MC346 - Paradigmas de programação
Prolog

João Meidanis

Conteúdo

Capítulo 1

Introdução

Prolog é uma linguagem descritiva e prescritiva. Ao mesmo tempo que descreve-se *o que* deve ser feito, prescreve-se *como* isto deve ser feito.

As presentes notas estão baseadas no livro de Clocksin e Mellish, *Programming in Prolog* [1]. Basicamente, o material apresentado aqui é a tradução das partes deste livro selecionadas para a disciplina MC346 – Paradigmas de Programação, da Universidade Estadual de Campinas, no Brasil.

1.1 Objetos e relações

Prolog lida com *objetos* e *relações* entre eles. A palavra "objeto" não tem o mesmo sentido que em orientação a objetos, pois os objetos Prolog não têm métodos e não há herança. Em Prolog, objetos são apenas coisas sobre as quais queremos raciocinar.

Prolog tem um tipo chamado *termo* que engloba todos os dados e também todos os programas nesta linguagem.

1.2 Programação em Prolog

Um programa em Prolog é composto de:

- fatos sobre certos objetos
- regras de inferência
- perguntas sobre os objetos

Dizemos a Prolog certos fatos e regras, e depois fazemos perguntas sobre estes fatos e regras. Por exemplo, podemos informar Prolog sobre irmãs e depois perguntar se Maria e Joana são irmãs. Prolog responderá sim ou não em função do que lhe dissemos.

Prolog na verdade faz muito mais do que responder sim ou não: a linguagem permite usar o computador como um aracbouço de fatos e regras, e proporciona meios de fazer inferências, indo de um fato a outro, e achando os valores das variáveis que podem levar a conclusões lógicas.

Prolog é geralmente usada como uma linguagem interpretada. Neste curso vamos usar o interpretador SWI-Prolog, uma implementação disponibilizada gratuitamente. Onde pegar: `http://www.swi-prolog.org`. Há para todos os tipos de sistemas operacionais.

1.3 Fatos

Eis alguns exemplos de como se informam fatos a Prolog:

`gosta(pedro, maria).`	Pedro gosta de Maria
`gosta(maria, pedro).`	Maria gosta de Pedro
`valioso(ouro).`	Ouro é valioso
`mulher(jane).`	Jane é mulher
`possui(jane, ouro).`	Jane possui ouro
`pai(pedro, maria).`	Pedro é pai de Maria
`entrega(romeu, livro, maria).`	Romeu entrega o livro a Maria

Observe que:

- nomes de relações e objetos iniciam-se com letra minúscula
- o nome da relação vem primeiro, depois vem a lista de objetos separados por vírgula e envolta em parênteses
- o ponto final é obrigatório ao final do fato.

Terminologia: relações são *predicados* e os objetos a que se referem são seus *argumentos*. Chamaremos de *banco de dados* à coleção de fatos e regras que damos a Prolog para resolver um certo problema.

1.4 Perguntas

Uma pergunta em Prolog tem a forma:

```
?- possui(maria, livro).
```

Estamos perguntando se Maria possui o livro. Prolog tenta *unificar* o fato da pergunta com os fatos do banco de dados. Dois fatos se unificam se têm o mesmo predicado e os mesmos argumentos na mesma ordem. Se Prolog achar um fato que unifica com a pergunta, vai responder "sim". Caso contrário, responderá "não".

Perceba que a resposta "não" em Prolog não significa necessariamente que o fato não é verdadeiro, mas simplesmente que Prolog não consegue *provar* o fato a partir de seu banco de dados. Confira isto no seguinte exemplo. Considere o banco de dados:

```
humano(socrates).
humano(aristoteles).
ateniense(socrates).
```

e a pergunta:

```
?- ateniense(aristoteles).
```

Embora seja verdade que Aristóteles tenha sido ateniense, não se pode provar isto a partir dos fatos dados.

1.5 Variáveis

As variáveis em Prolog servem para responder questões mais elaboradas, por exemplo "Do que Maria gosta?" ou então "Quem mora em Atenas?".

Variáveis distingüem-se dos objetos por terem nomes iniciados com letra maiúscula. Considere o seguinte banco de dados:

```
gosta(maria, flores).
gosta(maria, pedro).
gosta(paulo, maria).
```

Se fizermos a pergunta:

```
?- gosta(maria, X).
```

estaremos perguntando "Do que Maria gosta?". Prolog responde:

```
X = flores
```

e fica esperando por mais instruções, das quais falaremos em breve. Novamente, Prolog busca fatos que unifiquem com o fato da pergunta. A diferença é que uma variável está presente agora. Variáveis não instanciadas, como é o caso de `X` inicialmente, unificam com qualquer termo. Prolog examina os fatos na ordem em que aparecem no banco de dados, e portanto o fato `gosta(maria, flores)` é encontrado primeiro. A variável `X` unifica com `flores` e a partir deste momento passa a estar *instanciada* com o termo `flores`. Prolog também *marca* a posição no banco de dados onde a unificação foi feita.

Quando Prolog encontra um fato que unifica com uma pergunta, Prolog mostra os objetos que as variáveis guardam. No caso em questão, só há uma variável, que foi unificada com o objeto `flores`, então Prolog respondeu `X = flores`. Neste ponto Prolog aguarda novas instruções. Se teclarmos ENTER, isto será interpretado como significando que estamos satisfeitos com a resposta dada e Prolog interrompe a busca. Se em lugar disto teclarmos um ponto-e-vírgula (;) seguindo de ENTER, Prolog vai continuar sua busca *do ponto onde tinha parado*, tentando encontrar uma outra resposta à pergunta. Quando Prolog começa a busca a partir de uma posição previamente mercada ao invés de começar do início do banco de dados, dizemos que está tentando *ressatisfazer* a pergunta.

Suponha que peçamos a Prolog continuar a busca teclando ";" e ENTER, ou seja, queremos ver a pergunta satisfeita de outra forma, com outro valor para `X`. Isto significa que Prolog deve agora esquecer que `X` vale `flores` e continuar a busca com `X` voltando a estar não instanciada. A busca prossegue do ponto marcado no banco de dados. O próximo fato unificante é `gosta(maria, pedro)`. A variável `X` é agora instanciada a `pedro` e Prolog move a marca para o fato `gosta(maria, pedro)`, respondendo `X = pedro` e aguardando mais instruções. Se pedirmos continuação com ";" e ENTER, não há neste caso mais respostas possíveis, e por isso Prolog respoderá "não".

Vejamos rapidamente um outro exemplo sobre o mesmo banco de dados.

```
?- gosta(paulo, X).      pergunta inicial
X = maria ;              primeira resposta
no                       não há mais respostas
```

1.6 Conjunções

Questões mais complexas como "Será que Pedro gosta de Maria e Maria gosta de Pedro?" podem ser feitas com conjunções. Este problema consiste de duas metas separadas que Prolog deve tentar satisfazer, e existe uma notação para isto na linguagem. Considere o banco de dados:

```
gosta(maria, chocolate).
gosta(maria, vinho).
```

```
gosta(pedro, vinho).
gosta(pedro, maria).
```

A pergunta

```
?- gosta(pedro, maria), gosta(maria, pedro).
```

significa "Será Pedro gosta de Maria *e* Maria gosta de Pedro?". A vírgula é pronunciada "e" e serve para separar um número qualquer de metas diferentes que devem ser satisfeitas para responder a uma pergunta, o que se chama de *conjunção*. Prolog tentará satisfazer as metas uma a uma. No caso em questão, a resposta será "não", pois, embora a primeira meta seja um fato, a segunda não pode ser provada.

Conjunções combinadas com variáveis podem responder a perguntas bastante interessantes, por exemplo: "Há algo de que ambos Maria e Pedro gostam?". Isto seria escrito da seguinte forma:

```
?- gosta(maria, X), gosta(pedro, X).
```

Prolog tenta satisfazer a primeira meta; caso consiga, manterá uma marca no banco de dados no ponto onde houve unificação e tenta a segunda meta. Se a segunda meta é também satisfeita, Prolog coloca uma outra marca para a segunda meta. Observe que cada meta tem sua própria marca no banco de dados.

Se a segunda meta falhar, Prolog tentará ressatisfazer a meta anterior (neste caso, a primeira) a partir da marca desta meta. O caso em questão é ilustrativo do que chamamos de "backtracking" pois os seguintes eventos ocorrem:

1. a primeira meta encontra o fato unificador `gosta(maria, chocolate)`. A variável `X` é instanciada a `chocolate` em *todas* as ocorrências de `X` na pergunta. Prolog marca esta posição para a primeira meta e a instanciação de `X`.

2. a segunda meta, que virou `gosta(pedro, chocolate)` devido à instanciação de `X`, não unifica com nada no banco de dados, e por isso a meta falha. Prolog então tentará ressatistazer a meta anterior do ponto onde esta parou no banco de dados, desfazendo instanciações associadas.

3. Na ressatisfação da primeira meta, o próximo fato unificador é

   ```
   gosta(maria, vinho).
   ```

   Prolog move a marca para a nova posição e instancia `X` a `vinho`.

4. Prolog tenta a próxima meta, que agora é `gosta(pedro, vinho)`. Esta não é uma ressatisfação, mas sim uma meta inteiramente nova, e portanto a busca começa do início do banco de dados. Esta nova meta é satisfeita e Prolog coloca a marca desta meta no fato unificador.

5. Todas as metas da pergunta estão satisfeitas agora. Prolog imprime as instanciações de variáveis:

   ```
   X = vinho
   ```

   e aguarda instruções.

Resumindo, cada meta tem seu próprio marcador no banco de dados, que é usado caso a meta precise ser ressatisfeita. Junto com uma unificação ficam guardadas instanciações de variáveis dela resultantes, que serão desfeitas em caso de necessidade de ressatisfação. A este processo se dá o nome de *backtracking*.

1.7 Regras

Uma *regra* é uma afirmação geral sobre objetos e seus relacionamentos. Por exemplo, suponha que queremos representar a seguinte dependência entre fatos:

> Pedro gosta de todo mundo que gosta de vinho,

o que pode ser reescrito como:

> Pedro gosta de `X` se `X` gosta de vinho.

Em Prolog, regras consistem de uma *cabeça* e um *corpo*. A cabeça e o corpo são conectados pelo símbolo "`:-`" formado por dois pontos e hífen. O "`:-`" procuncia-se "se". A dependência acima seria escrita como:

```
gosta(pedro, X) :- gosta(X, vinho).
```

Note que regras também terminam com ponto final.

