MC600 - Notas de Aula

João Meidanis

Parte I

LISP

Capítulo 1

Introdução

LISP vem de "list processing".

John McCarthy criou LISP em 1960 [2].

LISP é uma linguagem tão antiga que na época em que foi criada os computadores SÓ USAVAM LETRAS MAIÚSCULAS. Quando vieram as letras minúsculas, foi adotada a convenção de que em LISP uma letra minúscula ou maiúscula é a mesma coisa. Portanto, as palavras SEN, sen e SeN em LISP são todas equivalentes.

O padrão Common Lisp, consolidado em 1990, veio unir os vários dialetos de LISP existentes na época para formar uma linguagem bastante bem especificada. Usaremos o livro *Common Lisp: The Language*, 2a. edição, de Guy Steele, onde este padrão é detalhado com grande cuidado, como referência principal sobre LISP neste curso [4]. Site na internet (um de vários): `http://www.supelec.fr/docs/cltl/cltl2.html`

Algumas aplicações famosas escritas em LISP seguem.

1.1 Calculando derivadas simbólicas

Início da computação simbólica: programa que achava a derivada de uma função, mas usando símbolos (x, y, etc.)

Exemplo:

$$\frac{d}{dx}(x^2 + 3x) = 2x + 3,$$
$$\frac{d}{dx}(\log \sin x) = \frac{\cos x}{\sin x}.$$

Para denotar as funções neste programa é usada uma notação pré-fixa, típica de LISP. A função $x^2 + 3x$, por exemplo, é denotada por:

```
(+ (* X X) (* 3 X)).
```

Se for definida uma função chamada `DERIV` para calcular a derivada, pode-se pedir:

```
(deriv '(+ (* X X) (* 3 X)))
```

(note o apóstrofe antes do argumento) e o interpretador responderá:

```
(+ (* 2 x) 3)
```

ou seja, $2x + 3$.

1.2 O psiquiatra

ELIZA, que pretendia simular um psiquatra, foi talvez o primeiro programa de computador a passar o Teste de Turing. Foi escrito por Joseph Weizenbaum em 1966 [5].

Para interagir com o programa é preciso usar a língua inglesa. Embora o autor em seu artigo original tenha dito que o programa pode ser portado para outras línguas (inclusive mencionando que já naquele momento existia uma versão em alemão), as versões mais difundidas são em inglês.

O programa utiliza uma estratégia de psiquiatra Rogeriano, na qual o paciente é sempre estimulado a falar, elaborando sobre o que já disse. Eis um fragmento de conversação:

> Men are all alike.
>
> IN WHAT WAY?
>
> They're always bugging us about something or other.
>
> CAN YOU THINK OF A SPECIFIC EXAMPLE?
>
> Well, my boyfriend made me come here.
>
> YOUR BOYFRIEND MADE YOU COME HERE

E por aí vai. As frases em maiúsculas são as respostas do programa.

Existe uma versão dentro do Emacs: `M-x doctor`.

Este programa causou tanta sensação, e ainda causa! Recentemente, alguém acoplou-o a um AOL Instant Messenger e deixou que pessoas ao acaso se plugassem para falar com ele. Os resultados foram surpreendentes! Confira na Internet.

1.3 MYCIN

MYCIN, um sistema pioneiro para diagnósticos médicos, foi um dos precursores dos sistemas especialistas [3].

MYCIN representa seu conhecimento sobre os sintomas e possíveis diagnósticos como um conjunto de regras da seguinte forma:

> SE a infecção é bacteremia primária
> E o local da cultura é um dos locais estéreis

E suspeita-se que a porta de entrada é o trato gastro-intestinal
ENTÃO há evidência sugerindo (0.7) que a infecção é bacteróide.

As regras são na verdade escritas como expressões LISP. O valor 0.7 é uma estimativa de que a conclusão seja correta dadas as evidências. Se as evidências são também incertas, as suas estimativas serão combinadas com as desta regra para chegar à estimativa de correção da conclusão.

1.4 Interpretador

A maneira padrão de interação com uma implementação de Common Lisp é através de um laço ler-avaliar-imprimir (*read-eval-print loop*): o sistema repetidamente lê uma expressão a partir de uma fonte de entrada de dados (geralmente o teclado ou um arquivo), avalia a expressão, e imprime o(s) valor(es) em um destino de saída (geralmente a tela ou um arquivo).

Expressões são também chamadas de formas, especialmente quando são destinadas à avaliação. Uma expressão pode ser simplesmente um símbolo, e neste caso o seu valor é o valor como dado do símbolo.

Quando uma lista é availada (exceto nos casos de macros e formas especiais), supõe-se que seja uma chamada de função. O primeiro elemento da lista é tomado como sendo o nome da função. Todos os outros elementos da lista são tratados como expressões a serem avaliadas também; um valor é obtido de cada uma, e estes valores se tornam os argumentos da função. A função é então aplicada aos argumentos, resultando em um ou mais valores (exceto nos casos de retornos não locais). Se e quando a função retornar, os valores retornados tornam-se os valores da lista avaliada.

Por exemplo, considere a avaliação da expressão (+ 3 (* 4 5)). O símbolo + denota a função de adição, que não é uma macro nem uma forma especial. Portanto as duas expressões 3 e (* 4 5) são avaliadas para produzir argumentos. A expressão 3 resulta em 3, e a expressão (* 4 5) é uma chamada de função (a função de multiplicação). Portanto as formas 4 e 5 são avaliadas, produzindo os argumentos 4 e 5 para a multiplicação. A função de multiplicação calcula o número 20 e retorna-o. Os valores 3 e 20 são então dados como argumentos à função de adição, que calcula e retorna o número 23. Portanto indicamos `(+ 3 (* 4 5)) => 23`.

Usaremos CMUCL, a implementação do Common Lisp feita pela Carnegie Mellon University, EUA.

Onde pegar: `http://www.cons.org/cmucl/credits.html`. Como instalar: faça *download* do site e siga as instruções. Nota: apenas para Unix/Linux. Se você vai usar Windows, há vários outros pacotes na internet. Mas cuidado: antes de entregar qualquer projeto da disciplina, teste seu código no CMUCL! Em geral, as modificações serão poucas ou menhuma, especialemnte se você usar construções portáteis. Só serão aceitos projetos que rodem em CMUCL.

Como usar (básico):

> entrar: `lisp`
> sair: `(quit)`
> usar com arquivo: escreva um arquivo de texto simples com as expressões a avaliar, dê a ele extensão `.lsp`, e chame `lisp < arquivo.lsp`
> usar sem arquivos: simplesmente tecle as expressões (ou formas) seguidas de `ENTER`
> repetir última expressão: `*`
> repetir penúltima expressão: `**`
> repetir antepenúltima expressão: `***`

Uso dentro do Emacs (básico):

> preparar: a variável Emacs de nome `inferior-lisp-program` deve ter valor igual ao comando usado para chamar o interpretador. Se você estiver usando o CMUCL, este valor é `"lisp"`
> iniciar: `M-x run-lisp` cria um buffer Emacs onde você pode interagir com o interpretador. É como se fosse uma shell interna ao Emacs.
> sair: `(quit)`
> usar com arquivo: se você estiver editando um arquivo `.lsp`, o Emacs usará `lisp-mode` para você editá-lo, o que fornece indentação e balanceamento de parênteses, por exemplo

avaliação imediata: se você estiver editando um arquivo `.lsp`, teclando `C-x C-e` o Emacs avalia a expressão anterior ao cursor no buffer do lisp
outras facilidades: peça `M-x describe-bindings`

Uma grande vantagem de usar o editor Emacs é que ele balanceia os parênteses para você. E cuida também da indentação.

