Programação em Prolog

UMA ABORDAGEM PRÁTICA

Eloi L. Favero
Departamento de Informática
CCEN - UFPA

(Versão 2006)

favero@ufpa.br

- Parte I: Fundamentos e Técnicas de Programação
  - **Histórico e Teoria do Prolog**
  - **Técnicas de Programação:**
    - Listas e Estruturas de Dados
    - Fluxo de Controle e Aritmética
    - Animação de Programas
    - Classificação e Ordenação
- Parte II: A Linguagem Prolog
  - **O Prolog Padrão ISO**
  - **Ambiente de Depuração**
- Parte III: Estudos de Casos e Aplicações
  - **Programação de Gramáticas:**
    Autômatos e Gramáticas Regulares, Gramáticas Livres de Contexto, Gramáticas de Atributos; Análise Léxica, Sintática e Semântica
  - **Linguagens Formais e Compiladores**
  - **Processamento de Linguagem Natural:**
    Léxico/Morfologia, Sintaxe, Geração de Linguagem Natural
  - **Inteligência Artificial: KNN, Bayes, Kmeans**
  - **Ngramas: Bigramas e Trigramas**

Eloi L. Favero
Departamento de Informática
favero@ufpa.br

2006

Para minha companheira Flori

e nossos filhos, Emmanuel, Ayun e Thiago

pelo amor e a inspiração.

Prefácio

Nós acreditamos que programação pode ser, e deve ser, uma atividade intelectual recompensadora; e que uma boa linguagem de programação é uma poderosa ferramenta conceitual — uma ferramenta para organizar, expressar, experimentar com, e ainda comunicar nossos pensamentos; ... The Art of Prolog de L. Sterling e E. Shapiro [16].

Objetivos do livro

O objetivo deste livro-texto é servir como material didático na língua portuguesa para ensino de programação em Prolog, visando: 1) atender diferentes públicos, de cursos de graduação até cursos de pós-graduação, nas áreas da Ciência da Computação e correlatas (por exemplo, Sistemas de Informação); 2) servir de material de apoio para cursos com diferentes cargas horárias, de 20 horas até 60 horas; e 3) servir como material de referência para usuários da linguagem Prolog (adotamos o padrão definido pela norma ISO).

Dependendo da duração do curso, e do público-alvo, diferentes capítulos e seções do livro são usados. No final deste prefácio sugerimos algumas seqüências de capítulos para alguns usos do livro.

A linguagem de programação Prolog tem sido usada principalmente em disciplinas relacionadas com a área de Inteligência Artificial (IA). O Prolog[1], também, traz bons resultados quando é usado nas atividades de programação de disciplinas tais como: Programação em Lógica (a parte prática), Linguagens Formais, Compiladores, Linguagens de Programação (paradigma lógico), entre outras.

A abordagem do livro é essencialmente prática, em oposição ao enfoque teórico da Programação em Lógica que destaca assuntos, tais como estratégias de resolução e computabilidade de um programa. O objetivo de um curso prático de Prolog é ensinar as técnicas necessárias para se desenvolver, de forma clara, elegante e concisa, programas que expressam a solução de problemas reais.

Em uma perspectiva complementar, o subtítulo do livro **Uma abordagem prática** está relacionado ao ditado que diz: *o talento nasce da prática fiel*. A habilidade de desenvolver programas pensando em Prolog nasce de atividades práticas de programação. Para isso, apresentamos inúmeros exercícios associados a cada novo assunto apresentado. Por outro lado, em especial na terceira parte do livro, apresentamos estudos de casos baseados em problemas coletados em diversas

[1]Usamos dois termos como sinônimos: "A linguagem Prolog"e "O Prolog"como um sistema que executa a linguagem Prolog.

áreas da Ciência da Computação, incluindo: Linguagens Formais, Compiladores, Processamento de Linguagem Natural, Inteligência Artificial e Mineração de Dados.