A cabeça desta regra é `gosta(pedro, X)`. A cabeça de uma regra descreve o que está sendo definido. O corpo, no caso `gosta(X, vinho)`, é uma conjunção de metas que devem ser satisfeitas para que a cabeça seja considerada verdadeira. Por exemplo, podemos tornar Pedro mais exigente sobre o que ele gosta adicionando mais metas ao corpo da regra:

```
gosta(pedro, X) :- gosta(X, vinho), gosta(X, menta).
```

ou, em outras palavras, "Pedro gosta de qualquer um que goste de vinho e de menta". Ou então, supondo que Pedro gosta de mulheres que gostam de vinho:

```
gosta(pedro, X) :- mulher(X), gosta(X, vinho).
```

Note que a mesma variável `X` ocorre três vezes na regra. Dizemos que o *escopo* de `X` é a regra toda. Isto significa que quando `X` for instanciada, as três ocorrências terão o mesmo valor.

A maneira como as regras funcionam em relação à satisfação de metas é a seguinte. Uma meta unifica com uma regra se ela unifica com a cabeça da regra. Agora, para verificar a veracidade da regra, o corpo é usado. Diferentemente dos fatos, onde basta haver unificação para que a meta seja satisfeita, no caso de uma regra a unificação na verdade transfere a verificação da satisfação para a conjunção de metas que formam o corpo da regra.

Vamos ilustrar este procedimento com nosso próximo exemplo, que envolve a família da rainha Vitória. Usaremos o predicado `pais` com três argumentos tal que `pais(X, Y, Z)` significa que "os pais de `X` são `Y` e `Z`". O segundo argumento é a mãe e o terceiro é o pai de `X`. Usaremos também os predicados `mulher` e `homem` para indicar o sexo das pessoas.

```
homem(alberto).
homem(eduardo).
```

```
mulher(alice).
mulher(vitoria).

pais(eduardo, vitoria, alberto).
pais(alice, vitoria, alberto).
```

Definiremos agora o predicado `irma_de` tal que `irma_de(X, Y)` seja satisfeito quando `X` for irmã de `Y`. Dizemos que `X` é irmã de `Y` quando:

- `X` é mulher
- `X` tem mãe `M` e pai `P`, e
- `Y` tem os mesmos pais de `X`.

Em Prolog, isto vira:

```
irma_de(X, Y) :-
        mulher(X),
        pais(X, M, P),
        pais(Y, M, P).
```

Se perguntarmos:

```
?- irma_de(alice, eduardo).
```

as seguintes metas serão tentadas e os resultados podem ser vistos na Tabela 1.1, onde numeramos as variáveis com índices de acordo com a meta, pois, embora a mesma letra signifique a mesma variável dentro de uma regra, em metas diferentes a mesma letra significa variáveis diferentes.

Note que, ao unificar com a cabeça da regra, a meta 1 deu origem a três outras metas, denotadas pelos números 2, 3, e 4, que vieram do corpo da regra. Ao final, Prolog consegue satisfazer a meta principal e responde: `sim`. Suponha agora que queiramos saber *de quem* Alice é irmã. A pergunta adequada seria

```
?- irma_de(alice, X).
```

Tabela 1.1: Processamento da meta `irma_de(alice, eduardo)`.

N	Meta	Marca	Variáveis
1	`irma_de(alice,eduardo)`	regra 1	$\texttt{X}_1 = \texttt{alice}$, $\texttt{Y}_1 = \texttt{eduardo}$
2	`mulher(alice)`	fato 1	-
3	`pais(alice,`$\texttt{M}_1$`,`$\texttt{P}_1$`)`	fato 2	$\texttt{M}_1 = \texttt{vitoria}$, $\texttt{P}_1 = \texttt{alberto}$
4	`pais(eduardo,vitoria,alberto)`	fato 1	-

Tabela 1.2: Processamento da meta `irma_de(alice, X)`.

N	Meta	Marca	Variáveis
1	`irma_de(alice,X)`	regra 1	$\texttt{X}_1 = \texttt{alice}$, $\texttt{Y}_1 = \texttt{X}$
2	`mulher(alice)`	fato 1	-
3	`pais(alice,`$\texttt{M}_1$`,`$\texttt{P}_1$`)`	fato 2	$\texttt{M}_1 = \texttt{vitoria}$, $\texttt{P}_1 = \texttt{alberto}$
4	`pais(`$\texttt{Y}_1$`,vitoria,alberto)`	fato 1	$\texttt{Y}_1 = \texttt{eduardo}$

O que ocorre então está na Tabela 1.2, onde `X` sem índice indicará a variável da pergunta.

Observe que $\texttt{X} = \texttt{Y}_1 = \texttt{eduardo}$ e portanto Prolog responde:

```
X = eduardo
```

e fica aguardando novas instruções. O que acontecerá se pedirmos respostas alternativas? Veja o exercício a seguir.

Em geral, um mesmo precicado é definido através de alguns fatos e algumas regras. Usaremos a palavra *cláusula* para nos referirmos a fatos e regras coletivamente.

Exercícios

1. Descreva o que acontece se forem pedidas respostas alternativas no exemplo envolvendo o predicado `irma_de` acima. Este é o comportamento esperado? Como consertar a regra, supondo que existe um predicado `dif(X,Y)` que é satisfeito quando `X` e `Y` são diferentes?

Capítulo 2

Sintaxe e unificação

Em Prolog há apenas um tipo, chamado de *termo*, que engloba todas as construções sintáticas da linguagem. Neste capítulo vamos estudar com algum detalhe este tipo e seus subtipos.

Um termo pode ser uma *constante*, uma *variável*, ou uma *estrutura*.

2.1 Constantes

As *constantes* pode ser *átomos* ou *números*. Quem conhece LISP notará uma diferença aqui: os números são considerados átomos em LISP, mas em Prolog números não são átomos (embora ambos sejam constantes).

Um *átomo* indica um objeto ou uma relação. Nomes de objetos como `maria`, `livro`, etc. são átomos. Nomes de átomos sempre começam com letra minúscula. Nomes de predicados são sempre atômicos também. Os grupos de caracteres `?-` (usado em perguntas) e `:-` (usado em regras) são também átomos. Átomos de comprimento igual a um são os *caracteres*, que podem ser lidos e impressos em Prolog, como veremos no capítulo sobre entrada e saída.

Em relação a *números*, Prolog acompanha as outras linguagens, permitindo inteiros positivos e negativos, números em ponto flutuante usando ponto decimal e

opcionalmente expoente de dez. Alguns exemplos de números válidos:

```
0, 1, -17, 2.35, -0.27653, 10e10, 6.02e-23
```

2.2 Variáveis

Sintaticamente, as *variáveis* têm nomes cujo primeiro caractere é uma letra maiúscula ou o sinal de sublinhado (*underscore*) "_". Estas últimas são chamadas de variáveis *anônimas*.

Variáveis com o mesmo nome aparecendo numa mesma cláusula são a mesma variável, ou seja, se uma ganha um valor, este valor passa imediatamente para as outras ocorrências, exceto para variáveis anônimas. As variáveis anônimas são diferentes das outras nos seguintes aspectos: (1) cada ocorrência delas indica uma variável diferente, mesmo dentro de uma mesma cláusula, e (2) ao serem usadas numa pergunta, seus valores não são impressos nas respostas. Variáveis anônimas são usadas quando queremos que unifiquem com qualquer termo mas não nos interessa com qual valor serão instanciadas.

2.3 Estruturas

As *estruturas* são termos mais complexos formados por um *funtor* seguido de *componentes* separadas por vírgula e colocadas entre parênteses. Por exemplo, para indicar um livro com seu título e autor podemos usar a estrutura abaixo:

```
livro(incidente_em_antares, verissimo)
```

Observe que os fatos de um banco de dados em Prolog são estruturas seguidas de um ponto final.

Estruturas podem ser aninhadas. Se quisermos sofisticar a indicação dos livros, colocando nome e sobrenome do autor para poder diferenciar entre vários autores com o mesmo sobrenome, podemos usar:

```
livro(incidente_em_antares, autor(erico, verissimo))
```

Estruturas podem ser argumentos de fatos no banco de dados:

```
pertence(ze, livro(incidente_em_antares, verissimo)).
```

O número de componentes de um funtor é a sua *aridade*. Funtores de aridade igual a zero são na verdade as constantes. Quando não há argumentos, a estrutura é escrita sem os parênteses.

Às vezes é conveniente escrever certas estruturas na forma infixa ao invés de prefixa. Quando isto acontece, dizemos que o funtor é escrito como *operador*. Um operador tem na verdade três propriedades que devem ser especificadas: sua posição, sua precedência e sua associatividade. A posição pode ser prefixa, infixa ou posfixa. A precedência é um número; quanto menor for, mais prioridade o operador terá nos cálculos. A associatividade pode ser esquerda ou direita, e indica como devem ser agrupadas subexpressões consistindo apenas deste operador.

Por exemplo, os operadores aritméticos `+`, `-`, `*`, `/` têm geralmente posição infixa. O operador unário de negação é em geral prefixo, e o operador de fatorial (`!`) é em geral posfixo. A precedência de `*` e `/` é maior que de `+` e `-`. A associatividade de todos os operadores aritméticos é esquerda, o que significa que uma expressão como `8/2/2` será interpretada como `(8/2)/2` e não como `8/(2/2)`.

Note que Prolog é diferente das outras linguagens no aspecto aritmético, pois uma expressão como `2 + 3` é simplesmente um fato como qualquer outro. Para a realização de cálculos aritméticos é sempre necessário usar o predicado `is` que será visto no capítulo sobre aritmética.

2.4 Igualdade e unificação

Em Prolog, igualdade significa unificação. A linguagem pré-define um predicado infixo "=" para a igualdade, mas em geral seu uso pode ser substituído pelo uso de variáveis de mesmo nome. Se não existisse, o predicado "=" poderia ser definido por apenas um fato:

```
X = X.
```

Uma outra característica da igualdade em Prolog, já que ela significa unificação, é que ela pode causar a instanciação de algumas variáveis, como temos visto nos exemplos introdutórios.

Em geral, dada uma meta da forma $T_1 = T_2$, onde T_1 e T_2 são termos quaisquer, a decisão sobre sua igualdade é feita de acordo com a Tabela 2.1. Nesta tabela são tratados os casos onde cada termo pode ser uma constante, variável, ou estrutura. Note que quando dizemos "variável" estamos na verdade nos referindo a variáveis não instanciadas, pois para variáveis instanciadas deve se usar o valor a que estão associadas para consultar a Tabela 2.1.

		T_2		
		constante	estrutura	variável
T_1	constante	só se for a mesma	nunca	sempre; causa instanciação
	estrutura	nunca	mesmo funtor, mesmo número de componentes e cada componente igual	sempre; causa instanciação
	variável	sempre; causa instanciação	sempre; causa instanciação	tornam-se ligadas

Tabela 2.1: Condições para que dois termos T_1 e T_2 se unifiquem, segundo seus subtipos. O subtipo "variável" significa "variável não instanciada". O subtipo "estrutura" significa estruturas com aridade maior ou igual a um.