Compilação vs. interpretação: além de interpretador, o Common Lisp prevê que seja compiladas funções ou arquivos LISP, para rodarem mais rápido. Para compilar uma função `deriv`, por exemplo, use: `(compile 'deriv)`. Para compilar um arquivo `deriv.lsp`, por exemplo, use: `(compile-file "deriv.lsp")`.

Erros: tecle `0` e volte ao nível inicial.

Depuração: para depurar seus programas, é importante saber que existe um mecanismo de trace: `(trace deriv)`. A partir disso cada chamada a `deriv` imprime o valor dos argumentos passados à função, e cada retorno imprime o valor retornado. Para cancelar o trace, use `(untrace deriv)`. Você pode tracear várias funções ao mesmo tempo: `(trace fft gcd string-upcase)`.

Exercícios

1. Procure na internet uma versão do programa que calcula derivadas em LISP e use-o para calcular as derivadas das funções dadas como exemplo acima.

2. Converse um pouco com o psiquiatra do Emacs. Traga seus diálogos para a classe.

3. Procure na Internet a experiência feita com o psiquiatra e usuários ao acaso do AOL. Traga alguns diálogos interessantes para a classe.

4. Carregue o interpretador e peça para avaliar: `(+ 1 (* 3 4))`. Avalie diretamente e também a partir de um arquivo dentro do Emacs.

Capítulo 2

Elementos da linguagem

Tipos em LISP: só para dados, não para variáveis. Lista completa de tipos: array, atom, base-character, bignum, bit, bit-vector, character, compiled-function, complex, cons, double-float, extended-character, fixnum, float, function, hash-table, integer, keyword, list, long-float, nil, null, number, package, pathname, random-state, ratio, rational, readtable, sequence, short-float, signed-byte, simple-array, simple-bit-vector, simple-string, simple-vector, single-float, standard-char, stream, string, symbol, t, unsigned-byte, vector.

Além destes, podem ser criados outros pelo usuário (classes, etc.).

Átomos: números, símbolos, ou strings.

Números: inteiros, razões, ponto flutuante, complexos.

Inteiros: fixnum e bignum.

Razões: 2/3, -17/23, etc.

Ponto flutuante: como se fosse real em Pascal.

Complexos: #C(0 1), a unidade imaginária i, etc.

Símbolos: haverá uma secção só para eles adiante. Por enquanto, vamos considerá-los como seqüências de caracteres que podem inlcuir letras, números e sinais especiais, exceto brancos e parênteses, e que não possam ser confundidos com números. Além disso, não devem conter apenas pontos.

Exemplos válidos:

```
FROBBOZ
frobboz
fRObBoz
unwind-protect
+&
1+
pascal_style
b^2-4*a*c
file.rel.43
/usr/games/zork
```

Exemplos inválidos:

```
+1
..
```

Pares-com-ponto, ou conses: são estruturas, ou registros, com dois componentes: *car* e *cdr*.

Listas: uma lista é definida como sendo ou bem a lista vazia ou bem um cons cujo cdr é uma lista. O símbolo `NIL` é usado para denotar a lista vazia. A expressão `()` é sinônima de `NIL`.

Listas são denotadas escrevendo seus elementos na ordem, separados por espaços em branco (caracteres: espaço, tab ou newline) e cercados por parênteses.

Um cons cujo segundo elemento não é uma lista é denotado de forma semelhante, exceto que entre o último elemento e o fecha-parênteses é colocado um ponto "." (cercado de espaço em branco) e a seguir o cdr do último elemento. Exemplo: `(a . 4)` é um cons cujo car é um símbolo e cujo cdr é um número. Daí o nome par-com-ponto.

Representação gráfica — caixas. Há uma representação gráfica para conses. Nesta representação gráfica, colocamos uma caixa com dois compartimen-

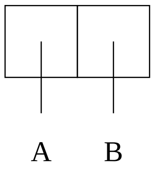

Figura 2.1: Representação gráfica de `(A . B)`.

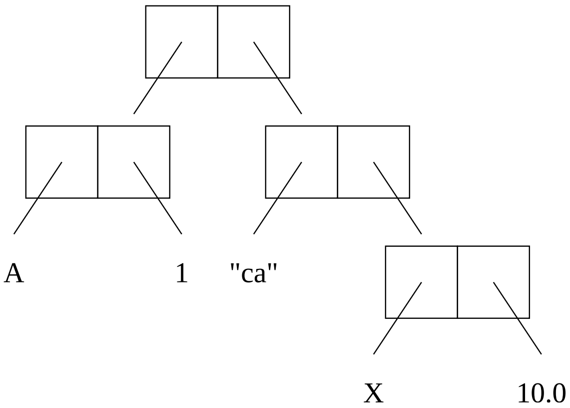

Figura 2.2: Representação gráfica de `( ( A . 1 ) . ( "ca". ( x . 10.0 ) ) )`.

tos, cada um deles com um apontador para uma componente do cons: o compartimento esquerdo aponta para o car e o direito para o cdr.

Exemplos: veja Figuras 2.1, 2.2, 2.3, 2.4.

Vamos aprender as primeiras funções LISP: CAR, CDR e CONS. São funções pré-definidas, existentes em qualquer ambiente LISP.

A função CAR retorna o primeiro componente de um cons. FIRST é um sinônimo de CAR. A função CDR retorna o segundo componente de um cons. REST é um sinônimo de CDR. Se CAR ou CDR forem aplicadas a (), o resultado é (). Dá erro se CAR ou CDR forem aplicadas a algo que não é um cons ou (). Exemplos:

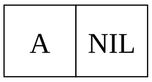

Figura 2.3: Representação gráfica de `(A . NIL)`, ou simplesmente `(A)`.

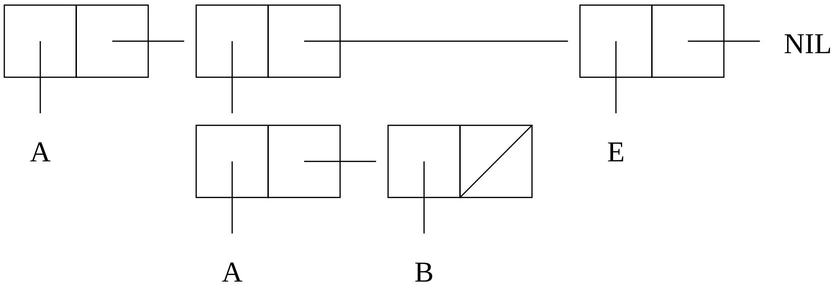

Figura 2.4: Representação gráfica de `( A ( A B ) E )`. Note as várias formas de expressar ponteiros a `NIL`.

```
(car '(A (A B) E)) => A
(car '((A . 1).  ("ca".  (x .  10.0)))) => (A . 1)
(cdr '(A (A B) E)) => ((A B) E)
(cdr '((A . 1).  ("ca".  (x .  10.0)))) => (("ca".  (x
.  10.0)))
```

Lembramos que a notação `=>` indica o resultado da avaliação de uma expressão LISP.