O poder de expressividade do Prolog

Desde o surgimento, início dos anos 70, o Prolog mantém sua popularidade devido à combinação de diversas características que definem o seu poder expressivo como linguagem de programação:

- Prolog implementa um paradigma de **programação declarativa** (em contraste com o paradigma imperativo de C, Pascal) em que se descreve *o que fazer* e não *como fazer*. Em Prolog, o conhecimento computacional (fatos e regras de inferência) é expresso numa forma declarativa de alto nível;

- Prolog trabalha com estruturas de dados de alto nível. Estruturas de dados complexas, tais como listas, árvores, grafos, etc. são naturalmente representadas e manipuladas em Prolog. Assim sendo, Prolog possibilita **prototipação rápida**, por exemplo, para validação de especificações de sistemas de software;

- Prolog implementa um sistema de resolução para a lógica de cláusulas definidas, que é um subconjunto significativo da lógica de primeira ordem. Apesar desta fundamentação lógica, não é necessário ter formação prévia em lógica para **programar pensando em Prolog**;

- Prolog vem com um sistema de processamento de **regras gramaticais** chamado *Definite Clause Grammar (DCG)*, que é essencial para o desenvolvimento de aplicações (ou protótipos) nas áreas de Linguagens Formais, Compiladores e Processamento em Linguagem Natural (PLN) — o primeiro Prolog foi desenvolvido para PLN;

- Prolog permite **metaprogramação**, isto é, um programa pode se examinar e se automodificar. Metaprogramação é sinônimo de linguagens de alta ordem (não primeira ordem), as quais permitem e facilitam o desenvolvimento de ferramentas para outras linguagens, por exemplo, interpretadores para sistemas de regras para IA.

Vale citar, também, que o Prolog possui uma sintaxe simples, que se mantém estável desde o surgimento do Prolog padrão Edimburgo, em 1977. O padrão ISO Prolog (1996) [10], 19 anos depois, veio afirmar a sintaxe do Prolog Edimburgo como padrão, apenas apresentando um conjunto adicional de predicados predefinidos. Nesse sentido, observa-se que a maior parte dos programas encontrados em livros da década de 80 rodam sem alterações (ou quase sem) nos interpretadores atuais (por exemplo, os programas do livro *Programming in Prolog* de Clocksin & Mellish [5], de 1981). Neste texto, sempre que possível, adotamos o padrão ISO Prolog; eventuais diferenças são apontadas.

Disciplina de Programação

Como professor, tenho observado que a prática em Prolog desenvolve algumas habilidades positivas de programação, que, uma vez adquiridas como talentos, são levadas para outros ambientes de programação, tais como C++ e Java. Estas habilidades são:

- **Abstração:** código de especificação de alto nível;

- **Limpeza:** código-fonte limpo, conciso, sem redundâncias;
- **Ordem:** código bem estruturado e organizado;
- **Reusabilidade:** em especial, no reuso de predicados reversíveis;
- **Refinamento de programas e Refatoração de código:** rescrita de código quando já está funcionando visando a generalização, a limpeza, a simplicidade, a ordem, o reuso; o uso do interpretador facilita esta atividade (Just In Time Programming).

Programas Prolog são compactos e bem particionados. Em Prolog é mais fácil identificar possibilidades de fatorar código, removendo predicados redundantes. E, ao mesmo tempo, é fácil identificar possibilidades de reuso de predicados. Por exemplo, o predicado *append* (para concatenar listas) é reusável em vários outros algoritmos. Isto se deve, em parte, porque em Prolog um procedimento pode assumir mais de um papel: o *append*, definido inicialmente para concatenar duas listas, pode ser usado para dividir uma lista em duas, a partir de um dado elemento.

O Prolog é baseado num sistema de resolução associado a um processo de unificação. Enquanto um programa procedural é baseado em **dados+controle** (como fazer), um programa em Prolog é baseado em **fatos+regras** (o que fazer). O controle em um programa Prolog está implícito em regras condicionais de alto nível. Estruturas de dados com tipagem fraca, unificação e resolução fazem de um programa Prolog um texto mais próximo de uma especificação abstrata de um problema e, ao mesmo tempo, executável.