Exercícios

1. Decida se as unificações abaixo acontecem, e quais são as instanciações de variáveis em caso haja unificação.

```
pilotos(A, londres) = pilotos(londres, paris)
pilotos(londres, A) = pilotos(londres, paris)
pilotos(A, londres) = pilotos(paris, londres)
```

```
ponto(X, Y, Z) = ponto(X1, Y1, Z1)
letra(C) = palavra(letra)
letra(C) = letra(palavra)
nome(alfa) = alfa
f(X, X) = f(a,b)
f(X, a(b,c)) = f(Z, a(Z,c))
```

2. Como se pode definir o predicado abaixo sem usar igualdade?

```
irmaos(X, Y) :-
    pai(X, PX),
    pai(Y, PY),
    PX = PY.
```

Capítulo 3

Aritmética

Prolog tem uma série de predicados pré-definidos para artimética, que podem ser divididos entre comparação e cálculo. Vamos começar pelos de comparação.

Os predicados de comparação são infixos e comparam apenas números ou variáveis instanciadas a números. São eles:

`=:=`	igual
`=\=`	diferente
`<`	menor
`>`	maior
`=<`	menor ou igual
`>=`	maior ou igual

Note que em Prolog o comparador "menor ou igual" é `=<`, e não `<=` como na maioria das linguagens. Isto foi feito para liberar o predicado `<=`, que parece uma seta, para outros usos pelo programador. Os predicados pré-definidos não pode ser redefinidos nem podem ter fatos ou regras adicionados a eles, por exemplo, tentar acrescentar um fato como `2 < 3.` é ilegal, mesmo que seja verdadeiro.

Para exemplificar o uso dos comparadores, considere o seguinte banco de dados contendo os príncipes de Gales nos séculos 9 e 10, e os anos em que reinaram. Os nomes estão em galês.

```
reinou(rhodri, 844, 878).
reinou(anarawd, 878, 916).
reinou(hywel_dda, 916, 950).
reinou(iago_ap_idwal, 950, 979).
reinou(hywal_ap_ieuaf, 979, 985).
reinou(cadwallon, 985, 986).
reinou(maredudd, 986, 999).
```

Quem foi o príncipe em um ano dado? A seguinte regra tenciona responder isto.

```
principe(X, Y) :-
    reinou(X, A, B),
    Y >= A,
    Y =< B.
```

Alguns usos:

```
?- principe(cadwallon, 986).
yes

?- principe(X, 900).
X = anarawd
yes

?- principe(X, 979).
X = iago_ap_idwal ;
X = hywel_ap_ieuaf ;
no
```

Para cálculos aritméticos, o predicado especial pré-definido `is` deve ser usado. O seu papel é transformar uma estrutura envolvendo operadores aritméticos no resultado desta expressão. O operado `is` é infixo, e de seu lado direito deve sempre aparecer uma expressão aritmética envolvendo apenas números ou varáveis instanciadas com números. Do lado esquerdo pode aparecer uma variável não instanciada, que será então instanciada com o resultado da expressão, ou pode também aparecer um número ou variável instanciada a um número, caso em que

`is` testa se o lado esquerdo e direito são iguais, servindo assim como operador de igualdade numérica.

O predicado `is` é o único que tem o poder de calcular resultados de operações aritméticas.

Para exemplificar, considere o seguinte banco de dados sobre a população e a área de diversos países em 1976. O predicado `pop` representará a população de um país em milhões de pessoas, e o predicado `area` informará a área de um país em milhões de quilômetros quadrados:

```
pop(eua, 203).
pop(india, 548).
pop(china, 800).
pop(brasil, 108).

area(eua, 8).
area(india, 3).
area(china, 10).
area(brasil, 8).
```

Para calcular a densidade populacional de um país, escrevemos a seguinte regra, que divide a população pela área:

```
dens(X, Y) :-
    pop(X, P),
    area(X, A),
    Y is P/A.
```

Alguns usos:

```
?- dens(china, X).
X = 80
yes

?- dens(turquia, X).
no
```

Os operadores para cálculos aritméticos em Prolog incluem pelo menos os seguintes:

`+`	soma
`-`	subtração
`*`	multiplicação
`/`	divisão
`//`	divisão inteira
`mod`	resto da divisão

Exercícios

1. Considere o seguinte banco de dados:

```
soma(5).
soma(2).
soma(2 + X).
soma(X + Y).
```

   e a meta

```
soma(2 + 3)
```

   Com quais dos fatos esta meta unifica? Quais são as instanciações de variáveis em cada caso?

2. Quais são os resultados das perguntas abaixo?

```
?- X is 2 + 3.
?- X is Y + Z.
?- 6 is 2 * 4.
?- X = 5, Y is X // 2.
?- Y is X // 2, X = 5.
```

Capítulo 4

Estruturas de dados e recursão

As estruturas de Prolog são muito versáteis para represetar estruturas de dados nos programas. Neste capítulo veremos alguns exemplos.

4.1 Estruturas como árvores

As estruturas podem ser desenhadas como árvores, onde o funtor fica na raiz e os componentes são seus filhos, como na Figura 4.1.

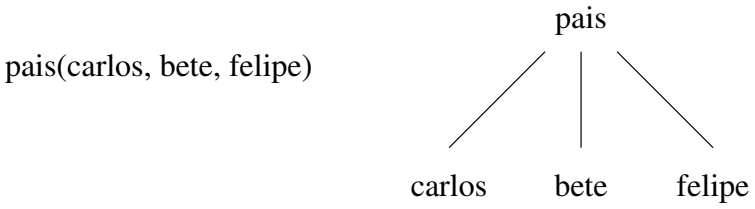

Figura 4.1: Estruturas como árvores.

Frases da língua portuguesa podem ter suas sintaxes representadas por estruturas em Prolog. Um tipo de sentença muito simples, consisitindo de sujeito e predicado (não confundir com predicado Prolog!), poderia ser:

```
sentenca(sujeito(X), predicado(verbo(Y), objeto(Z)))
```

Tomando a sentença “Pedro ama Maria” e instanciando as variáveis da estrutura com palavras da sentença, ficamos com o resultado mostrado na Figura 4.2.

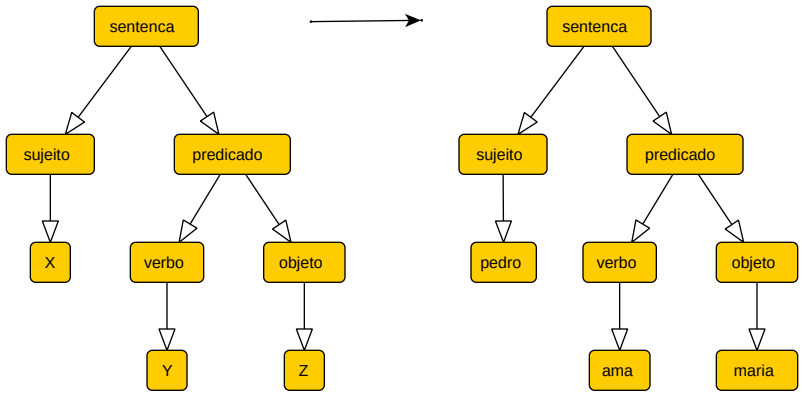

Figura 4.2: Sintaxe de sentenças.

A representação de estruturas pode também dar uma descrição pictótica das variáveis, mostrando ocorrências de uma mesma variável. Por exemplo, a estrutura `f(X, g(X, a))` poderia ser representada como na Figura 4.3. A figura não é mais uma árvore, mas sim um grafo orientado acíclico.

4.2 Listas

As listas são estruturas de dados importantes em Prolog. Existem diversos predicados pré-definidos para lidar com listas em Prolog. Existem também predicados para transformar estruturas em listas e vice-versa.

Em Prolog, Uma lista é:

- ou uma *lista vazia*, escrita como `[ ]`,
- ou uma estrutura com dois componentes: a *cabeça* e a *cauda*.

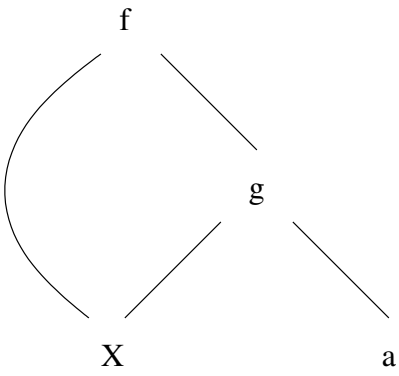

Figura 4.3: Diagrama mostrando ocorrências da mesma variável.

O funtor (binário) usado para representar a estrutura de lista é o ponto "`.`". Para quem conhece LISP, este funtor lembra o par-com-ponto. Na Figura 4.4 vemos uma lista que seria escrita como `.(a, .(b, .(c, [ ])))`, em Prolog. Em LISP, esta mesma lista seria representada como `(a . (b . (c . ())))`.

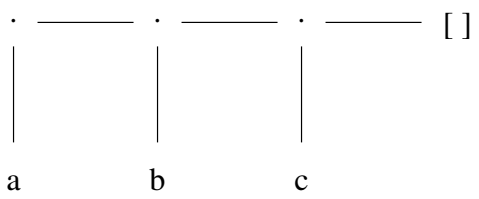

Figura 4.4: Uma lista em Prolog.

A exemplo de LISP, Prolog também tem uma maneira alternativa, mais prática, de denotar listas, que não usa o funtor "`.`" explicitamente. Basta colocar os elementos separados por vírgulas entre colchetes: `[a, b, c]`. Qualquer termo pode ser componente de uma lista, por exemplo, variáveis ou outras listas:

```
[o, homem, [gosta, de, pescar]]
[a, V1, b, [X, Y]]
```

Listas são processadas dividindo-as em cabeça e cauda (exceto a lista vazia). Para

quem conhece LISP, esta divisão é análoga a tomar car e cdr de uma lista. Observe alguns exemplos:

Lista	Cabeça	Cauda
`[a, b, c]`	`a`	`[b, c]`
`[ ]`	não tem	não tem
`[[o, gato], sentou]`	`[o, gato]`	`[sentou]`
`[o, [gato, sentou]]`	`o`	`[[gato, sentou]]`
`[o, [gato, sentou], ali]`	`o`	`[[gato, sentou], ali]`
`[X + Y, x + y]`	`X + Y`	`[x + y]`

Para combinar com a notação simplificada para listas, Prolog também tem uma notação especial, mais intuitiva, para indicar `.(X, Y)` usando a barra vertical "`|`": `[X|Y]`. Esta notação é muito usada para decompor uma lista em cabeça e cauda, como no exemplo abaixo, onde aparecem um banco de dados e algumas perguntas:

```
p([1, 2, 3]).
p([o, gato, sentou, [no, capacho]]).

?- p([X|Y]).
X = 1 , Y = [2, 3] ;
X = o, Y = [gato, sentou, [no, capacho]] ;
no

?- p([_,_,_,[_|X]]).
X = [capacho]
```

Uma última observação: é possível criar estruturas que parecem listas mas não são, por exemplo, `[cavalo|branco]`, que não é lista porque sua cauda não é uma lista nem é a lista vazia.

