenotaçãoLista),
  length(DenotaçãoLista, TamanhoDenotação),
  máximo(TamanhoDenotação, TamanhoExpressão, Máximo),
  Tab1 is Tab + TamanhoCategoria,
  Tab2 is Tab1 + ((Máximo - TamanhoExpressão)//2) + 1,
  tab(Tab2),
  name(ExprN, ExpressãoLista),
  write(ExprN),
  nl,
  tab(Tab),
  write(Cat),
  Linha is Máximo + 2,
  linha(Linha),
  write(Regra),
  nl,
  Tab3 is Tab1 + ((Máximo - TamanhoDenotação)//2) + 1,
  tab(Tab3),
  name(Denot, DenotaçãoLista),
  write(Denot),
  nl,
  Final is Tab1 + Linha + 5.

% Derivação binária
prawitz(ecp(Expr, Cat, Denot, Regra, [Arg1, Arg2]), Tab, Final) :-
  prawitz(Arg1, Tab, Intermediário),
  Tab1 is Intermediário + 3,
  prawitz(Arg2, Tab1, Final),
  nl,
  expr(Expr, ExpressãoLista),
```

```
  length(ExpressãoLista, TamanhoExpressão),
  tamanho(Cat, TamanhoCategoria),
  denot(Denot, DenotaçãoLista),
  length(DenotaçãoLista, TamanhoDenotação),
% máximo(TamanhoDenotação, TamanhoExpressão, Máximo),
% Tab2 is Tab + TamanhoCategoria,
  Tab3 is Tab + ((Final - TamanhoExpressão)//2),
  tab(Tab3),
  name(ExprN, ExpressãoLista),
  write(ExprN),
  nl,
  tab(Tab),
  write(Cat),
  tamanho(Cat, TamanhoCategoria),
  Linha is Final - Tab - TamanhoCategoria - 3,
  linha(Linha),
  write(Regra),
  nl,
  Tab4 is Tab + ((Final - TamanhoDenotação)//2),
  tab(Tab4),
  name(DenotN, DenotaçãoLista),
  write(DenotN),
  nl.

expr([Final],FinalLIsta) :-
  name(Final, FinalLIsta).
expr([Primeiro | Resto], ExpressãoLista) :-
  name(Primeiro, PrimeiroLista),
  expr(Resto, RestoLista),
  append(PrimeiroLista, [32 | RestoLista],  ExpressãoLista).

denot(X, Denot) :-
  atomic(X),
  name(X, Denot).
denot([P,A], Denot) :-
  denot(P, PLista),
  denot(A, AListа),
  append(PLista, [32 | AListа], Parcial),
  append([40 | Parcial], [41], Denot).
denot([Op, Var, Exp],  Denot) :-
  denot(Op, OpLista),
  denot(Var, VarLista),
```

```
  denot(Exp, ExpLista),
  append(VarLista, [32 | ExpLista], Parcial1),
  append(OpLista, [32 | Parcial1], Parcial2),
  append([40 | Parcial2], [41], Denot).

máximo(X, Y, X) :-
  X >= Y.
máximo(X, Y, Y) :-
  X < Y.

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Quantidade de caracteres numa categoria %
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Predicado auxiliar para categorias atômicas
tamanho_atomico(Categoria, CategoriaTamanho) :-
  atomic(Categoria),
  name(Categoria, CategoriaLista),
  length(CategoriaLista,CategoriaTamanho).

% Categoria atômica
tamanho(Categoria, Tamanho) :-
  tamanho_atomico(Categoria, Tamanho),
  !.

% Aplicação para direita formada por duas categorias atômicas
tamanho(Complexo, Tamanho) :-
  Complexo =.. [/, X, Y],
  tamanho_atomico(X, TamanhoX),
  tamanho_atomico(Y, TamanhoY),
  Tamanho is TamanhoX + TamanhoY + 1,
  !.