A função CONS recebe dois argumentos e retorna uma nova caixa contendo estes argumentos como componentes, na ordem dada. Não há sinônimos para CONS. Exemplos:

```
(cons 'a 'b) => (A . B)
(cons '1 '(a b)) => (1 A B)
```

A liguagem LISP foi a primeira a adotar um sistema de gerenciamento de memória conhecido como "coleção de lixo" (*garbage collection*). Neste sistema, o programador não precisa se preocupar em alocar ou desalocar memória explicitamente. Ao invés disso, o prórpio interpretador da linguagem se encarrega de alocar e desalocar memória quando necessário. A alocação de memória ocorre sempre que CONS é chamada, direta ou indiretamente. Quanto à desalocação, ela ocorre quando é necessário alocar mais memória mas acabou a memória do sistema. Neste momento o interpretador sai "catando lixo", que são caixas que não estão sendo mais apontadas por nenhuma estrutura do sistema, e que podem portanto ser liberadas. Por exemplo, na avaliação da expressão:

```
(car '(A (A B) E)) => A
```

o interpretador teve que construir 5 caixas para calcular o CAR da expressão. Depois deste cálculo as caixas ficaram "soltas" no sistema, e serão recolhidas na próxima coleta de lixo.

Exercícios

1. Quais dos seguintes símbolos são válidos: `1-`, `9.1`, `max-value`?
2. LISP sabe mexer com frações: avalie `(+ 2/3 4/5)`.
3. Não há limite para os inteiros em LISP: avalie 2^{1000} usando a expressão `(expt 2 1000)`.
4. Quais das seguintes expressões denotam listas: `()`, `( (A . B) C)`, ((A . (B . C)).
5. Desenhe a representação gráfica de: `((A B)(C D)(E F G) H)`, `(A 1 2)`, `(((A) 1) 2)`.
6. Calcule o CAR e o CDR das expressões dos dois últimos exercícios.
7. Para formar `(A B C)` a partir de átomos é necessário fazer: `(cons 'a (cons 'b (cons 'c ())))`. Monte expressões semlhantes para os conses dos dois dois últimos exercícios.

Capítulo 3

Estrutura da linguagem

Programa em LISP: define uma ou mais funções; por isto se diz "programação funcional".

LISP puro vs. funções com efeitos colaterais. Uma função tem efeitos colaterais quando modifica seus argumentos, ou modifica variáveis globais. Funções com efeitos colaterais são mais difícies de entender e manter, por isso devem ser evitadas.

Básico de DEFUN DEFUN é usado para definir funções em LISP. Mais tarde veremos a sintaxe completa. Por enquanto usaremos a sintaxe básica

`defun` *nome lista-de-args corpo*

Exemplos: calcular quadrado, calcular discriminante.

Básico de aritmética: `+, -, *, /`. Lembrar que a notação é prefixa.

Básico de condicionais (apenas IF e COND). IF sempre tem 3 argumentos, COND tem um número qualquer de argumentos; em cada um deles, a primeira forma é um teste. Símbolos especiais T e NIL: T é usado para condições "default" dentro de um COND. NIL é falso, qualquer outra coisa é verdadeiro. Lembrar que os argumentos são avaliados apenas se necessário, ou seja, IF é uma forma especial.

Exemplos: ver se discriminante é positivo, calcular máximo de dois ou três números.

Variáveis locais: LET. Exemplo: calcular raiz de equação do segundo grau, usando LET para o discriminante. Introduz SQRT.

Exercícios

1. Escreva uma função para a média de dois números.
2. Escreva uma função para testar se um número é positivo. Retornar T ou NIL.

Capítulo 4

Símbolos

Símbolos são objetos em LISP que aparecem em vários contextos: nomes de variáveis e de funções, por exemplo. Todo símbolo tem um nome (*print name*ou *pname*). Dado um símbolo, é possível obter seu nome como uma string, e vice-versa. Todo símbolo tem uma lista de propriedades (*property list* ou *plist*). Duas destas propriedades que nos interessam muito são o **valor como dado** e o **valor como função** de um símbolo.

Valor como dado e como função O valor como dado de um símbolo é usado para avaliar o símbolo numa posição não-funcional e o valor como função é usado para avaliar o símbolo na posição funional, que é a primeira posição de uma lista.

Símbolos não precisam ser explicitamente criados. A primeira vez que o iterpretador vê um símbolo, ele o cria automaticamente.

Como atribuir: DEFUN, SETF (impuro), LET, passagem de parâmetro (puro). DEFUN é a maneira de se definir novas funções. SETF deve ser usado apenas para variáveis globais ou vetores — evitar seu uso indiscriminado e em simples "tradução" de construções imperativas (de Pascal, C) para LISP. SETF também é muito útil para armazenar expresões grandes para testar suas funções junto ao interpretador.

QUOTE é uma forma especial que retorna seu argumento sem avaliação.

É utilizada para introduzir expressões constantes. É tão usada que inventaram uma abreviatura: apóstrofe precedendo uma expressão significa QUOTE aplicada a esta expressão. Exemplo:

```
(quote a) => A
'(cons 'a 'a) => (CONS (QUOTE A) (QUOTE A))
```

Existe também uma forma de especificar expressões quase constantes, isto é, expressões nas quais grande parte é constante exceto alguns pontos onde se deseja avaliação. O acento grave é usado no lugar do apóstrofe, e as sub-expressões que se deseja avaliar são precedidas de vírgula:

```
`(list (+ 1 2) ,(+ 2 3) ,(+ 3 4)) => (LIST (+ 1 2) 5 7)
```

Exercícios

1. Suponha que foram definidos:

   ```
   (defun xxx (x)
     (+ 1 x))

   (setf xxx 5)
   ```

   Qual o valor das seguintes expressões?

   (a) `(xxx 2)`
   (b) `(xxx (+ (xxx 5) 3))`
   (c) `(+ 4 xxx)`
   (d) `(xxx xxx)`

2. Qual o valor das expressões:

   (a) `(car '((a b c d)))`
   (b) `(cdr '((a b c d)))`
   (c) `(car (cdr (car (cdr '((((a b) (c d)) (e f)) (g h))))))`
   (d) `(cons (car '(a b f)) (cons (cons 'c '(x)) nil))`

Capítulo 5

Recursão

O que é recursão: função que se chama direta ou indiretamente. Exemplo: fatorial.

Como ela pode substituir laços de repetição. Fazer alguns loops simples em Pascal ou C e transformá-los em funções LISP. Exemplos: somar números de 1 a n, contar o número de positivos numa lista.

Recursão de rabo (*tail recursion*) — mais eficiente. É quando o resultado das chamadas recursivas é retornado sem modificação pela função. Os loops sempre resultam em funções com recursão de rabo. Exemplo:

Vantagens da recursão — código mais compacto, mais fácil de manter.

Existem construções LISP para loops, que serão tratadas mais adiante.

Exercícios

1. Exemplo de laço para transformar em recursão.

2. Combinação n tomados m a m.

Capítulo 6

Aritmética

As quatro operações básicas: $+$, $-$, $*$, $/$ em todas as possibilidades: 0, 1, 2, 3 ou mais argumentos.

A função `+` recebe um número qualquer de argumentos e retorna a soma de todos eles. Se houver zero argumentos, retorna 0 que é o elemento neutro para a adição.

A função `-` subtrai do primeiro argumento todos os outros, exceto quando há só um argumento: daí ela retorna o oposto dele. É um erro chamá-la com zero argumentos.