Exercícios

1. Decida se as unificações abaixo acontecem, e quais são as instanciações de variáveis em cada caso positivo.

```
[X, Y, Z] = [pedro, adora, peixe]
[gato] = [X|Y]
[X, Y|Z] = [maria, aprecia, vinho]
[[a, X]|Z] = [[X, lebre], [veio, aqui]]
[anos|T] = [anos, dourados]
[vale, tudo] = [tudo, X]
[cavalo|Q] = [P|branco]
[ ] = [X|Y]
```

4.3 Recursão

Suponha que tenhamos uma lista de cores preferidas, por exemplo

```
[azul, verde, vermelho, amarelo]
```

e queiramos saber se uma determinada cor está nesta lista. A maneira de fazer isto em Prolog é ver se a cor está na cabeça da lista; se estiver, ficamos satisfeitos; se não estiver, procuramos na cauda da lista, ou seja, verificamos a cabeça da *cauda* agora. E a cabeça da cauda da cauda a seguir. Se chegarmos ao fim da lista, que será a lista vazia, falhamos: a cor não estava na lista inicial.

Para implementar isto em Prolog, primeiramente estabelecemos que trata-se de definir uma *relação* entre objetos e listas onde eles aparecem. Escreveremos um predicado `member` tal que `member(X, Y)` é verdadeiro quando o termo `X` é um elemento da lista `Y`. Há duas condições a verificar. Em primeiro lugar, é um fato que `X` é membro de `Y` se `X` for igual à cabeça de `Y`, o que pode ser escrito assim:

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

ou, simplificando,

```
member(X, [X|_]).
```

Note o uso da variável anônima. Neste fato, não nos interessa o que é a cauda de `Y`.

A segunda (e última) regra diz que `X` é membro de `Y` se `X` é membro da cauda de `Y`. Aqui entrará recursão para verificar se `X` está na cauda. Veja como fica:

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

ou, simplificando,

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

Observe novamente o uso da variável anônima. Nesta regra, não nos interessa o que está na cabeça da lista, que foi tratada na primeira cláusula do predicado `member`. Observe ainda que trocamos `Z` por `Y`, já que `Y` sumiu na simplificação.

O que escrevemos foi basicamente uma definição do predicado `member`, porém a maneira de processar perguntas de Prolog faz com que esta "definiçao" possa ser usada computacionalmente. Exemplos:

```
?- member(d, [a, b, c, d, e, f, g]).
yes
?- member(2, [3, a, d, 4]).
no
```

Para definir um predicado resursivo, é preciso atentar para as *condições de parada* e para o *caso recursivo*. No caso de `member`, há na verdade duas condições de parada: ou achamos o objeto na lista, ou chegamos ao fim dela sem achá-lo. A primeira condição é tratada pela primeira cláusula, que fará a busca parar se o primeiro argumento de `member` unificar com a cabeça do segundo argumento. A segunda condição de parada ocorre quando o segundo argumento é a lista vazia, que não unifica com nenhuma das cláusulas e faz o predicado falhar.

Em relação ao caso recursivo, note que a regra é escrita de tal forma que a chamada recursiva ocorre sobre uma lista *menor* que a lista dada. Assim temos certeza de acabaremos por encontrar a lista vazia, a menos é claro que encontremos o objeto antes.

Certos cuidados devem ser tomados ao fazer definições recursivas. Um deles é evitar circularidade, por exemplo:

```
pai(X, Y) :- filho(Y, X).
filho(X, Y) :- pai(Y, X).
```

Claramente, Prolog não conseguirá inferir nada a partir destas definições, pois entrará num *loop* infinito.

Outro cuidado é com a ordem das cláusulas na definição de um predicado. Considere a seguinte definição:

```
homem(X) :- homem(Y), filho(X, Y).
homem(adao).
```

Ao tentar responder à pergunta

```
?- homem(X).
```

Prolog estrará em *loop* até esgotar a memória. O predicado `homem` está definido usando *recursão esquerda*, ou seja, a chamada recursiva é a meta mais à esquerda no corpo, gerando uma meta equivalente à meta original, o que se repetirá eternamente até acabar a memória. Para consertar o predicado, basta trocar a ordem das cláusulas:

```
homem(adao).
homem(X) :- homem(Y), filho(X, Y).
```

Em geral, é aconselhável colocar os fatos antes das regras. Os predicados podem funcionar bem em chamadas com constantes, mas dar errado em chamadas com variáveis. Considere mais um exemplo:

```
lista([A|B]) :- lista(B).
lista([ ]).
```

É a própria definição de lista. Ela funciona bem com constantes:

```
?- lista([a, b, c, d]).
yes
?- lista([ ]).
yes
?- lista(f(1, 2, 3)).
no
```

Mas, se perguntarmos

```
?- lista(X).
```

Prolog entrará em *loop*. Mais uma vez, inverter a ordem das cláusulas resolverá o problema.

4.4 Juntando listas

O predicado `append` é usado para juntar duas listas formando uma terceira. Por exemplo, é verdade que

```
append([a, b, c], [1, 2, 3], [a, b, c, 1, 2, 3]).
```

Este predicado pode ser usado para criar uma lista que é a concatenação de duas outras:

```
?- append([alfa, beta], [gama, delta], X).
X = [alfa, beta, gama, delta]
```

Mas também pode ser usado de outras formas:

```
?- append(X, [b, c, d], [a, b, c, d]).
X = [a]
```

A sua definição é a seguinte:

```
append([ ], L, L).
append([X|L1], L2, [X|L3]) :- append(L1, L2, L3).
```

A primeira cláusula é a condição de parada. A lista vazia concatenada com qualquer lista resulta na própria lista. A segunda condição se apóia nos seguintes princípios:

1. O primeiro elemento da primeira lista será também o primeiro elemento da terceira lista.
2. Concatenando a cauda da primeira lista com a segunda lista resulta na cauda da terceira lista.
3. Temos que usar o próprio `append` para obter a concatenação do item 2 acima.
4. Conforme aplicamos a segunda cláusula, o primeiro argumento vai diminuindo, até ser a lista vazia; logo, a recursão termina.

4.5 Acumuladores

Freqüentemente, precisamos percorrer uma estrutura em Prolog e calcular resultados que dependem do que já foi encontrado até o momento. A técnica de acumuladores consiste em utilizar um ou mais argumentos do predicado para representar "a resposta até o momento" durante este percurso. Estes argumentos recebem o nome de *acumuladores*.

No próximo exemplo mostraremos uma definição para o predicado `listlen` sem acumulador e depois com acumulador. A meta `listlen(L, N)` é satisfeita quando o comprimento da lista `L` é igual a `N`. Prolog possui o predicado pré-definido `length` para esta finalidade, mas vamos definir nossa própria versão. Eis a definição sem acumuladores:

```
listlen([ ], 0).
listlen([H|T], N) :- listlen(T, N1), N is N1 + 1.
```

Observe que o primeiro argumento deve vir instanciado para que esta definição funcione. A definição com acumulador baseia-se no mesmo princípio recursivo, mas acumula a resposta a cada passo num argumento extra. Usamos um predicado auxiliar `lenacc` que é uma generalização de `listlen`. A meta `lenacc(L, A, N)` é satisfeita quando o comprimento de `L` adicionado ao número `A` resulta em `N`. Para obter o comprimento da lista, basta chamar `lenacc` com o segundo argumento igual a zero. Eis as definições:

```
lenacc([ ], A, A).
lenacc([H|T], A, N) :- A1 is A + 1, lenacc(T, A1, N).

listlenacc(L, N) :- lenacc(L, 0, N).
```

Acompanhe a seqüência de submetas criadas para calcular o comprimento de `[a, b, c, d, e]`:

```
listlenacc([a, b, c, d, e], N)
lenacc([a, b, c, d, e], 0, N)
lenacc([b, c, d, e], 1, N)
lenacc([c, d, e], 2, N)
lenacc([d, e], 3, N)
lenacc([e], 4, N)
lenacc([ ], 5, N)
```

onde todos os `N` são variáveis distintas mas ligadas entre si no procesamento das regras definidas. Esta última meta unifica com a condição de parada (a primeira cláusula de `lenacc`) e instancia `N` como 5, o que faz com que o `N` de `listlen` seja também igual a 5.

Acumuladores não precisam ser números inteiros. Considere a seguinte definição do predicado `rev`, que inverte a ordem dos elementos de uma lista (Prolog tem sua própria versão chamada `reverse`):

```
rev(L1, L3) :- revacc(L1, [ ], L3).

revacc([ ], L3, L3).
revacc([H|L1], L2, L3) :- revacc(L1, [H|L2], L3).
```

O segundo argumento de `revacc` serve como acumulador. Observe a seqüência de metas usadas para responder à pergunta `?- rev([a, b, c, d], L3).`:

```
rev([a, b, c, d], L3)
revacc([a, b, c, d], [ ], L3)
revacc([b, c, d], [a], L3)
revacc([c, d], [b, a], L3)
revacc([d], [c, b, a], L3)
revacc([ ], [d, c, b, a], L3)
```

Nestas metas todas as variáveis `L3` são distintas mas ligadas. Os elementos vão saindo de `L1` e entrando em `L2` a cada chamada. Os elementos saem e entram pela frente nas listas, combinando com as operações de lista em Prolog que só permitem manipular diretamente a cabeça. No final, a cláusula de parada instancia `L3` à reversa.

Ao lidar com acumuladores, é muito importante não perder de vista o *significado* dos predicados a definir. Neste exemplo, é essencial entender o que significa o predicado `revacc`: a meta `revacc(L1, L2, L3)` é satisfeita quando a reversa da lista `L1` concatenada com a lista `L2` resulta em `L3`. De posse desta definição em palavras é mais fácil chegar à definição em Prolog.

Exercícios

1. Escreva um predicado `last(L, X)` que é satisfeito quando o termo `X` é o último elemento da lista `L`.

2. Escreva um predicado `efface(L1, X, L2)` que é satisfeito quando `L2` é a lista obtida pela remoção da primeira ocorrência de `X` em `L1`.

3. Escreva um predicado `delete(L1, X, L2)` que é satisfeito quando `L2` é a lista obtida pela remoção de todas as ocorrências de `X` em `L1`.

Capítulo 5

Backtracking e o "corte"

Resumindo o que vimos sobre a operação de Prolog até agora:

1. uma pergunta é a conjunção de várias metas, que chamaremos de 1, 2, 3, . . . , n. Estas metas são processadas na ordem dada na tentativa de satisfazê-las.