% Aplicação para esquerda formada por duas categorias atômicas
tamanho(Complexo, Tamanho) :-
  Complexo =.. [\,X,Y],
  tamanho_atomico(X,TamanhoX),
  tamanho_atomico(Y,TamanhoY),
  Tamanho is TamanhoX + TamanhoY + 1,
  !.

% Aplicação para direita formada por categoria atômica e complexa
tamanho(Complexo, Tamanho) :-
```

```
  Complexo =.. [/, X, Y],
  tamanho_atomico(X, TamanhoX),
  tamanho(Y, TamanhoY),
  Tamanho is TamanhoX + TamanhoY + 4,
  !.

% Aplicação para esquerda formada por categoria atômica e complexa
tamanho(Complexo, Tamanho) :-
  Complexo =.. [\, X, Y],
  tamanho_atomico(X, TamanhoX),
  tamanho(Y, TamanhoY),
  Tamanho is TamanhoX + TamanhoY + 4,
  !.

% Aplicação para direita formada por categoria complexa e atômica
tamanho(Complexo, Tamanho) :-
  Complexo =.. [/, X, Y],
  tamanho(X, TamanhoX),
  tamanho_atomico(Y, TamanhoY),
  Tamanho is TamanhoX + TamanhoY + 3,
  !.

% Aplicação para esquerda formada por categoria complexa e atômica
tamanho(Complexo, Tamanho) :-
  Complexo =.. [\, X, Y],
  tamanho(X, TamanhoX),
  tamanho_atomico(Y, TamanhoY),
  Tamanho is TamanhoX + TamanhoY + 3,
  !.

% Aplicação para direita formada por duas categorias complexas
tamanho(Complexo, Tamanho) :-
  Complexo =.. [/, X, Y],
  tamanho(X, TamanhoX),
  tamanho(Y, TamanhoY),
  Tamanho is TamanhoX + TamanhoY + 6,
  !.

% Aplicação para esquerda formada por duas categorias complexas
tamanho(Complexo, Tamanho) :-
  Complexo =.. [\, X, Y],
  tamanho(X, TamanhoX),
```

```
  tamanho(Y, TamanhoY),
  Tamanho is TamanhoX + TamanhoY + 6.

%%%%%%%%
% Linha %
%%%%%%%%
% Desenha uma linha de tamanho especificado
linha(0) :- !.
linha(X) :-
  write('-'),
  Y is X - 1,
  linha(Y).
```

Programa A.4: Código para apresentar os resultados no estilo de Prawitz, versão código LaTeX

```
% prawitzTeX.pl
% Apresentação de análises no estilo de Prawitz, usando macros do
% pacote semantic do LaTeX
% incluir o pacote semantic com \usepackage[inference]{semantic}
% Autor: Daniel Martineschen
% Criação: 20/02/2004

:- [gramcat].

gravaTeX(X) :-
        tell('apresenta.tex'),
        apresLatex(X),
        told.

apresLatex(X) :-
  conexo([], Resultado, X, []),
  nl,
  gera_saida(Resultado, 0),
  nl, nl,
  fail.
apresLatex(_).

% indentação de itens lexicais
gera_saida(ecp(Palavra, Categoria, Denotação, i_lex, []), Tab) :-
  !,
  % Tab1 eh a tabulacao dos parametros de \inference
  Tab1 is Tab + 2,
  tab(Tab), write('\\inference'), nl,
```

```
  tab(Tab1),write('[$'),
  escreveCat(Categoria,0), write('$]'), nl,
  tab(Tab1),write('{\\mbox{'),
  escreveExpr(Palavra),write('}}'),nl,
  tab(Tab1),write('{'), lambda(Denotação), write('}'), nl,
  tab(Tab1),write('['), regraTeX(i_lex), write(']'), nl.