A função `*` recebe um número qualquer de argumentos e retorna o produto de todos eles. Se houver zero argumentos, retorna 1 que é o elemento neutro para a multiplicação.

A função `/` divide o primeiro argumento sucessivamente por cada um dos outros, exceto quando há só um argumento: daí ela retorna o inverso dele. É um erro chamá-la com zero argumentos. Dá erro também se houver só um argumento e ele for zero, ou se houver mais de um argumento e algum dos divisores for zero. Esta função produz razões se os argumentos são inteiros mas o resultado não é inteiro.

```
(+) => 0
(+ 3) => 3
```

```
(+ 3 5) => 8
(+ 3 5 6) => 14

(-) => ERRO
(- 3) => -3
(- 3 5) => -2
(- 3 5 6) => -8

(*) => 1
(* 3) => 3
(* 3 5) => 15
(* 3 5 6) => 90

(/) => ERRO
(/ 3) => 1/3
(/ 3 5) => 3/5
(/ 3 5 6) => 1/10
```

Arredondamento e truncamento: `floor` arredonda para baixo; `ceiling` arredonda para cima; `truncate` arredonda em direção ao zero; `round` arredonda para o inteiro mais próximo, e escolhe o par se houver dois inteiros mais próximos.

Funções `1+` e `1-`: incremento e decremento.

Funções `gcd` e `lcm`: máximo divisor comum (*greatest common divisor*) e mínimo múltiplo comum (*least common multiple*). Aceitam um número qualquer de argumentos inteiros, positivos ou negativos, retornando sempre um número positivo ou nulo. Com zero argumentos, retornam os elementos neutros para as respectivas operações: 0 e 1.

Função `abs`: retorna o valor absoluto de seu argumento.

Funções exponenciais: `exp`, calcula e^x; `expt`, calcula x^y.

Funções logarítmicas: `log`, calcula $\log_y x$. O segundo argumento é opcional, e caso seja omitido a base default é $e = 2.7182817$, a base dos logaritmos neperianos ou naturais.

Funções trigonométricas: `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`. Há também a constante `pi`.

Raiz quadrada e quadrado: `sqrt`, `sqr`.

Números complexos; função `conjugate`, `realpart`, `imagart`. Todas as funções acima que faça sentido aplicar em complexos aceitam complexos e produzem as respostas corretas. Algumas delas não faz sentido aplicar em complexos: arredondar e truncar, por exemplo.

Exercícios

1. Abra um livro do ensino médio sobre frações e use LISP para resolver expressões complicadas envolvendo frações. Confira as respostas com as do livro.

2. Faça o mesmo para números complexos.

Capítulo 7

Definição de funções

Função vs. macro (DEFUN e DEFMACRO): macro apenas expande e não avalia nada.

Argumentos: avaliados e não avaliados. Os argumentos de uma função são sempre avaliados antes de serem passados para ela. Numa macro, não são, mas macro é só para expandir, é só uma conveniência de notação. O próprio DEFUN é uma macro. Formas especiais também não avaliar seus argumentos (exemplo: condicionais).

Argumentos opcionais: tem que ser sempre os últimos, e são introduzidos através da chave `&optional`.

Número variável de argumentos: a chave `&rest` junta todos os argumentos restantes numa lista e passa esta lista à função.

PROGN implícito no corpo de uma função.

Exercícios

1. Escreva uma função que aceite argumentos opcionais.
2. Escreva uma função que aceite um número qualquer de argumentos.

3. Escreva uma função que aceite argumentos através de chaves e não de posição.

Capítulo 8

Condicionais

Neste capítulo veremos certas expressões condicionais que servem para testar condições que selecionarão uma entre várias expressões a avaliar.

A forma especial IF recebe em geral três argumentos. Inicialmente, o primeiro argumento é avaliado. Caso seja verdadeiro (isto é, seu valor é diferente de NIL), o segundo argumento é avaliar e seu valor retornado pelo IF. Se a avaliação do primeiro argumento resultar em falso (isto é, NIL), então o terceiro argumento é avaliar e seu valor retornado pelo IF. Assim, o primeiro argumento funciona como uma condição que determina quem será escolhido como valor final: o segundo argumento ou o terceiro argumento. O terceiro argumento de IF pode ser omitido, caso em que é considerado igual a NIL.

IF é chamada de *forma especial* porque não necessariamente avalia todos os seus argumentos, ao contrário das funções.

Embora qualquer valor diferente de NIL seja considerado verdadeiro, existe o valor especial T que é geralmente usado quando uma função precisa retornar um valor verdadeiro específico (por exemplo, as funções de comparação `<`, `>`, etc.).

Os símbolos NIL e T são constantes em Common LISP. Seus valores são NIL e T, respectivamente, e não podem ser modificados.

A macro COND implementa uma espécie de "case", ou seja, uma seleção entre múltiplas alternativas. O formato geral de um COND é a seguinte:

`cond` $\{(teste\ \{forma\}^*)\}^*$

onde as estelas indicam repetição zero ou mais vezes. Assim, a macro COND tem um número qualquer de *cláusulas*, sendo cada uma delas uma lista de formas (expressões a avaliar). Uma cláusula consiste de um *teste* seguido de zero ou mais *conseqüentes*.

A avaliação de um COND processa as cláusulas da primeira à última. Para cada cláusula, o teste é avaliado. Se o resultado é NIL, o processamento passa para a próxima cláusula. Caso contrário, cada um dos conseqüentes desta cláusula é availado e o valor do último é retornado pelo COND. Se não houver conseqüentes, o valor do teste (que é necessariamente diferente de NIL) é retornado.

Se as cláusulas acabarem sem que nenhum teste seja verdadeiro, o valor retornado pelo COND é NIL. É comum colocar-se T como teste da última cláusula, para evitar que isto aconteça.

A macro OR recebe zero ou mais argumentos e fornece uma espécie de "ou" lógico. Porém, ao invés de retornar somente T ou NIL, a forma OR retorna algo que pode ser mais útil em determinadas situações.

Especificamente, OR avalia os argumentos da esquerda para a direita. Se algum deles der verdadeiro, OR retorna este valor e não availa os demais. Se nenhum der verdadeiro, OR retorna NIL. A forma especial OR retorna portanto o valor do primeiro argumento que for diferente de NIL, não avaliando os demais.

A macro AND funciona de maneira similar: ela avalia os argumentos um a um, pára e retorna NIL ao achar o primeiro cujo valor seja NIL, e retorna o valor do último se todos forem diferentes de NIL. Um caso especial é o de zero argumentos, quando retorna T.

Note que OR a AND foram definidos de tal forma que podem prefeitamente ser usados como funções booleanas. Além disto existe a função NOT, que retorna T se o argumento é NIl e retorna NIL caso contrário.

Predicados são funções LISP que retornam um valor booleano. Alguns dos predicados mais importantes: `null`, `atom`, `consp`, `listp`, `numberp`. Eles testam se seus argumentos são, respectivamente: nulo, átomo, cons (par-com-ponto), lista, e número.

Exercícios

1. Quais das seguintes funções têm comportamento igual quando aplicadas a um único argumento: `null`, `not`, `and`, `or`.