2. a tentativa de satisfação de uma meta k consiste numa busca no banco de dados, a partir do início, por uma cláusula unificante. Se não houver tal cláusula, a meta falha. Se houver cláusula unificante, marca-se o ponto no banco de dados onde ela ocorre, e instanciam-se e ligam-se as variáveis conforme necessário. Dizemos neste caso que a meta *casou*. Se a cláusula unificante for um fato, a meta é satisfeita. Se for a cabeça de uma regra, a meta dá origem a um novo nível de submetas, criadas a partir da cauda da regra, que chamaremos de $k.1$, $k.2$, etc. A satisfação de k agora depende da satisfação conjunta de todas as submetas.

3. quando uma meta é satisfeita, passa-se para a próxima meta. Se não houver próxima, Prolog pára e informa o resultado (positivo), juntamente com os valores das variáveis da pergunta.

4. quando uma meta falha, a meta anterior sofre tentativa de ressatisfação. Chamamos esta ação de *backtracking* (retrocesso). Se não existir meta anterior, Prolog pára e informa o resultado (negativo) da busca.

5. a tentativa de ressatisfação de uma meta é semelhante à de satisfação, com as seguintes ressalvas:

   (a) desfazem-se as instanciações e ligações causadas pela última unificação desta meta.

   (b) a busca continua do ponto marcado no banco de dados ao invés de começar do ínicio;

Neste capítulo vamos examinar o processo de *backtracking* com mais detalhe e conhecer o "corte", um mecanismo especial que inibe o *backtracking* em certas condições.

A notação "*k.m*" para indicar metas geradas por um casamento da meta *k* com uma regra é interessante para saber quem é a "meta-mãe" de cada meta gerada, o que vai ser importante no nosso estudo do corte. Porém, em certas situações, podemos preferir simplesmente usar uma numeração sequencial para as metas, como fazem alguns depuradores de sistemas Prolog.

5.1 Gerando múltiplas soluções

Considere o seguinte banco de dados.

```
pai(maria, jorge).
pai(pedro, jorge).
pai(sueli, haroldo).
pai(jorge, eduardo).
pai(X) :- pai(_,X).
```

Há dois predicados `pai`: um binário, cujos argumentos são uma pessoa e seu pai, nesta ordem, e um unário, que é baseado no outro, e diferencia pais de não-pais. A pergunta

```
?- pai(X).
```

causará o seguinte resultado:

```
X = jorge ;
X = jorge ;
X = haroldo ;
X = eduardo ;
no
```

Note que Jorge apareceu duas vezes, pois há dois fatos onde ele aparece como pai de alguém. Talvez quiséssemos que cada pai aparecesse uma vez só, mas a operação padrão de Prolog causará o resultado acima.

Uma situação semelhante ocorre com o predicado `member` visto anteriormente, quando a lista contém repetições. Uma meta do tipo

```
member(a, [a, b, c, a, c, a, d, a, b, r, a])
```

pode ser satisfeita várias vezes antes de falhar (no caso ilustrado, cinco vezes). Há situações onde gostaríamos que ela fosse satisfeita uma vez só. Podemos instruir Prolog a descartar escolhas desnecessárias com o uso do corte.

As situações mencionadas acima envolviam limitar um número finito de alternativas a uma só. Há casos onde precisamos gerar um número infinito de alternativas, não porque pretendemos considerá-las todas, mas porque não sabemos de antemão quando vai aparecer a alternativa que nos interessa. Considere a seguinte definição de predicado:

```
inteiro(0).
inteiro(N) :- inteiro(M), N is M + 1.
```

A pergunta

```
?- inteiro(N).
```

causará a geração de todos os inteiros a partir do zero, em ordem crescente. Isto pode ser usado em parceria com outro predicado que seleciona alguns entre estes inteiros para uma determinada aplicação.

5.2 O "corte"

O "corte" é um mecanismo especial em Prolog que instrui o sistema a não reconsiderar certas alternativas ao fazer *backtracking*. Isto pode ser importante para poupar memória e tempo de processamento. Em alguns casos, o "corte" pode significar a diferença entre um programa que funciona e outro que não funciona.

Sintaticamente, o "corte" é um predicado denotado por "`!`", com zero argumentos. Como meta, ele é sempre satisfeito da primeira vez que é encontrado, e sempre falha em qualquer tentativa de ressatisfação. Além disso, como efeito colateral ele impede a ressatisfação da meta que lhe deu origem, chamada de sua *meta mãe*. Se o "corte" é a meta $k.l$, sua meta mãe é a meta k. Numa tentativa de ressatisfação do "corte", ele causa a falha da meta k como um todo, e o *backtracking* continua tentando ressatisfazer a meta anterior a k.

Vejamos um exemplo. Considere a regra

```
g :- a, b, c, !, d, e, f.
```

Prolog realiza o *backtracking* normalmente entre as metas `a`, `b` e `c` até que o sucesso de `c` cause a satisfação do "corte" e Prolog passe para a meta `d`. O processo de *backtracking* acontece normalmente entre `d`, `e` e `f`, mas se `d` em algum momento falhar, o que ocorre é que a meta envolvendo `g` que casou com esta regra falha imediatamente também.

5.3 Usos do corte

Há três usos principais do corte:

- indicar que a regra certa foi encontrada
- combinação corte-falha indicando negação
- limitar uma busca finita ou infinita

5.3.1 Confirmando a escolha certa

Os predicados em Prolog têm em geral várias cláusulas. Em alguns predicados, diferentes cláusulas são dirigidas a diferentes padrões de dados de entrada. Às vezes é possível selecionar qual regra é apropriada para cada entrada só pela sintaxe da entrada, por exemplo, uma regra com `[X|Y]` não casará com a lista vazia. Outras vezes, como quando os predicados envolvem números, isto não é possível. Nestes casos, o corte pode ajudar a fazer com que a meta só case com o caso destinado a ela.

Considere a seguinte definição de um predicado `fat(N, F)` que calcula o fatorial de um número:

```
fat(0, 1) :- !.
fat(N, F) :- N1 is N - 1, fat(N1, F1), F is F1 * N.
```

O corte aqui serve para impedir que uma meta da forma `fat(0, F)` case com a segunda cláusula em caso de ressatisfação. Veja o que ocorre com e sem o corte.

Com o corte:

```
?- fat(5, F).
F = 120 ;
no
```

Sem o corte:

```
?- fat(5, F).
F = 120 ;
```
(*loop* infinito — *out of memory*)

A definição de `fat` acima ainda não é a melhor possível. Veja na seção dos exercícios o que precisa ser feito para melhorá-la.

5.3.2 Combinação corte-falha

Existe em Prolog um predicado pré-definido sem argumentos chamado `fail` que sempre falha. Pode-se usar em seu lugar qualquer meta incodicionalmente falsa como por exemplo `0 > 1` mas é mais claro e elegante usar `fail`. Usado em combinação com o corte, ele pode implementar negação.

Suponha que precisemos de um predicado `nonmember(X, L)` que é o contrário de `member(X, L)`, ou seja, é satisfeito exatamente quando `membar(X, L)` falha, isto é, quando `X` não é membro de `L`. Eis a implementação de `nonmember` usando corte e falha:

```
nonmember(X, L) :- member(X, L), !, fail.
nonmember(_, _).
```

Ao processar uma meta da forma `nonmember(X, L)`, Prolog vai tentar a primeira cláusula. Se `member(X, L)` for satisfeito, o corte será processado e logo a seguir vem `fail`. Devido ao corte, sabemos que esta tentativa de ressatisfação vai fazer a meta `nonmember(X, L)` falhar sem tentar a segunda cláusula, que é exatamente o que queremos.

No caso de `member(X, L)` falhar, o corte não será processado e Prolog tentará a segunda cláusula, que devido às variáveis anônimas sempre é satisfeita. Conclusão: neste caso, `nonmember(X, L)` é satisfeito, que também é o que queremos.

Este mesmo método pode ser usado para implementar a negação de qualquer predicado. O uso deste artifício é tão comum que existe uma notação em Prolog para indicar esta forma de negação: `\+` antecedendo uma meta significa a negação dela. Por exemplo, podemos definir

```
nonmember(X, L) :- \+ member(X, L).
```

Contudo, em geral estas negações só funcionam para metas onde os argumentos vêm todos instanciados.

5.3.3 Limitando buscas

Muitas vezes em Prolog usamos um predicado para gerar várias alternativas que serão testadas por um segundo predicado para escolher uma delas. Em alguns casos, o predicado gerador tem a capacidade de gerar infinitas alternativas, e o corte pode ser útil para limitar esta geração.

Considere o seguinte predicado para executar divisão inteira. Os sistemas Prolog em geral têm este recurso pré-definido na linguagem, através do operador `//`, mas usaremos esta versão mais ineficiente para ilustrar o tema desta seção.

```
divide(Numerador, Denominador, Resultado) :-
        inteiro(Resultado),
        Prod1 is Resultado * Denominador,
        Prod2 is Prod1 + Denominador,
        Prod1 =< Numerador,
        Prod2 > Numerador,
        !.
```

Esta definição usa o predicado `inteiro` definido anteriormente (página 37) para gerar candidatos a quociente inteiro, que são testados pelas metas subseqüentes. Note que sem o corte teríamos um *loop* infinito em tentativas de ressatisfação.

5.4 Cuidados com o corte

Cortes têm um impacto significativo no comportamento de qualquer predicado, e por isso é necessário ter clareza sobre exatamente que uso queremos fazer de um predicado ao considerar a inclusão de cortes em sua definição.

Para exemplificar esta questão, suponha que queiramos usar `member` apenas para testar se elementos dados pertencem a listas dadas, sem nos importarmos com o número de vezes que aparecem. Neste caso, a definição

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

é apropriada. O predicado é satisfeito uma única vez para cada elemento distinto da lista. Porém, perdemos a possibilidade de usá-lo como gerador de múltiplas alternativas:

```
?- member(X, [b, c, a]).
X = b ;
no
```

O caso seguinte ilustrará como o corte pode impedir que o operador de igualdade "=" seja substituído pelo uso de variáveis com o mesmo nome. Considere o predicado abaixo, que dá o número de pais que uma pessoa tem:

```
pais(adao, 0).
pais(eva, 0).
pais(_, 2).
```

Isto é, Adão e Eva têm zero pais e qualquer outra pessoa tem dois pais. Se usarmos este predicado para descobrir quantos pais uma pessoa tem, tudo vai bem:

```
?- pais(eva, N).
N = 0
?- pais(pedro, N).
N = 2
```

Mas quando tentamos verificar uma afirmação algo inesperado acontece:

```
?- pais(eva, 2).
yes
```

pois a meta não casa com a cláusula específica para Eva, mas não há nada para impedir que a busca continue, e acabe casando com a cláusula errada.