% indentação para as outras regras
gera_saida(X, Tab) :-
  X =..   [ecp,
           Expressão,
           Categoria,
           Denotação,
           Operação,
           Argumentos],
  Operação \= i_lex,
  % Tab1 eh a tabulacao dos parametros de \inference
  Tab1 is Tab + 2,
  tab(Tab), write('\\inference'), nl,
  tab(Tab1), write('[$'),
  escreveCat(Categoria,0), write('$]'), nl,
  tab(Tab1), write('{'), nl,
  gera_saida_lista(Argumentos, Tab1),
  tab(Tab1), write('\\\\ \\mbox{'),
  escreveExpr(Expressão), write('}'), nl,
  tab(Tab1), write('}'), nl,
  tab(Tab1), write('{'), lambda(Denotação), write('}'), nl,
  tab(Tab1), write('['), regraTeX(Operação), write(']'), nl.

% gera_saida_lista: gera codigo latex para uma lista de ecps
gera_saida_lista([], _).
gera_saida_lista([Primeiro | Resto], Tab) :-
  gera_saida(Primeiro, Tab),
  gera_saida_lista(Resto, Tab).

regraTeX(X) :-
  traduz(X, Y),
  write(Y).

% traduz
traduz(apl_d, 'AplD').
traduz(apl_e, 'AplE').
```

```
traduz(per_d, 'PerD').
traduz(per_e, 'PerE').
traduz(com_d, 'ComD').
traduz(com_e, 'ComE').
traduz(pro_d, 'ProD').
traduz(pro_e, 'ProE').
traduz(i_lex, 'Lex').

% escreveCat(+Cat, +Nivel)
% predicado para escrever as categorias em LaTeX, trocando
% '\' por '\backslash'
% a variavel Nivel indica se a categoria deve receber parenteses:
% se Nivel > 0 a categoria a escrever é uma sub-categoria

% nivel 0: categoria complexa e que nao deve ser cercada por
% parenteses
escreveCat(Cat, 0) :-
  Cat =.. [Conec, X, Y],
  escreveCat(X, 1),
  (Conec = \ -> write('\\backslash '); write(Conec)),
  escreveCat(Y, 1), !.
% nivel > 0: categoria complexa que é sub-categoria de outra; deve
% receber parenteses
escreveCat(Cat, Nivel) :-
  Cat =.. [Conec, X, Y],
  NovoNivel is Nivel + 1,
  write('('),
  escreveCat(X, NovoNivel),
  (Conec = \ -> write('\\backslash '); write(Conec)),
  escreveCat(Y, NovoNivel),
  write(')'), !.
%categoria atomica: nao importa o Nivel, ela nao deve receber
%parenteses
escreveCat(Cat, _) :-
  atomic(Cat),
  write(Cat).

% lambda(+Termo):
% predicado para converter termos-lambda da forma de lista para
% comandos do modo matemático do LaTeX
% termo /\ termo
lambda([Termo1, /\, Termo2]) :-
```

```
  lambda(Termo1),
  write('\\wedge '),
  lambda(Termo2), !.
% termo => termo
lambda([Termo1, =>, Termo2]):-
  lambda(Termo1),
  write('\\rightarrow '),
  lambda(Termo2), !.
% variaveis e constantes
lambda(Termo) :-
  atomic(Termo),
  write(Termo), !.
% predicados de mais de um argumento
lambda([[Pred, Arg1], Arg2]) :-
  write(Pred),
  write('('), write(Arg1), write(')'),
  write('('), write(Arg2), write(')'),
  !.
% termos [Pred, Arg], ou seja, Pred(Arg)
lambda([Pred, Arg]):-
  lambda(Pred),
  write('('), lambda(Arg), write(')'), !.
% termos [lmbd var [...]], com [...] um termo lambda
lambda([lmbd, Var, Termo]):-
  write('\\lambda '),
  write(Var), write(' ['),
  lambda(Termo), write(']').