O *desenvolvimento por refinamentos sucessivos* é uma técnica de engenharia de software para programação. O Prolog, quando comparado com as linguagens imperativas, oferece algumas vantagens para trabalharmos com refinamento de programas: notação concisa e fundamentada em lógica; notação declarativa; procedimentos reversíveis; usa somente variáveis locais; etc. Estas características facilitam a escrita de duas versões equivalentes de um programa, sendo uma versão um refinamento da outra.

Estas habilidades de certa forma estão relacionadas ao poder expressivo da linguagem Prolog, que permite a codificação de programas altamente concisos, para tratar de problemas relativamente complexos. Por exemplo, um compilador é um programa relativamente complexo, até mesmo para minilinguagem. Em Prolog podemos implementar um compilador para uma minilinguagem em algumas páginas de código-fonte (como exemplificamos num dos capítulos do livro).

Usos para o Livro

O livro é constituído de 21 capítulos organizados em três partes. A parte I, capítulos 1 até 8, apresenta os fundamentos e técnicas da programação Prolog. A parte II, capítulos 9 e 10, apresenta um guia de referência rápida da sintaxe do Prolog ISO e o uso do depurador. A parte III explora alguns tópicos mostrando como programar soluções para problemas complexos: os capítulos 11 até 16 envolvem principalmente programação de gramáticas e os capítulos de 17 a 21 são tópicos de IA (aprendizagem de máquina e mineração de dados).

Pela escolha de uma determinada seqüência de capítulos, o livro-texto pode ser usado tanto em cursos rápidos de Prolog para estudantes sem experiência em programação, por exemplo, para estudantes de Lingüística interessados em PLN, como para cursos semestrais avançados para estudantes de pós-graduação em Ciência da Computação.

Seguem algumas sugestões de usos para o livro-texto:

- **Curso rápido de Prolog** (30 horas). São recomendados os primeiros quatro capítulos da Parte I, podem ser vistos só superficialmente os capítulos 5, 6, 7 e 8.

- **Programação em Lógica e Prolog** (60 horas). O livro é indicado para a parte da linguagem Prolog. Todos os capítulos da Parte I, mais alguns da Parte III.
- **Fundamentos de programação para IA** (60 horas). Começando com Curso rápido de Prolog. Mais alguns tópicos da parte III.
- **Fundamentos de Prolog para Linguagens Formais e Compiladores** (entre 20 e 60 horas). Ou para o desenvolvimento de processadores de linguagens. Começando com o Curso rápido de Prolog, mais os capítulos sobre Programação de Gramáticas e PLN.
- **Paradigma Lógico** dentro de uma disciplina de Linguagens de Programação (10 horas). Os primeiros quatro capítulos da Parte I, ou apenas o capítulo 1 e parte do capítulo 2 (só duas horas-aula).

Sobre o autor

O autor é Bacharel em Ciências da Computação pela UFRGS (1983-1987), Mestre em Ciências da Computação pela UFRGS (1987-1990) e é Doutor em Ciências da Computação com ênfase em Computação Inteligente pela UFPE (1996-2000). Professor e pesquisador do Departamento de Informática da UFPA desde 1991, atua também nos cursos de Mestrado e Doutorado do Programa de Pós-Gradução em Engenharia Elétrica e Computação (PPGEEC) da UFPA, na subárea de Computação Aplicada. O autor vem desenvolvendo pesquisa em algumas subáreas da Inteligência Artificial (IA): IA Simbólica, Processamento de Linguagem Natural e Agentes Inteligentes para Web.