Uma forma de arrumar isto é a seguinte:

```
pais(adao, N) :- !, N = 0.
pais(eva, N) :- !, N = 0.
pais(_, 2).
```

Agora as duas primeiras cláusulas vão "capturar" as metas envolvendo Adão e Eva e "travar" ali com o corte, deixando a verificação do valor de `N` para o corpo da regra.

A conclusão final é que ao introduzir cortes para que o predicado sirva a um certo tipo de metas, não há garantia que ele continuará funcionando a contento para outros tipos de metas.

Exercícios

1. Conserte o predicato `fat(N, F)` para que não entre em *loop* en chamadas onde `N` é um número negativo e nem em chamadas verificadoras, onde ambos `N` e `F` vêm instanciados.

2. Alguém teve a idéia de usar `nonmember` conforme definido no texto para gerar todos os termos que **não** estão na lista `[a, b, c]` com a pergunta

   ```
   ?- nonmember(X, [a, b, c]).
   ```

   Vai funcionar? Por quê?

3. Suponha que alguém queira listar os elementos comuns a duas listas usando a seguinte pergunta:

   ```
   ?- member1(X, [a, b, a, c, a]),
      member2(X, [c, a, c, a]).
   ```

   Quais serão os resultados nas seguintes situações:

   (a) `member1` sem corte e `member2` sem corte?

   (b) `member1` sem corte e `member2` com corte?

   (c) `member1` com corte e `member2` sem corte?

   (d) `member1` com corte e `member2` com corte?

Relembrando as definições com e sem corte:

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

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

Capítulo 6

Entrada e saída de dados

Neste capítulo veremos alguns dos mecanismos que Prolog oferece para a entrada e saída de dados. Dividiremos a apresentação em três partes: leitura e escrita de termos; leitura e escrita de caracteres; leitura e escrita de arquivos. Além disso, abordaremos a questão dos operadores, que influenciam o modo como a leitura e a escrita ocorrem.

A descrição baseia-se no SWI Prolog. Outros sistemas podem diferir desta implementação.

6.1 Leitura e escrita de termos

Prolog oferece o predicado pré-definido `read` para a entrada de termos. A meta `read(X)` é satisfeita quando `X` unifica com o próximo termo lido no dispositivo de entrada. É preciso colocar um ponto final para sinalizar o fim do termo, sendo que este ponto final não é considerado parte do termo lido. Unificando ou não, o termo lido é consumido, ou seja, a próxima leitura seguirá daí para a frente. O termo lido pode conter variáveis, que serão tratadas como tal, mas seu escopo se restringe ao termo lido.

Se o termo lido não tiver a sintaxe de um termo em Prolog, ocorre erro de leitura. Se o fim do arquivo for encontrado, `X` será instanciada ao átomo especial

`end_of_file`. É um erro tentar ler após encontrar o fim do arquivo. Em caso de ressatisfação, `read` falha.

O seguinte predicado lê um número do dispositivo de entrada e é satisfeito quando o número lido é menor que 50.

```
pequeno :- read(N), N < 50.
```

Exemplo:

```
?- pequeno.
|:  40.
yes
```

Observe o prompt '`|:`' usado por Prolog para indicar que está esperando um termo.

Para escrever termos, Prolog disponibiliza o predicado pré-definido `write`, que aceita um argumento e imprime no dispoditivo de saída o termo instanciado a este argumento. Se o argumento contém variáveis não instanciadas, estas serão impressas com seus nomes internos, geralmente consistindo de um "`_`" seguido de um código interno alfanumérico. Além de `write`, existe em Prolog o predicado pré-definido `nl`, sem arugmento, que causa mudança de linha na impressão (*newline*). Assim, se quisermos dividir a saída em várias linhas, devemos usar `nl`:

```
?- write(pedro), nl, write(ama), nl, write(maria).
pedro
ama
maria
yes
```

Assim como ocorre com `read`, metas dos predicados `write`, e `nl` só são satisfeitas na primeira vez.

6.2 Leitura e escrita de caracteres

Como vimos, as constantes do tipo caractere em Prolog são denotadas usando-se apóstrofes, por exemplo, `'e'`, `'\n'`, etc. Para a leitura de caracteres, Prolog oferece o predicado pré-definido `get_char(X)`, que é satisfeito unificando `X` com o próximo caractere lido do dispositivo de entrada. É possível que em alguns sistemas a entrada só venha a Prolog linha a linha, o que significa que até que seja teclado um ENTER o interpretador não recebe nenhum caractere. Assim como acontece com `read`, o caractere lido é consumido independentemente de `get_char(X)` ser satisfeito ou não. O predicado `get_char` falha em tentativas de ressatisfação. Se chegarmos ao fim do arquivo, o átomo especial `end_of_file` é retornado.

A título de exemplo, eis abaixo um predicado que lê uma linha de caracteres e informa o número de caracteres na linha, exceto o *newline*, que indica o fim da linha. Usaremos um acumulador para contar os caracteres até o momento.

```
conta_linha(N) :- conta_aux(0, N).

conta_aux(A, N) :- get_char('\n'), !, A = N.
conta_aux(A, N) :- A1 is A + 1, conta_aux(A1, N).
```

O predicado `conta_aux(A, N)` é satisfeito quando o número de caracteres a serem lidos até o `\n` somado a `A` dá `N`. Observe que não é correto colocar um `get_char` na segunda cláusula de `conta_aux`, pois um caractere foi consumido na chamada da primeira cláusula independentemente de ter sido `\n` ou não.

Para escrever caracteres, há o predicado pré-definido `put_char(X)`, onde `X` deve ser um caractere, ou um átomo cujo nome tem apenas um caractere. Se `X` estiver não instanciada ou for outro tipo de termo que não os descritos acima, ocorre erro.

6.3 Ler e escrever arquivos

Os predicados de leitura que vimos até agora utilizam sempre o que chamamos de *dispositivo corrente de entrada* em Prolog. Há um predicado `current_input(X)`

que instancia `X` ao dispositivo associado no momento à entrada de dados. Normalmente, este dispositivo é o teclado, que é indicado em Prolog pelo átomo especial `user_input`.

De modo semelhante, a saída de dados é feita sempre através do *dispositivo corrente de saída* em Prolog. Há um predicado `current_output(X)` que instancia `X` ao dispositivo associado à saída de dados naquele momento. Normalmente, este dispositivo é a tela do computador, que em Prolog indica-se pelo átomo especial `user_output`.

É possível trocar os dispositivos correntes de entrada e saída para arquivos, por exemplo. Para tanto, é necessário inicialmente abrir um arquivo através do predicado pré-definido `open`. Por exemplo, a pergunta

```
?- open('arq.txt', read, X).
```

vai instanciar a variável `X` a um dispositivo de entrada que na verdade acessa o arquivo `arq.txt`. De forma semelhante, a pergunta

```
?- open('arq.saida', write, X).
```

vai associar `X` a um dispositivo de saída que direciona os dados para o arquivo `arq.saida`. Note que o segundo argumento de `open` determina que tipo de dispositivo está sendo criado.

Após abrir um arquivo, associando-o a um dispositivo (chamado de *stream* em Prolog), pode-se usá-lo para entrada ou saída de dados através dos predicados pré-definidos `set_input` e `set_output`, que recebem um dispositivo como argumento e tornam-no o dispositivo corrente de entrada ou saída, respectivamente. Assim, um programa em Prolog que pretenda usar um arquivo como entrada deve proceder da seguinte forma:

```
programa :-
        open('arq.txt', read, X),
        current_input(Stream),
        set_input(X),
        codigo_propriamente_dito,
```

```
set_input(Stream),
close(X).
```

Note que foi salvo o dispositivo anterior na variável `Stream`, para ser reestabelecido como dispositivo de entrada ao término do programa. Atitude semelhante deve ser usada em relação à saída.

6.4 Carregando um banco de dados

Ao escrevermos programas em Prolog, geralmente colocamos todas as cláusulas dos predicados que queremos definir num arquivo, que depois carregamos no sistema Prolog. Para carregar arquivos existe um predicado pré-definido em Prolog chamado `consult`. Quando `X` está instanciado ao nome de um arquivo, a meta `consult(X)` causa a leitura e armazenamento no banco de dados de Prolog das cláusulas contidas neste arquivo. Esta operação é tão comum que há uma abreviatura para ela: colocar vários nomes de arquivos numa lista e usá-la como pergunta para consultá-los todos:

```
?- [arq1, arq2, arq3].
```

O predicado `consult` remove as cláusulas dos predicados consultados no banco de dados antes de carregas as novas definições.

6.5 Operadores

Como dissemos anteriormente, operadores conferem maior legibilidade, permitindo que certos funtores sejam lidos e escritos de forma prefixa, infixa ou prefixa. Para tanto, é necessário também informar a precedência e a associatividade destes operadores. Apenas funtores de aridade um ou dois podem ser operadores.

Prolog oferece um predicado pré-definido `op(Prec, Espec, Nome)` para definir novos operadores. O argumento `Prec` indica a precedência, que é um inteiro entre 1 e 1200. Quanto mais alto este número, maior a precedência.

O argumento `Espec` serve para definir a aridade, a posição e a associatividade do operador. Os seguintes átomos podem ser usados no segundo argumento:

```
xfx xfy yfx yfy
fx fy
xf yf
```

Aqui `f` indica a posição do operador (funtor) e `x` e `y` as posições dos argumentos. Na primeira linha temos portanto especificações para operadores binários infixos. Na segunda linha, especificações para operadores unários prefixos e na última linha, para operadores unários posfixos.

As letras `x` e `y` dão informações sobre a associatividade. Numa expressão sem parênteses, `x` significa qualquer expressão contendo operadores de precedência estritamente menor que a de `f`, enquanto `y` significa qualquer expressão contendo operadores de precedência menor ou igual a de `f`. Assim, em particular `yfx` significa que o operador associa à esquerda, e `xfy` significa que o operador associa à direita. Se um operador é declarado com `xfx`, ele não associa.

Para exemplificar, eis a definição de alguns dos operadores em Prolog vistos até agora.

```
?- op(1200, xfx, ':-').
?- op(1200, fx, '?-').
?- op(1000, xfy, ',').
?- op(900, fy, '\+').
?- op(700, xfx, '=').
?- op(700, xfx, '<').
?- op(700, xfx, '>').
?- op(700, xfx, 'is').
?- op(500, yfx, '+').
?- op(500, yfx, '-').
?- op(400, yfx, '*').
?- op(400, yfx, '//').
?- op(400, yfx, '/').
?- op(400, yfx, 'mod').
?- op(200, fy, '-').
```

Exercícios

1. Escreva um predicado `estrelas(N)` que imprime `N` caracteres "`*`" no dispositivo de saída.

2. Escreva um predicado `guess(N)` que incita o usuário a adivinhar o número `N`. O predicado repetidamente lê um número, compara-o com `N`, e imprime "`Muito baixo!`", "`Acertou!`", "`Muito alto!`", conforme o caso, orientando o usuário na direção certa.