Capítulo 9

O método do quadradão

O método do quadradão é uma maneira de raciocinar que pode ser útil para escrever a definição de funções recursivas em LISP. A idéia do método é usar um argumento específico para determinar como o resultado da chamada recursiva deve ser trabalhado para obter o resultado a retornar. O método tem este nome devido à figura que desenhamos para tirar as conclusões, que pode ser vista na Figura 9.1.

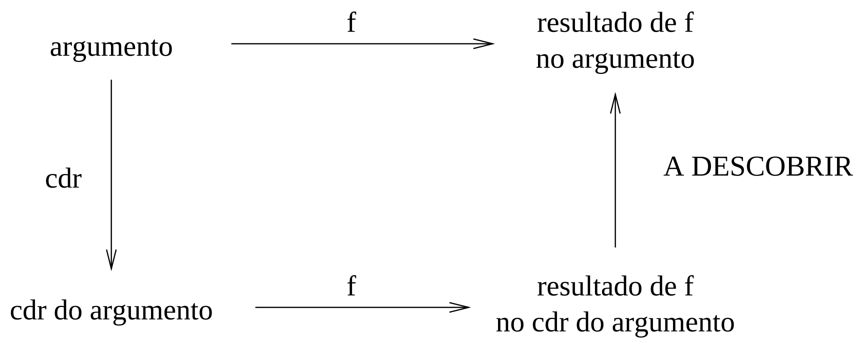

Figura 9.1: O método do quadradão.

Começamos escolhendo um argumento específico para o cálculo da função. Este argumento é colocado no canto superior esquerdo do quadrado. A partir dele, traçamos uma seta para a direita, que simboliza a aplicação da função

f a definir. No canto superior direito colocamos o resultado de f aplicada ao argumento.

A seguir, descendo a partir do argumento escolhido, escrevemos o `cdr` deste argumento no canto inferior do quadrado. Traçamos uma seta de cima para baixo simbolizando a aplicação da função `cdr`. Na parte inferior do quandrado, traçamos outra seta simbolizando a aplicação de f, levando ao valor de f aplicada ao `cdr` do argumento. Finalmente, olhamos para o quadrado e tentamos descobrir como é que se obtém o valor no canto superior direito a partir do valor no canto inferior direito. É a seta que está indicada por `?` na figura.

Vamos a um exemplo. Suponha que desejemos escrever a função `count` que recebe um item e uma lista e conta o número de ocorrências deste item na lista. Acompanhe pela Figura 9.2. Em nossa definição usaremos o nome `my-count` para a função para que, caso seja fornecida a um interpretador LISP, a função `count` do sistema não seja alterada.

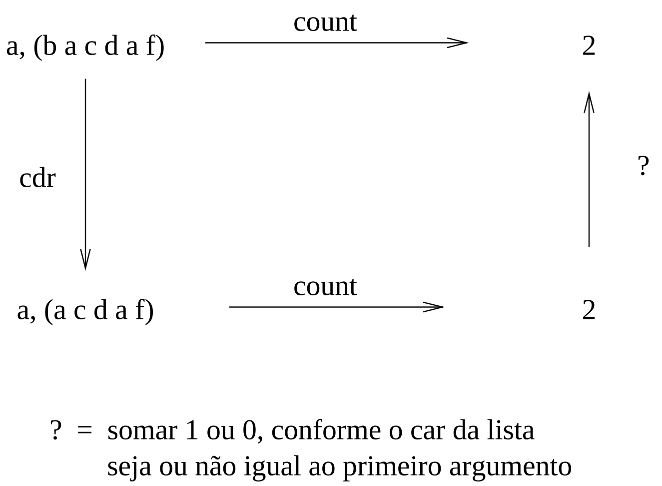

Figura 9.2: O método do quadradão aplicado à função `count`.

Escolhemos como argumentos o item `a` e a lista `(b a c d a f)`. O resultado de `count` deve ser 2, conforme ilustrado na figura. Aplicando o `cdr` ao argumento à lista dada, temos `(a c d a f)`. Mantemos o primeiro argumento inalterado, ou seja, continua sendo `a`. A aplicação de `count` na parte inferior da figura resulta em 2 também. Como fazemos para obter 2 a partir de 2? Neste caso, não é necessário fazer nada, porém, se o primeiro elemento da lista fosse igual a `a`, teríamos que somar uma unidade. Assim, temos a seguinte expressão para obter o resultado a partir da chamada recursiva:

```
    (+ (my-count item (cdr lista))
       (if (equal item (car lista)) 1 0)
```

Agora falta apenas acrescentar a condição de parada: se a lista for vazia, o contador dá zero. A definição completa fica assim:

```
(defun my-count (item lista)
  (if (null lista)
      0
    (+ (my-count item (cdr lista))
       (if (equal item (car lista)) 1 0)
       )
    )
  )
```

Fazer o exemplo da função `my-sort`. Utilize insertion sort. Neste caso, uma nova função terá que ser criada. Isto é normal em LISP. É melhor ter várias funções pequenas do que uma grande.

Exercícios

1. Escreva a função `last`, que recebe uma lista e retorna a última caixa desta lista, ou NIL se a lista for vazia.

2. Escreva a função `member`, que receb um item e uma lista e retorna o sufixo da lista a partir do ponto onde ocorre o item pela primeira vez nela, ou NIL caso o item não ocorra na lista.

Capítulo 10

Funções para listas

Algumas das funções que mencionamso abaixo vêm acompanahdas de uma definição. Nestes casos, precedemos o nome da função com `my-` para que, caso seja fornecida a um interpretador LISP, não altere a função original do sistema.

Existem as funções first, second, etc., até tenth: retorna o primeiro, segundo, etc. elemento da lista (há funções até para o décimo).

nth indice lista: retorna o n-ésimo elemento de uma lista. Os índices começam de zero.

```
(defun my-nth (indice lista)
  (if (= indice 0)
      (car lista)
    (nth (1- indice) (cdr lista))
    )
  )
```

elt lista indice: mesma coisa, só que a ordem dos argumentos é trocada.

last lista: retorna uma lista com o último elemento da lista dada. Se a lista dada for vazia, retorna NIL. Observe que esta função não retorna o último elemento, mas a última caixa. Porém, se for desejado o último elemento, basta aplicar CAR ao resultado desta função. Foi feito desta forma para poder distingüir uma lista vazia de uma lista tendo NIL como último elemento.

```
(defun my-last (lista)
  (if (null lista)
      ()
    (if (null (cdr lista))
        lista
      (my-last (cdr lista))
      )
    )
  )
```

caar, cdar, etc. (até 6 letras a e c no meio, entre c e r): retornam a composição de até 6 aplicações de CAR e CDR. Por exemplo: `(caddr x)` equivale a `(car (cdr (cdr x)))`.