% escreveExpr(+Expr)
% Converte uma lista de expressoes Expr para texto
escreveExpr([]).
escreveExpr([Prim]) :-
  write(Prim), !.
escreveExpr([Prim|Resto]) :-
  write(Prim), write(' '),
  escreveExpr(Resto).
```

REFERÊNCIAS BIBLIOGRÁFICAS

AHO, A. V.; SETHI, R., e ULLMAN, J. D. (1988). **Compilers. Principles, Techniques, and Tools**. Reading, Massachussets, Addison-Wesley.

AJDUKIEWICZ, K. (1935). Die syntaktische Konnexität. **Studia Philosophica**, (1):1–27.

BAR-HILLEL, Y. (1953). A quasi-arithmetical notation for syntatic description. **Language**, (29):47–58.

BAR-HILLEL, Y.; GAIFMAN, C., e SHAMIR, E. (1960). **On categorial and phrase struture grammars**. In *The Bulletin of Research Council of Israel*, v. 9F, p. 1–16.

BORGES NETO, J. (1999). **Introdução às Gramáticas Categoriais**. Inédito.

COHEN, J. M. (1967). The equivalence of two concepts of categorial grammar. **Information and Control**, (10):475–84.

COVINGTON, M. (1994). **Natural Language Processing for Prolog Programmers**. Englewood Cliffs, Prentice-Hall.

CRESSWELL, M. J. (1988). **Categorial languages**. In BUSZKOWSKI, W.; MARCISZEWSKI, W., e VAN BENTHEM, J., editores, *Categorial Grammars*, p. 113–26. John Benjamins, Amsterdam.

CURRY, H. B. (1930). Grundlagen der kombinatorischen Logik. **American Journal of Mathematics**, (52):pp. 509–36, 789–934.

CURRY, H. B. (1961). **Some logical aspects of grammatical struture**. In JAKOBSON, R. O., editor, *Structure of Language and its Mathematical Aspects: Proceedings of the Symposia in Applied Mathematics*, v. XII, p. 56–68. American Mathematical Society, Providence, RI.

DOWTY, D.; WALL, R. E., e PETERS, S. (1981). **Introduction to Montague Semantics**. Dordrecht, Reidel.

FREGE, G. (1891). **Funktion und Begriff, traduzido como 'Function and Concept'**. In GEACH, P. e BLACK, M., editores, *Translations from the Philosophical Writings of Gottlob Frege*, p. 21–41. Blackwell, Oxford.

FREGE, G. (1967). **Begriffsschrift, eine der arithmethischen nachgebildete Formelsprache des reinen Denkens, traduzido como 'Begriffsschift, a formula language, modeled upon that of arithmetic, for pure thought'**. In HEIJENOORT, J., editor, *From Frege to Gödel: a Source Book in Mathematical Logic*, p. 1–82. Harvard University Press, Cambridge, MA.

HINDLEY, J. e SELDIN, R. (1986). **Introduction to Combinators and λ-*Calculus***. Cambridge, Cambridge University Press.

LAMBEK, J. (1958). The mathematics of sentence srtucture. **American Mathematical Monthly**, (65):154–70.

PAGANI, L. A. (2003). Analisador gramatical por deslocamento e redução para gramáticas categoriais. **http://www.cce.ufpr.br/~pagani/gramcat.pdf**. Acessado em julho de 2003.

PEREIRA, F. e SHIEBER, S. (1987). **Prolog and Natural-Language Processing**. Stanford, CSLI.

PEREIRA, F. e WARREN, D. (1980). Definite clause grammars for language analysis: a survey of the formalism and a comparison with augmented transition networks. **Artificial Intelligence**, (13):231–78.

PRAWITZ, D. (1965). **Natural Deduction - A Proof-Theoretical Study**. Stockholm, Almqvist & Wiksell.

SCHÖNFINKEL, M. (1924). Über die Bausteine der mathematischen Logik. **Mathematische Annalen**, (92):305–16.

WOOD, M. M. (1993). **Categorial Grammars**. London and New York, Routledge.