3. Escreva um predicado que lê uma linha e imprime a mesma linha trocando todos os caracteres `'a'` por `'b'`.

Capítulo 7

Predicados pré-definidos

Neste capítulo veremos alguns predicados pré-definidos importantes que não foram tratados até agora. Para organizar a exposição, vamos dividi-los em várias partes: verdadeiros, tipos, banco de dados, listas e conjuntos, e outros.

7.1 Verdadeiros

`true` satisfeito sempre, só uma vez.

`repeat` satisfeito sempre, inclusive em todas as ressatisfações.

7.2 Tipos

`var(X)` é satisfeito quando `X` é uma variável não instanciada.

`novar(X)` é satisfeito quando `X` é um termo ou uma variável instanciada. O contrário de `var(X)`.

`atom(X)` é satisfeito quando `X` é um átomo (constante não numérica).

`number(X)` é satisfeito quando `X` é um número.

`atomic(X)` é satisfeito quando `X` é um átomo ou um número.

7.3 Banco de dados

`listing` é satisfeito uma vez, e lista todas as cláusulas do banco de dados.

`listing(P)` é satisfeito uma vez, e lista todas as cláusulas do predicado `P`.

`asserta(X), assertz(X)` são satisfeitos uma vez, e adicionam a cláusula `X` ao banco de dados. O predicado `asserta` adiciona a cláusula nova **antes** das outras do mesmo predicado. O predicado `assertz` adiciona a cláusula nova **depois** das outras do mesmo predicado.

`retract(X)` é satisfeito uma vez, e remove a cláusula `X` do banco de dados.

7.4 Listas

`last(X, L)` é satisfeito quando `X` é o último elemento da lista `L`.

`reverse(L, M)` é satisfeito quando a lista `L` é a reversa da lista `M`.

`delete(X, L, M)` é satisfeito quando a lista `M` é obtida da lista `L` pela remoção de todas as ocorrências de `X` em `L`.

7.5 Conjuntos (listas sem repetições)

`subset(X, Y)` é satisfeito quando `X` é um subconjunto de `Y`, isto é, todos os elementos de `X` estão em `Y`.

`intersection(X, Y, Z)` é satisfeito quando a lista `Z` contém todos os elementos comuns a `X` e a `Y`, e apenas estes.

`union(X, Y, Z)` é satisfeito quando a lista `Z` contém todos os elementos que estão em `X` ou em `Y`, e apenas estes.

7.6 Outros

`X =.. L` é satisfeito se `X` é um termo e `L` é uma lista onde aparecem o funtor e os argumentos de `X` na ordem. Exemplos:

```
?- gosta(maria, pedro) =.. L.
L = [gosta, maria, pedro]
?- X =.. [a, b, c, d].
X = a(b, c, d)
```

`random(N)` em SWI Prolog é um operador que pode ser usado em uma expressão aritmética à direita de `is`, e produz um inteiro aleatório no intervalo 0 a `N-1`. Exemplo: `X is random(30000).`

`findall(X, M, L)` instancia `L` a uma lista contendo todos os objetos `X` para os quais a meta `M` é satisfeita. O argumento `M` é um termo que será usado como meta. A variável `X` deve aparecer em `M`. Exemplo: `findall(X, member(X,[a,b,c]), L)` fará com que `L` seja instanciada com a própria lista `[a,b,c]`.

`;` é um operador binário que significa "ou". É satisfeito quando uma das duas metas é satisfeita. Em geral, pode ser substituído por duas cláusulas. Por exemplo,

```
atomic(X) :- (atom(X) ; number(X)).
```

é equivalente a

```
atomic(X) :- atom(X).
atomic(X) :- number(X).
```

Capítulo 8

Estilo e depuração

A melhor maneira de evitar ou minimizar erros é programar com cuidado, seguindo as regras de estilo, consagradas ao longo de várias décadas pelos melhores programamdores. O provérbio "é melhor prevenir do que remediar" aplica-se muito bem neste cenário. Seguem-se algumas recomendações de estilo ao escrever programas em Prolog.

- Coloque em linhas consecutivas as cláusulas de um mesmo predicado, e separe cada predicado do próximo com uma ou mais linhas em branco.
- Se uma cláusula cabe toda dentro de uma linha (até cerca de 70 caracteres), deixe-a em uma linha. Caso contrário, coloque apenas a cabeça e o ":-" na primeira linha, e, nas linhas subseqüentes, coloque as submetas do corpo, indentadas com TAB e terminadas por vírgula, exceto a última, que é terminada com ponto final.
- Evite predicados com muitas regras. Se um predicado tem mais de cinco ou dez regras, considere a possibilidade de quebrá-lo em vários outros predicados.
- Evite o uso de ";", substituindo-o por mais de uma cláusula quando possível.
- Use variáveis anônimas para variáveis que ocorrem apenas uma vez numa cláusula.

Ao lado destas recomendações gerais, é bom estar atento aos seguintes cuidados ao definir um predicado:

- Verifique se seu nome está digitado corretamente em todas as ocorrências. Erros de digitação são comuns.
- Verifique o número de argumentos e certifique-se de que combina com o seu projeto para este predicado.
- Identifique todos os operadores usados e determine sua precedência, associatividade e argumentos para verificar se estão de acordo com o planejado. Em caso de dúvidas, use parênteses.
- Observe o escopo de cada variável. Observe quais variáveis vão compartilhar o mesmo valor quando uma delas ficar instanciada. Observe se todas as variáveis da cabeça de uma regra aparecem no corpo da regra.
- Tente determinar quais argumentos devem vir instanciados ou não instanciados em cada cláusula.
- Identifique as cláusulas que representam condições de parada. Verifique se todas as condições de parada estão contempladas.

Fique atento também para certos erros comuns que costumam assolar os programas feitos às pressas:

- Não esqueça o ponto final ao temino de cada cláusula. No final do arquivo, certifique-se de que haja um *newline* após o último ponto final.
- Verifique o casamento dos parênteses e colchetes. Um bom editor (por exemplo, Emacs) pode ajudar nesta tarefa.
- Cuidado com erros tipográficos nos nomes de predicados pré-definidos. Tenha sempre à mão (ou à Web) um manual completo da implementação de Prolog que você está usando. Consulte-o para certificar-se do uso correto dos predicados pré-definidos em seu programa.

8.1 Depuração

Programas feitos com cuidado tendem a apresentar menos erros, mas é difícil garantir que não haverá erros. Caso seu programa não esteja funcionando, ou seja, provocando erros de execução, dando a resposta errada ou simplesmente respondendo "`no`", você pode usar os potentes recursos de depuração de Prolog para localizar e corrigir os erros. Há predicados pré-definidos para ajudá-lo nesta tarefa. Nesta seção descreveremos os predicados de depuração existentes na implementação SWI Prolog. Outras implementações podem ter predicados ligeiramente diferentes para depuração.

O predicado `trace`, sem argumentos, liga o mecanismo de acompanhamento de metas. Cada meta é impressa nos seguintes eventos:

CALL quando ocorre uma tentativa de satisfação da meta

EXIT quando a meta é satisfeita

REDO quando ocorre uma tentativa de ressatisfação da meta

FAIL quando a meta falha

Para cancelar o efeito de `trace`, há o predicado `notrace`. O predicado `spy(P)` "espia" `P`, ou seja, faz com que os eventos relacionados às metas do predicado `P` sejam acompanhados. Para cancelar este efeito, use `nospy(P)`. O predicado `debugging`, sem arugmentos, indica o status da depuração e lista todos os predicados que estão sob espionagem.

Quando o acompanhamento de metas está ligado, Prolog pára a execução em cada evento relevante e nos mostra o evento e a meta que está sendo examinada. Neste ponto temos controle sobre como continuar o acompanhamento, com várias opções. As opções são escolhidas por teclas, geralmente a primeira letra de um verbo em inglês que nos lembra a ação a realizar. A Tabela 8.1 contém algumas das opções disponíveis.

Opção	Verbo	Descrição
w	write	imprime a meta
c	creep	segue para o próximo evento
s	skip	salta até o próximo evento desta meta
l	leap	salta até o próximo evento acompanhado
r	retry	volta à primeira satisfação da meta
f	fail	causa a falha da meta
b	break	inicia uma sessão recursiva do interpretador
a	abort	interrompe a depuração

Tabela 8.1: Possíveis ações numa parada de depuração.

Capítulo 9

Gramáticas

Um dos principais usos de Prolog é em processamento de linguagens naturais. Seu emprego é tão intenso nesta área que Prolog oferece mecanismos específicos para tratar este problema. Nesta seção conheceremos alguns destes mecanismos.

9.1 O problema da análise léxica

Para enteder uma frase numa língua qualquer, o primeiro passo geralmente consiste em verificar se a frase segue as regras gramaticais da língua. Computacionalmente, uma das maneiras mais simples de fazer esta verificação é usando gramáticas "livres de contexto", como a que especificamos abaixo:

```
sentenca --> sujeito, predicado.

sujeito --> artigo, substantivo.

predicado --> verbo.
predicado --> verbo, objeto.

artigo --> [o].
artigo --> [a].

substantivo --> [pera].
substantivo --> [homem].
```

```
verbo --> [come].
verbo --> [canta].

objeto --> artigo, substantivo.
```

A gramática consiste em um certo número de regras, que dizem como os diversos ítens da frase podem ser decompostos. Por exemplo, as regras acima definem que uma sentença válida deve ser composta de um sujeito seguido de um predicado. Por sua vez, um sujeito é formado por um artigo seguido de um substantivo. E assim por diante.

Desta forma, para verificar se uma sentença é válida, devemos tentar "encaixá-la" na definição acima. Tome como exemplo a frase "o homem come a pera". Ela pode ser encaixada na gramática que demos da maneira indicada na Figura 9.1

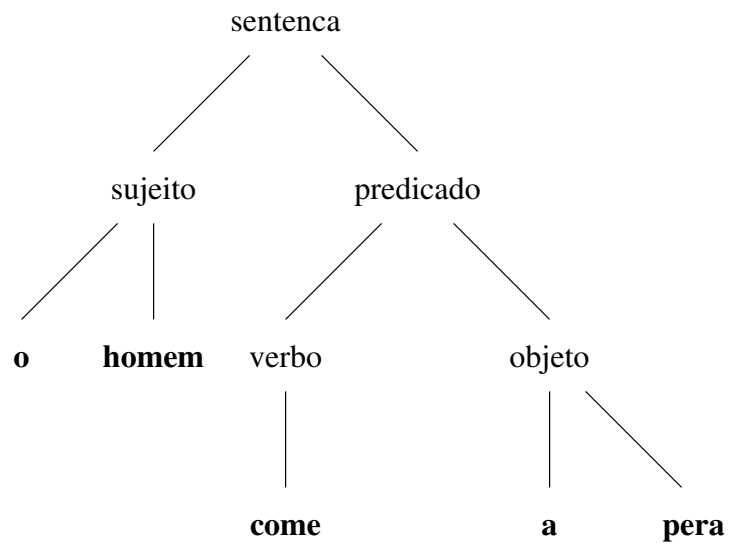

Figura 9.1: Análise léxica da sentença "o homem come a pera".