length lista: retorna o comprimento (número de elementos no nível de topo) da lista.

```
(defun my-length (lista)
  (if (null lista)
      0
    (1+ (my-length (cdr lista)))
    )
  )
```

reverse lista: retorna uma lista com os elementos em ordem inversa relativa à ordem dada.

```
(defun my-reverse (lista)
  (my-revappend lista ())
  )

(defun my-revappend (lista1 lista2)
  (if (null lista1)
```

```
      lista2
    (my-revappend (cdr lista1) (cons (car lista1) lista2))
    )
  )
```

list &rest args: constrói e retorna uma lista com os argumentos dados. Aceita um número qualquer de argumentos. Com zero argumentos, retorna NIL.

```
(defun my-list (&rest args)
  args
  )
```

append &rest lists: retorna a concatenação de uma quantidade qualquer de listas.

```
(defun my-append2 (lista1 lista2)
  (if (null lista1)
      lista2
    (cons (car lista1) (my-append2 (cdr lista1) lista2))
    )
  )
```

subst novo velho arvore: substitui uma expressão por outra numa lista, em todos os níveis (por isto chamamos de árvore).

```
(defun my-subst (novo velho arvore)
  (cond ((equal velho arvore) novo)
        ((atom arvore) arvore)
        (t (cons (my-subst novo velho (car arvore))
                 (my-subst novo velho (cdr arvore))
                 )
           )
        )
  )
```

member item lista: se item pertence à lista, retorna a parte final da lista começando na primeira ocorrência de item. Se item não pertence à lista, retorna NIL. Pode ser usado como predicado para saber se um certo item pertence a uma lista.

```
(defun my-member (item lista)
  (cond ((null lista) nil)
        ((equal item (car lista)) lista)
        (t (my-member item (cdr lista)))
        )
  )
```

position item lista: retorna a primeira posição em que item aparece na lista. As posições são numeradas a partir de zero. Se o item não está na lista, retorna NIL.

```
(defun my-position (item lista)
  (cond ((null lista) nil)
        ((equal item (car lista)) 0)
        (t (let ((pos (my-position item (cdr lista))))
             (and pos (1+ pos))
             )
           )
        )
  )
```

count item lista: retorna o número de ocorrências do item na lista dada.

subseq lista comeco &optional final: retorna a subseqüência de uma lista, a partir de uma posição dada, e opcionalmente terminando numa outra posição dada. As posições são numeradas a partir de zero.

```
(defun my-subseq (lista comeco)
  (if (= comeco 0)
      lista
    (my-subseq (cdr lista) (1- comeco))
```

```
    )
  )
```

remove item lista: retorna uma nova lista obtida da lista dada pela remoção dos elementos iguais a item. A ordem relativa dos elementos restantes não é alterada.

```
(defun my-remove (item lista)
  (cond ((null lista) nil)
        ((equal item (car lista)) (my-remove item (cdr lista)))
        (t (cons (car lista) (my-remove item (cdr lista))))
        )
  )
```

mapcar funcao lista: retorna uma lista composta do resultado de aplicar a função dada a cada elemento da lista dada. Nota: funções com mais de um argumento podem ser usadas, desde que sejam fornecidas tantas listas quantos argumentos a função requer. Se as listas não forem todas do mesmo tamanho, o resultado terá o comprimento da menor.

```
(defun my-mapcar1 (funcao lista)
  (if (null lista)
      nil
    (cons (funcall funcao (car lista))
          (my-mapcar1 funcao (cdr lista))
          )
    )
  )
```

Exercícios

1. Escreva a função `append` para um número qualquer de arugmentos. Pode usar a função `append2` definida anteriormente.

Capítulo 11

Funções para conjuntos

Às vezes queremos usar listas para representar conjuntos de objetos, sem nos importarmos com a ordem deles na lista. LISP oferece suporte com várias funções que procuram imitar as operações mais comuns entre comjuntos: união, intersecção, etc.

Nota: conjuntos não têm elementos repetidos, enquanto que listas podem ter. Em cada operação, indicaremos o que ocorre quando há repetições nas listas dadas como argumentos.

As definições serão novamente com `my-` precedendo o nome da função, para permitir testes sem comprometer a função original definida no Common LISP.

union lista1 lista2: retorna uma lista contendo todos os elementos que estão em uma das listas dadas. Se cada uma das listas dadas não contém elementos repetidos, garante-se que o resultado não contém repetições. Contudo, se as listas de entrada tiverem elementos repetidos, o resultado pode ou não conter repetições.

```
(defun my-union (lista1 lista2)
  (cond ((null lista1) lista2)
        ((member (car lista1) lista2)
         (my-union (cdr lista1) lista2))
        (t (cons (car lista1) (my-union (cdr lista1) lista2)))
```

```
        )
  )
```

intersection lista1 lista2: retorna uma lista com os elementos comuns a lista1 e lista2. Se cada uma das listas dadas não contém elementos repetidos, garante-se que o resultado não contém repetições. Contudo, se as listas de entrada tiverem elementos repetidos, o resultado pode ou não conter repetições.

```
(defun my-intersection (lista1 lista2)
  (cond ((null lista1) nil)
        ((member (car lista1) lista2)
         (cons (car lista1) (my-intersection (cdr lista1) lista2)))
        (t (my-intersection (cdr lista1) lista2))
        )
  )
```

set-difference lista1 lista2: retorna uma lista com os elementos de lista1 que não estão em lista2.

```
(defun my-set-difference (lista1 lista2)
  (cond ((null lista1) nil)
        ((member (car lista1) lista2)
         (my-set-difference (cdr lista1) lista2))
        (t (cons (car lista1) (my-set-difference (cdr lista1) lista2)))
        )
  )
```

subsetp lista1 lista2: retorna verdadeiro quando cada elemento de lista1 aparece na lista2.

```
(defun my-subsetp (lista1 lista2)
  (cond ((null lista1) t)
        ((member (car lista1) lista2)
         (my-subsetp (cdr lista1) lista2))
        (t nil)
        )
  )
```

Todas estas funções admitem outras variações através de parâmetros adicionais ou maneiras diferentes de testar igualdade. Consulte a documentação oficial do Common Lisp para mais detalhes.

Exercícios

1.

Capítulo 12

Pacotes (módulos)

Objetivos do sistema de pacotes em Lisp: modularidade e divisâo do espaço de nomes.

Um pacote é um mapeamento de nomes para símbolos. A cada momento, apenas um pacote é usado para este mapeamento, o pacote corrente. A variável global `*package*` contém como valor o pacote corrente. Cada pacote têm seu nome e a lista de todos os pacotes existentes no sistema num certo momento pode ser obtida chamando-se a função pré-definida `list-all-packages`. Quando um argumento para uma função ou macro é um pacote, ele geralmente é dado como um símbolo cujo nome é o nome do pacote.

Um pacote tem símbolos internos e externos. Os símbolos externos de um pacote são a sua interface com o mundo exterior, por isso devem ter nomes bem escolhidos e anunciados a todos os usuários do sistema. Símbolos internos são para uso interno apenas, e não devem ser acessados por outros pacotes.

Definindo pacotes, entrando e saindo deles. A função `make-package` é usada para criar novos pacotes. Tipicamente recebe como argumento um símbolo, cujo nome será o nome do novo pacote. A função `delete-package` remove o pacote especificado do sistema. A macro `in-package` faz com que o seu argumento passe a ser o pacote corrente.

Acessando símbolos de outros pacotes: qualificação. Freqüentemente é ncecessário fazer referências a símbolos de outros pacotes que nâo o corrente. Isto é feito usando nomes *qualificados*, que são formados por um nome de pacote, dois pontos (":"), e o nome do símbolo. Por exemplo, `jogador:iniciar` refere-se a um símbolo de nome `iniciar` num pacote chamamdo `jogador`. Para isto, o símbolo deve ser um símbolo externo do referido pacote.

Deixando outros pacotes ver símbolos sem qualificação: exportação. A função `export` recebe um símbolo e o torna externo a um pacote.