Sumário

Lista de Figuras

Parte I

Fundamentos e Técnicas de Programação em Prolog

CAPÍTULO 1

Introdução ao Prolog

Alan Robinson (1965)
Resolução e unificação
|
R. Kowalski (1972)
Interpretação procedural
|
Alain Colmerauer (1973)
Primeiro Prolog
|
David Warren (1977)
Prolog Edimburgo

Inicialmente apresentamos um histórico do Prolog. Em seguida, uma introdução à linguagem Prolog, definindo a nomenclatura básica, que inclui os conceitos de fato, regra, pergunta, cláusula, predicado, procedimento e programa.

Os conceitos de programação introduzidos aqui são detalhados e exemplificados nos demais capítulos da primeira parte do livro. Maiores detalhes sobre a sintaxe do Prolog são encontrados na segunda parte deste livro-texto.

1.1 Histórico

Um programa Prolog é uma coleção de fatos e de regras que definem relações entre os objetos do discurso do problema. Uma computação em Prolog envolve a dedução de conseqüências a partir das regras e fatos. O significado do programa é o conjunto de todas as conseqüências deduzíveis pela iterativa aplicação das regras sobre os fatos iniciais e os novos fatos gerados. Portanto, Prolog é uma linguagem com uma fundamentação lógica, em teoria da prova.

A linguagem de programação Prolog se originou a partir de trabalhos de pesquisa em procedimentos de resolução para lógica de primeira ordem. Mais precisamente, o nascimento do Prolog aconteceu por volta de 1965, quando Alan Robinson desenvolveu os dois componentes-chave de um provador de teoremas para lógica de cláusulas (equivalente à lógica de primeira ordem):

- o procedimento de resolução;
- o algoritmo de unificação.

A partir do resultado teórico de 1965, várias tentativas foram feitas para desenvolver uma máquina de computação baseada no princípio de resolução, o que só aconteceu na década de 70.

Por um lado, R. Kowalski [12], em 1972, formulou uma interpretação procedimental para as cláusulas de Horn (uma versão restrita da lógica de cláusulas, para a qual Robinson desenvolveu a resolução). Ele mostrou que uma cláusula lógica, tal como **A if B1 and B2**, que equivale à implicação lógica **(B1 and B2) -> A**, pode ser lida e executada como um procedimento em uma linguagem de programação recursiva, na qual o `A` é a cabeça do procedimento e os `B1 and B2` são o seu corpo. Assim, o programa

```
prog if ler(Dado) and calcular(Dado, Result) and impr(Result)
```

é lido como: para executar `prog` executa-se `ler(Dado)` e depois executa-se `calcular(Dado, Result)` e então executa-se `impr(Result)`. Esta leitura equivale a um procedimento imperativo:

```
procedure prog;
  begin
     ler(Dado);
     calcular(Dado, Result);
     impr(Result);
  end;
```

Esta interpretação de cláusulas é consistente com o processo de resolução, no qual a unificação é quem faz a manipulação dos dados: atribuição de valores, construção de estruturas de dados, seleção de dados e passagem de parâmetros. Assim, o surgimento da Programação em Lógica (PL), em 1972, resultou da integração destes três conceitos: resolução, unificação e cláusulas de Horn.

Prolog vem de **PROgramming in LOGic.** Em 1973, Alain Colmerauer [6] na Universidade de Aix-Marseille desenvolveu um provador de teoremas para implementar sistemas de Processamento de Linguagem Natural, chamado **Prolog** (Programation et Logique), que empregava a interpretação de Kowalski. Em 1974, David Warren [17] esteve dois meses em Marseille, onde conheceu a linguagem Prolog e escreveu um programa chamado Warplan, que resolvia um problema de planejamento (busca de uma lista de ações, para, a partir de um estado inicial, chegar a um estado final). De volta a Edimburgo e pensando em um tópico para sua tese, Warren estava estudando a idéia de construir um compilador para Prolog a fim de resolver o problema da fraca performance