Ao definir os diversos itens da frase, há sempre um que é o principal, do qual emanam todos os demais. No nosso caso, este item é "sentenca", definido na primeira regra. E, ao definir cada item em função de outros, fatalmente acabamos chegando aos itens chamados terminais, para os quais a definição é dada em termos de exemplos concretos. Os itens verbo, artigo e susbtantivo são terminais na

gramática dada anteriormente. Observe que indicamos isto usando uma notação de lista para os exemplos.

Pois bem, o problema da análise léxica e dada uma frase qualquer, determinar se ela se encaixa numa descrição de gramática cmo a que demos nesta seção, e construir uma árvore léxica para ela, nos moldes da Figura 9.1. Um programa que faça isto é chamado de analisador léxico.

A formalização de regras sintáticas que ilustramos nesta seção é chamada de DCG, sigla do inglês *Definite Clause Grammar*. Muitas implementações Prolog fornecem mecanismos para lidar com estas gramáticas e executar análise léxica, complementada por várias funcionalidades extras, como veremos no restante deste capítulo.

9.2 Análise léxica em Prolog

Ao realizar análise léxica em Prolog, é usual representar palavras por átomos e frases por listas de átomos. Assim, a frase "o homem come a pera" seria representada pela lista

```
[o, homem, come, a, pera]
```

Isto posto, é natural implementar a análise léxica como um predicado que será satisfeito quando a frase dada (em forma de lista) segue as regras gramaticais indicadas, e falha caso contrário. Assim, teremos um predicado `sentenca(X)` que é satisfeito quando `X` for uma lista de palavras que formam uma sentença válida.

Como testar isto? Bem, segundo as regras, uma sentença válida deve ser formada por um sujeito seguido de um predicado. Observe que a "predicado" está agora sendo umada em dois sentidos diferentes. Um deles são os predicados em Prolog. O outro é o predicado que é um item, ou seja, uma parte, de uma frase válida. Cuidado para não confundir estes dois usos!

Continuando, seria interessante ter predicados (de Prolog) para validar os outros itens da frase, por exemplo, um predicado `sujeito(X)` que é satisfeito quando

`X` representa um sujeito válido, e `predicado(X)` que é satisfeito quando `X` representa um predicado válido. Com isto, poderíamos escrever o predicado `sentenca` em função de outros, como segue:

```
sentenca(X) :-
      append(Y, Z, X), sujeito(Y), predicado(Z).
```

Note o uso de `append`, que quebra a frase inicial de todas as formas possíveis e testa, para cama uma delas, se os pedaços resultantes podem ser vistos como sujeitos e predicados. Para que o predicado `sentenca` possa funcionar, é preciso definir todos os outros dos quais depende. A definição final ficaria algo como:

```
sentenca(X) :-
      append(Y, Z, X), sujeito(Y), predicado(Z).

sujeito(X) :-
      append(Y, Z, X), artigo(Y), substantivo(Z).

predicado(X) :- verbo(X).
predicado(X) :- append(Y, Z, X), verbo(Y), objeto(Z).

artigo([o]).
artigo([a]).

substantivo([pera]).
substantivo([homem]).

verbo([come]).
verbo([canta]).

objeto(X) :-
      append(Y, Z, X), artigo(Y), substantivo(Z).
```

Observe que, para os itens terminais, as definições se transformam em fatos Prolog e não em regras Prolog. Note também que definições alternativas naturalmente se transformam em cláusulas alternativas em Prolog.

Agora o programa está completo, e será capaz de verificar frases de acordo com a nossa gramática. A pergunta

```
?- sentenca([o, homem, come, a, pera]).
yes
```

é respondida corretamente. Também frases gramaticalmente incorretas como

```
?- sentenca([homem, pera, come, a, o]).
no
```

são classificadas como tais.

Apesar de correto, o esquema acima não é o mais eficiente possível, pois `append` experimenta todas as combinações de quebra da frase, sendo que em geral apenas uma ou duas delas serão corretas. O mesmo acontece nos pedaços de frases passados aos predicados de todos os intens tornando a busca ineficiente.

Mas, como melhorar a performance? A resposta está nos acumuladores, que estudamos na Seção 4.5. Começamos por fazer com que o predicado de cada item tenha não um mas **dois** parâmetros, com o seguinte significado, exemplificado abaixo para o predicado `sujeito`:

> `sujeito(X, Y)` é satisfeito quando
> existe um sujeito no início da lista `X`
> e o resto da lista após o sujeito é `Y`.

Para ajudar a entender o conceito, damos a seguir algumas perguntas que seriam respondidas afirmativamente com esta nova interpretação:

```
?- sujeito([o, gajo, come, a pera], [come, a, pera]).
?- sujeito([a, pera, canta], [canta]).
?- sujeito([o, homem, come, a pera], X).
?- sujeito([a, pera, canta], X).
```

sendo que, nas duas últimas, a variável `X` fica instanciada com o que sobrar da lista após um sujeito.

Precisamos também modificar as definições dos predicados para acomodar o novo significado. Por incrível que pareça, as definições ficam mais simples, e evitam o uso de `append`. Tomando como exemplo o caso de `sujeito`, ficamos com:

```
sujeito(X, Y) :- artigo(X, Z), substantivo(Z, Y).
```

Ou seja, existe um sujeito no início de `X` deixando uma cauda igual a `Y` se existe um artigo no início de `X` deixando uma cauda `Z`, e no início desta cauda `Z` existe um substantivo, deixando uma cauda `Y`. A Figura 9.2 mostra o que seriam os pedaços `X`, `Y` e `Z` num caso prático.

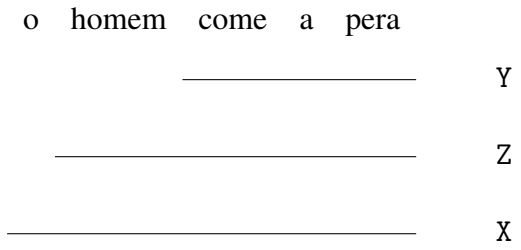

Figura 9.2: Valor dos parâmetros `X`, `Y` e `Z` numa análise léxica para `sujeito`.

Note, porém, que a nova definição de `sujeito` lembra muito a segunda cláusula da definição de `append`. É como se a tivessemos incorporado em cada item. Vejamos como fica o programa inteiro.

```
sentenca(S0, S) :- sujeito(S0, S1), predicado(S1, S).

sujeito(S0, S) :- artigo(S0, S1), substantivo(S1, S).

predicado(S0, S) :- verbo(S0, S).
predicado(S0, S) :- verbo(S0, S1), objeto(S1, S).

artigo([o|S], S).
artigo([a|S], S).

substantivo([pera|S], S).
substantivo([homem|S], S).

verbo([come|S], S).
verbo([canta|S], S).

objeto(S0, S) :- artigo(S0, S1), substantivo(S1, S).
```

Observe como a nova definição se reflete nos itens terminais. Eles também têm dois parâmetros, sendo o primeiro a entrada e o segundo, o resto após o terminal.

No caso dos terminais, onde temos átomos concretos para o item, o segundo parâmetro é simplesmente a cauda do primeiro, após consumido o átomo em questão.

Estas novas definições tornam o programa mais eficiente, mantendo-o correto. Porém, ainda dá pra melhorar a forma, simplificando o trabalho do programador, como veremos na próxima seção.

9.3 Notação para regras gramaticais

SWI-Prolog, assim como várias outras implementações, oferece uma maneira mais conveniente de escever gramáticas livres de contexto. Basicamente, tudo o que temos que fazer é descrever a gramática exatamente como foi feito no início deste capítulo, usando o funtor `-->` para indicar as regras gramaticais. Prolog sabe traduzir esta notação em cláusulas da maneira mais eficiente explicada na Seção 9.2, gerando um predicado de dois parâmteros para cada item.

Para testá-lo, pode-se usar perguntas como as seguintes:

```
?- sentenca([o, homem, come, a, pera], [ ]).
yes
?- sujeito([o, homem, canta], X).
X = [canta]
```

9.4 Argumentos adicionais

Além dos dois argumentos que a notação para regras gramaticais automaticamente cria, é possível acrescentar argumentos adicionais para propósitos específicos. Vamos trabalhar um exemplo onde esta funcionalidade pode ser útil. Trata-se da concordância em número entre sujeito e verbo. Sentenças como “os homens come a pera” ou então “o homem comem a pera” não são corretas, mas seriam aceitas pela extensão natural de nossa gramática inicial ao incluir o artigo “os”, o substantivo “homens” e o verbo “comem”, ou seja, as formas plurais de itens terminais já presentes na gramática.

Para conseguir a concordância em número entre o sujeito e o predicado, criaremos um argumento adicional para indicar se a frase está no singular ou no plural. Este argumento será passado a todos os itens, e servirá para garantir que sujeitos no singular combinem-se apenas com predicados no plural para constituir frases corretas. A gramática modificada ficará assim:

```
sentenca --> sentenca(X).

sentenca(X) --> sujeito(X), predicado(X).

sujeito(X) --> artigo(X), substantivo(X).

predicado(X) --> verbo(X).
predicado(X) --> verbo(X), objeto(_).

artigo(singular) --> [o].
artigo(singular) --> [a].
artigo(plural) --> [os].
artigo(plural) --> [as].

substantivo(singular) --> [pera].
substantivo(singular) --> [homem].
substantivo(plural) --> [peras].
substantivo(plural) --> [homens].

verbo(singular) --> [come].
verbo(singular) --> [canta].
verbo(plural) --> [comem].
verbo(plural) --> [cantam].

objeto(X) --> artigo(X), substantivo(X).
```

Note que as regras continuam essencialmente as mesmas, apenas com o argumento adicional indicando se a frase está no singular ou no plural. Nos itens terminais, foi necessário acrescentar fatos adicionais com as versões dos itens no prural. Agora nossas regras são capazes de verificar a concordância em número:

```
?- sentenca(X, [os, homens, comem, as, peras], [ ]).
X = plural

?- sentenca(X, [o, homens, come, as, pera], [ ]).
no
```

Observe ainda que, na segunda cláusula para predicado, a concordância se dá entre predicado e verbo, mas o objeto não precisa concordar com este verbo, visto que ele está ligado ao sujeito. Assim, frases como

```
?- sentenca(X, [o, homem, come, as, peras], [ ]).
X = singular
```

são também corretamente aceitas e classificadas.

Bibliografia

[1] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog: using the ISO standard*. Springer-Verlag, 5a. edition, 2003. ISBN 3-540-00678-8.