Usando símbolos de outros pacotes sem qualificação: importação. Caso se deseje acessar um símbolo de outro pacote sem qualificação, deve-se importar o símbolo para o pacote corrente através da função `import`, que recebe uma lista de símbolos e os importa no pacote corrente. A partir daí, eles podem ser usados sem qualificação. Outra possibilidade é utilizar a função `use-package`, que internaliza no pacote corrente todos os símbolos externos do pacote usado como argumento. A partir daí, eles podem ser usados sem qualificação.

Alguns pacotes do Common Lisp: `common-lisp`, um pacote que contém as funções básicas do Common Lisp; `common-lisp-user`, o pacote default onde as definições do usuário são colocadas; `keywords`, pacote das palavras-chave (*keywords*), que são símbolos iniciados em dois pontos (":"), cujo valor como dado são eles mesmos e são constantes (não podem ter seu valor modificado).

Para colocar num novo pacote um lote de definições, basta preceder as definições pelas linhas de código abaixo:

```
(make-package 'pacote)
(in-package pacote)
(use-package 'common-lisp)
```

A seguir, pode-se colocar uma chamada de `export` para exportar símbolos que se deseja.

Exercícios

1. Crie um novo pacote, defina nele uma função e exporte esta função. Depois vá para o pacote `common-lisp-user` e chame esta função usando

a notação `pacote:funcao`.

Capítulo 13

Arrays e Loops

13.1 Arrays

Criando arrays: a função `make-array` retorna uma nova array. Seu parâmetro é uma lista de dimensões, por exemplo, `(make-array '(4 3 7))` retorna uma array tri-dimensional onde o primeiro índice vai de 0 a 3, o segundo vai de 0 a 2, e o terceiro vai de 0 a 6.

Acessando arrays: a função `aref` recebe uma array e índices e retorna o elemento da array na posição especificada pelos índices. Por exemplo, se a variável `mat` contiver a array criada no parágrafo anterior, a chamada `(aref mat 2 1 6)` retorna o elemento indexado por 2, 1 e 6 (equivalente a algo como `mat[2][1][6]` em C ou `mat[2,1,6]` em Pascal).

A macro `setf` pode ser usada com `aref` para modificar o elemento numa dada posição de uma array.

13.2 Loops

Há várias formas de fazer contruções iterativas em Lisp, mas aqui vamos apenas dar dois exemplos: as macros `dolist` e `dotimes`. O leitor interessado poderá encontrar outras construções mais complexas no manual de Common Lisp.

A forma `dolist` executa iteração sobre os elementos de uma lista. O formato geral é:

> `dolist` (*var listform* [*resultform*])
> $\{$*declaration*$\}*$ $\{$*tag* | *statement*$\}*$

Primeiramente, a forma *listform* é avaliada e deve produzir uma lista. Em seguida o corpo é executado uma vez para cada elemento da lista, com a variável *var* tendo como valor este elemento. Então a forma *resultform* é avaliada e seu valor é retornado pela macro `dolist`.

A forma `dotimes` executa iteração sobre uma seqüência de inteiros. O formato geral é:

> `dotimes` (*var countform* [*resultform*])
> $\{$*declaration*$\}*$ $\{$*tag* | *statement*$\}*$

Primeiramente, a forma *countform* é avaliada e deve produzir um inteiro. Em seguida o corpo é executado uma vez para cada inteiro de zero ao valor deste inteiro menos um, com a variável *var* tendo como valor este inteiro. Então a forma *resultform* é avaliada e seu valor é retornado pela macro `dotimes`.

Em ambos os casos, se *resultform* é omitida, o resultado é NIL.

Exercícios

1. Escreva uma expressão que imprime os elementos de uma lista usando `dolist`.
2. Escreva uma expressão que some os elementos de 0 a n usando `dotimes`.

Parte II

Prolog

Capítulo 14

Introdução

Prolog é ao mesmo tempo 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 MC600.

14.1 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 apneas coisas sobre as quais queremos raciocinar.

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

14.2 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.

14.3 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.

14.4 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.

14.5 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 apenas uma resposta 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(X, paulo).`	pergunta inicial
`X = maria ;`	primeira resposta
`no`	não há mais respostas

14.6 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 "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?". Itso 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 e 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. E 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*.

14.7 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". O exemplo acima seria escrito como:

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

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, chocolate).
```

ou, em outras palavras, "Pedro gosta de qualquer um que goste de vinho e de chocolate". 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 sarisfeita, 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 14.1, onde numeramos as variáveis com índices de acordo com 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 1.1, 1.2, e 1.3, 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

N	Meta	Marca	Variáveis
1	`irma_de(alice, eduardo)`	`irma_de`, regra 1	X_1 = `alice`, Y_1 = `eduardo`
1.1	`mulher(alice)`	`mulher`, fato 1	-
1.2	pais(alice, M_1, P_1)	`pais`, fato 2	M_1 = `vitoria`, P_1 = `alberto`
1.3	`pais(eduardo, vitoria, alberto)`	`pais`, fato 1	-

Tabela 14.1: Processamento da pergunta `?- irma_de(alice, eduardo).`

N	Meta	Marca	Variáveis
1	`irma_de(alice, X)`	`irma_de`, regra 1	X_1 = `alice`, Y_1 = X
1.1	`mulher(alice)`	`mulher`, fato 1	-
1.2	pais(alice, M_3, P_3)	`pais`, fato 2	M_1 = `vitoria`, P_1 = `alberto`
1.3	pais(Y_1, vitoria, alberto)	`pais`, fato 1	Y_1 = `eduardo`

Tabela 14.2: Processamento da meta `irma_de(alice, X)`.

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

O que ocorre então está na Tabela 14.2, onde `X` sem índice indicará a variável da pergunta.

Observe que X = Y_1 = `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 15

Termos

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*.

15.1 Constantes

As *constantes* pode ser *átomos* ou *números*. Note que em LISP os números são considerados átomos, mas em Prolog a nomenclatura é diferente, pois números não são átomos.

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
```

15.2 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.

15.3 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 uma 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(pedro, livro(incidente_em_antares, verissimo)).
```

O número de componentes de um funtor é a sua *aridade*. Funtores de aridade iguala zero são na verdade as constantes. Note que, diferentemente de LISP, 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 trs 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.

Capítulo 16

Igualdade e Unificação

Em Prolog, igualdade significa unificação. Existe um predicado infixo pré-definido `=` 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 $T1$ = $T2$, onde $T1$ e $T2$ são termos quaisquer, a decisão sobre sua igualdade é feita de acordo com a Tabela 16.1. 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 16.1.

Exercícios

		$T2$		
		constante	variável	estrutura
$T1$	constante	só se for a mesma	sempre; causa instanciação	nunca
	variável	sempre; causa instanciação	tornam-se ligadas	sempre; causa instanciação
	estrutura	nunca	sempre; causa instanciação	mesmo funtor, mesmo número de componentes e cada componente igual

Tabela 16.1: Condições para que dois termos unifiquem, segundo seus subtipos. O subtipo "variável" significa "variável não instanciada". O subtipo "estrutura" significa estruturas com aridade maior ou igual a um.

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

```
pilotos(A, londres) = pilotos(londres, paris)
ponto(X, Y, Z) = ponto(X1, Y1, Z1)
letra(C) = palavra(letra)
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 17

Aritmética

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

O spredicados 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(rhodi, 844, 878).
reinou(anarawd, 878, 916).
reinou(hywel_dda, 916, 950).
reinou(lago_ap_idwal, 950, 979).
reinou(hywal_ap_ieuaf, 979, 965).
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 = lago_ap_ieuaf ;
X = cadwallon ;
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 iunstanciada ao resultado da expressão. 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 também.