dos interpretadores de Prolog, pois, nesta época, havia um consenso de que sistemas provadores de teoremas baseados em lógica eram irremediavelmente ineficientes. Em 1977, na Universidade de Edimburgo (na Escócia), David Warren conseguiu desenvolver o primeiro compilador para Prolog, chamado de **Prolog-10**, também conhecido como Prolog padrão Edimburgo [15]. Este compilador e outros desenvolvidos nesta época ainda não satisfaziam o requisito de performance. Em 1983, Warren publicou um trabalho definindo as instruções para uma máquina abstrata de Prolog, hoje conhecida como WAM (Warren abstract machine) [2] que é a base de todas as implementações comerciais e acadêmicas hoje disponíveis. Com relação aos compiladores anteriores, uma WAM apresentava as seguintes técnicas de otimização: pontos de escolha separados dos ambientes dos predicados; passagem de argumentos por registradores (em vez da pilha); e escolha de cláusulas usando índices.

Programação em Lógica vs. Prolog

Programação em Lógica (PL) não é sinônimo de Prolog. Inicialmente, em PL temos duas grandes classes de programas: cláusulas definidas (ou Horn) e não definidas (estudadas no próximo capítulo). Prolog é um sistema que executa programas para cláusulas definidas. Por outro lado, muitos programas em Prolog não têm uma representação em PL pura, como veremos nos próximos capítulos. Na prática, a PL, como disciplina de fundamentos teóricos da computação, deve seu sucesso ao Prolog. Na seqüência, apresentamos uma introdução ao Prolog. PL é retomada no próximo capítulo.

1.2 Programa = regras + fatos

Um programa Prolog é uma coleção de fatos e de regras que definem relações entre os objetos do discurso do problema. Uma computação em Prolog envolve a dedução de conseqüências a partir das regras e fatos. O **significado de um programa** é o conjunto de todas as conseqüências deduzíveis pela iterativa aplicação das regras sobre os fatos iniciais e os novos fatos gerados.

```
1  %% programa da familia
2  pai(tare, abraao).  %1
3  pai(tare, nacor).   %2
4  pai(tare, aran).    %3
5  pai(aran, lot).     %4   /* 7 fatos */
6  pai(aran, melca).   %5
7  pai(aran, jesca).   %6
8  mae(sara, isaac).   %7
9  %%
10     fem(X):-mae(X,Y).                  %1
11 irmao(X,Y):-pai(P,X),pai(P,Y), X\==Y. %2   /* 3 regras */
12   tio(T,X):-pai(P,X),irmao(P,T).       %3
```

1.2.1 Regras e fatos em Prolog

Um **programa** em Prolog é uma coleção de unidades lógicas chamadas de predicados. Cada **predicado** é uma coleção de cláusulas. Uma **cláusula** é uma regra ou um fato.

O programa da família, em Prolog, apresenta a árvore genealógica da família bíblica de Abraão. O programa descreve um conjunto de relações lógicas envolvendo os objetos do domínio do problema, que são as pessoas da família bíblica identificadas pelos seus nomes. Estas relações lógicas são estabelecidas através das cláusulas do programa.

Como convenção nos textos-fonte dos programas não usamos a acentuação nas palavras e nomes, como está exemplificado no programa mencionado. Segue uma descrição sintática dos elementos presentes no programa da família bíblica.

1.2.2 Termos e predicados

Em um programa em Prolog:

- Uma **variável** representa um elemento não especificado do domínio. Sintaticamente, uma variável sempre inicia com letra maiúscula;

- Uma **constante** representa um elemento específico do domínio. Pode ser numérica ou uma cadeia de caracteres (tipicamente, iniciando com letra minúscula).

No programa da família bíblica, são constantes todos os nomes das pessoas: `tare, abraao, nacor, sara, ...`; e, são variáveis: `X, Y, T, P.`

Regras e fatos, em Prolog, são representados por cláusulas (no programa da família bíblica, temos uma regra ou fato em cada linha). Assim, em Prolog, um programa é um conjunto de cláusulas; cada cláusula corresponde a uma fórmula lógica. **Cláusulas** são agrupadas em predicados. Um **predicado** é definido por um conjunto de regras e fatos com o mesmo nome (por exemplo, `pai` no programa da família bíblica).