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 18

Estruturas de dados

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

18.1 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 18.1.

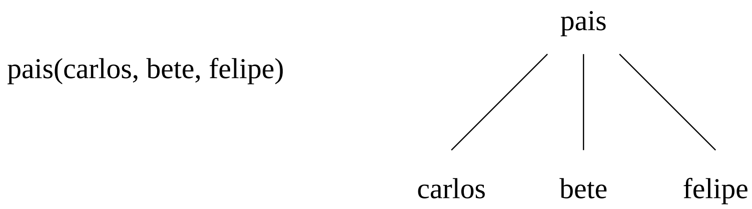

Figura 18.1: Estruturas como árvores.

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 18.2.

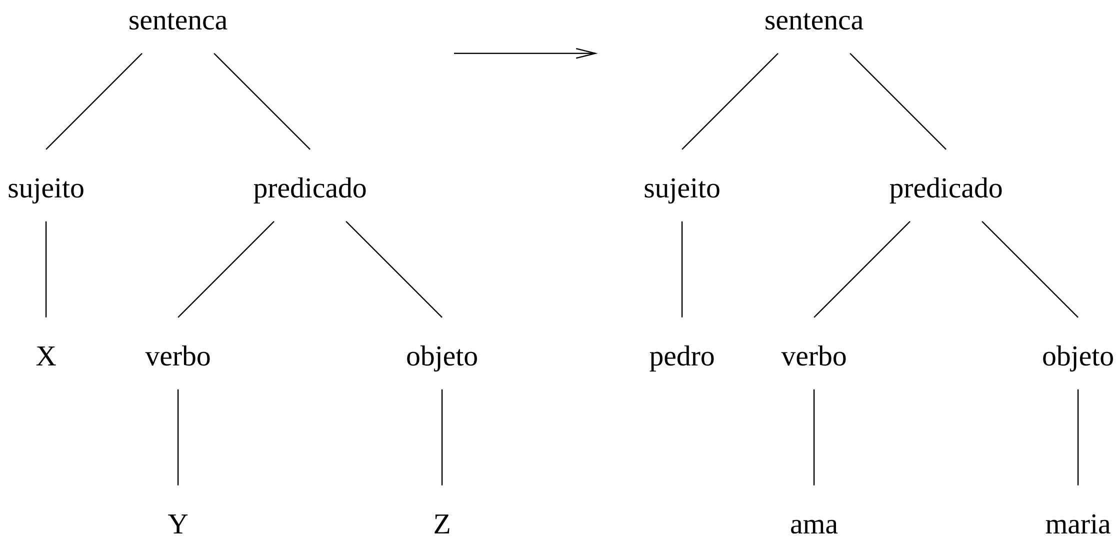

Figura 18.2: Sintaxe de sentenças.

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 18.3. A figura não é mais uma árvore, mas sim um grafo orientado acíclico.

18.2 Listas

Assim como em LISP, as listas são estruturas de dados importantes em Prolog. A notação para listas é diferente em cada linguagem, como veremos a seguir.

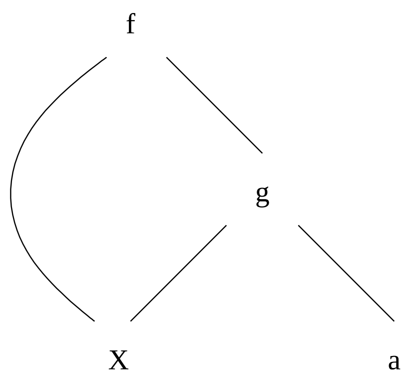

Figura 18.3: Diagrama mostrando ocorrências da mesma variável.

Em Prolog, uma lista é ou uma *lista vazia* ou uma estrutura com dois componentes: a *cabeça* e a *cauda*. A lista vazia é escrita como `[ ]`. O funtor usado para representar a estrutura de lista é o ponto "`.`", lembrando o par-com-ponto de LISP. Assim, uma lista que em LISP seria representada como `(a . (b . (c . ())))` em Prolog ficaria `.(a, .(b, .(c, [ ])))`, que está representada na Figura 18.4.

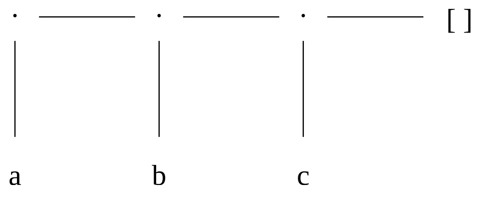

Figura 18.4: Uma lista em Prolog.

A exemplo de LISP, Prolog também tem uma maneira alternativa, mais prática, de denotar listas que evite usar 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 en cabeça e cauda (exceto a lista vazia). É exatamente como o car e o cdr em LISP. 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]
```

18.3 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.

18.4 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 de do itemrefi:append-cauda acima.

4. Conforme aplicamos a segunda cláusula, o primeiro argumento vai diminuindo, até ser a lista vazia, logo a recursão termina.

18.5 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]) :- 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 `listlen`, 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).

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

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

```
listlen([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 19

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) a busca continua do ponto marcado no banco de dados ao invés de começar do ínicio;
   (b) desfazem-se as instanciações e ligações causadas pela última unificação desta meta.

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.

19.1 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.

19.2 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.

19.3 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

19.3.1 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.

19.3.2 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.

19.3.3 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 77) 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.

19.4 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 uma situação em que o operador de igualdade "=" não pode ser 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 20

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.

20.1 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 queisermos 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 `write`, a meta `nl` só é satisfeita uma vez.

20.2 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 nãop 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.

20.3 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 pré-definido `current_output(X)` que instancia `X` ao dispositivo associado no momento à saída de dados. Normalmente, esta 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 (também chamado de *stream* em Prolog), pode-se usá-lo como entrada ou saída 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 restabelecido como dispositivo de entrada ao término do programa. Atitude semelhante deve ser usada em relação à saída.

20.4 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.

20.5 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 necessaio 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 `x` 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 21

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.

21.1 Verdadeiros

`true` satisfeito sempre, só uma vez.

`repeat` satisfeito sempre, inclusive todas as ressatisfações.

21.2 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.

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

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

21.3 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.

21.4 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`.

21.5 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` e em `Y`, e apenas estes.

21.6 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`.

`findall(X, M, L)` instancia `L` a uma lista contendo todos os objetos `X` para os quais a meta `M` é satisfeita. O conjunto `M` é um termo que será usado como meta. A variável `X` deve aparecer em `M`.

`;` 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 exmeplo,

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

é equivalente a

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

Capítulo 22

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 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.
- Evite o uso de "`;`" substituindo-o por mais de uma cláusula quando possível.
- Use variáveisd 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.

22.1 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 variar ligeiramente.

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 a meta é ressatisfeita

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 22.1 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 22.1: Possíveis ações numa parada de depuração.

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.

[2] John L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.

[3] Edward H. Shortliffe. *Computer-Based Medical Consultation: MYCIN*. Elsevier North Holland, 1976.

[4] Guy L. Steele. *Common Lisp the Language*. Digital Press, second edition, 1990. ISBN 1-55558-041-6.

[5] Joseph Weizenbaum. ELIZA - a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, January 1966.