Revisando o programa da família bíblica, podemos identificar duas partes, a primeira descrita por **fatos**, definidos pelas relações `pai/2` e `mae/2`; e a segunda pelas **regras** `irmao/2`, `fem(inina)/1` e `tio/2`. `pai/2` denota o predicado `pai` com dois argumentos, enquanto `fem/1` denota o predicado `fem` com um argumento. O predicado `pai/2` é definido por seis fatos. Em resumo, o programa da família bíblica é descrito por sete fatos e três regras que definem cinco predicados.

Mais formalmente, cláusulas são fórmulas lógicas construídas a partir de fórmulas atômicas:

- Se `P` é uma fórmula atômica, então a sua negação também é uma fórmula, representada em Prolog por `\+ P`.

- Se `P` e `Q` são fórmulas, então também a conjunção `(P,Q)`, a disjunção `(P;Q)` e a condicional `(P:-Q)` são fórmulas, não atômicas.

Fórmulas não atômicas são formadas por operadores lógicos (`, ; :- \+`), respectivamente (e, ou, condicional, negação). O símbolo `:-` representa uma implicação lógica invertida, chamada de condicional. Assim, `P:-Q` é equivalente a $Q \rightarrow P$, como veremos no próximo capítulo. Um fato, por exemplo, `pai(tare,nacor)` também representa uma regra condicional: `true` $\rightarrow$ `pai(tare,nacor)`. Fórmulas atômicas são construídas a partir de **termos**. No programa da família bíblica, são exemplos de fórmulas: `pai(tare, nacor), fem(X), pai(P,X), ...`

Termos são construídos a partir de variáveis e constantes. Como segue:

- Toda constante é um **termo** e toda variável é um **termo**;

- Se $t_1, .., t_n$ são termos e f é um símbolo funcional, então $f(t_1, .., t_n)$ também é um termo;

- Se $t_1, .., t_n$ são termos e p é um símbolo predicativo, então $p(t_1, .., t_n)$ também é um termo, que corresponde a um predicado.

Assim, um termo é um símbolo seguidos de n argumentos: $t(t_1, .., t_n)$. Nesta representação t é chamado de **functor**. Um functor pode ser um símbolo funcional ou um símbolo predicativo.

Quando $n = 0$ temos dois casos: se o símbolo é funcional temos uma constante que é um termo; e se o símbolo for predicativo temos uma constante lógica que é uma fórmula atômica, por exemplo, `true` ou `false`.

Functores com um ou dois argumentos podem também ser escritos na forma de operadores. Por exemplo, `1+2` corresponde a `+(1,2)`, `-1` corresponde a `-(1)`; e `1=2` corresponde a `=(1,2)`. O uso de operadores em Prolog tem como objetivo tornar a escrita de termos e predicados mais econômica e fácil, utilizando menos parênteses e vírgulas. Compare: `+(1,2)` com `1+2`, o primeiro utiliza três caracteres a mais, dois parênteses e uma vírgula.

Representação de conhecimento

Prolog é uma linguagem declarativa, na qual a computação é baseada na representação do conhecimento do domínio do problema. Nesta representação devemos considerar todos os objetos sobre os quais queremos falar, as propriedades de cada objeto e as relações entre objetos.

Termos definidos por **símbolos funcionais**, por exemplo, `idade(X)`, representam funções que retornam valores que são constantes no domínio do problema. Por exemplo, `idade(joao)` retornando 25, `sexo(joao)` retornando `masculino`. Idade e sexo são atributos ou qualidades de um indivíduo. Termos definidos por **símbolos predicativos** são predicados. Eles representam regras e fatos do Prolog. Um predicado representa uma relação lógica (verdadeira ou falsa) entre indivíduos, por exemplo, `pai(Joao,Maria)` e `irmao(X,Y)`. Formalmente, um predicado representa uma fórmula que é um objeto lógico, isto é, retorna sempre um valor lógico (verdadeiro ou falso).

Em **linguagem natural**, existe também uma diferença semântica entre um símbolo funcional e um símbolo predicativo. O primeiro representa uma oração incompleta, ou um nominal, por exemplo, `idade(X)`, `idade(joao)` pode ser lido como *A idade de X ... , A idade de João....* O segundo representa sempre uma oração completa, por exemplo, o predicado `idade(joao,23)` é lido como *A idade de João é 23;* e o predicado `forte(joao)` é lido como *João é forte.*

No programa da família bíblica temos a codificação do conhecimento nas cláusulas em duas formas:

- explícito: indivíduos (constantes) e fatos (ou relações) entre os indivíduos, `pai/2` e `mae/2`;
- implícito: as regras `fem/1`, `tio/2` e `irmao/2`, que geram novos fatos a partir dos fatos iniciais.

1.2.3 Convenções para leitura de cláusulas

Para um fato com 2 parâmetros podemos ter duas leituras trocando-se de ordem os parâmetros. Para o fato `pai(tare,abraao)` temos duas leituras: *Taré é pai de Abraão* ou *Abraão é pai de Taré*. Preferimos a primeira forma, porém em Prolog não existe uma convenção padrão, fica a critério do programador.

Numa regra `cabeca:-corpo`, o corpo pode ser constituído de uma lista de predicados, ligados por vírgula, denotando uma conjunção lógica. A regra `fem(X):-mae(X,Y)` é lida como uma fórmula condicional: (*X é feminino*) **se** (*X é mãe de Y*). E a regra `tio(T,X):-pai(P,X),`

`irmao(P,T)` é lida como uma fórmula condicional: (*T é tio de X*) **se** (*P é pai de X) e (P é irmão de T)*.

Abaixo, segue uma lista de exemplos de leitura de cláusulas:

`pai(tare, abraao).`
leia-se *Tare é pai de Abraão*

`mae(sara, isaac).`
leia-se *Sara é mãe de Isaac*

`fem(X) :- mae(X,Y).`
leia-se (*X é feminina*) **se** (*X é mãe de Y*)

`irmao(X,Y) :- pai(P,X),pai(P,Y),X\==Y.`
leia-se (*X é irmão de Y*) **se** *(P é pai de X) e (P é pai de Y) e (X é diferente de Y)*

1.2.4 Perguntas

O significado de um programa é o conjunto de conseqüências que são deduzidas a partir dos fatos e regras. Assim sendo, um sistema Prolog (isto é, um sistema que executa um programa Prolog) pode ser visto como um interpretador de perguntas. Para sabermos se um fato arbitrário é conseqüência do programa, perguntamos ao sistema, que responde Sim (Yes) se for verdade e Não (No) caso contrário.

```
?- fem(sara).
   Yes.
?- fem(aran).
   No
?- fem(X).
   X=sara
   Yes
?- irmao(melca, lot).
   Yes
?- irmao(lot,X).
   X=melca; X=jesca
   Yes
?- tio(nacor,jesca).
   Yes
```

Acima, temos uma lista de perguntas que podemos fazer ao interpretador quando está executando o programa da família bíblica.

Na primeira pergunta, `?-fem(sara)` *(Sara é feminina?)*, o sistema respondeu `Yes`. Para deduzir a resposta, o sistema usou a regra 1 e o fato 7, como segue:

`mae(sara, isaac)`	*Sara é mãe de Isaac*
`fem(X):-mae(X,Y)`	*(X é feminina)* **se** *(X é mãe de Y)*
———	———
`fem(sara)`	*Sara é feminina*

Este mesmo esquema responde à pergunta `?-fem(X)` (*Quem é feminina?*). O fato deduzido `fem(sara)` é comparado com o predicado `fem(X)` onde se deduz que `X=sara`.

De forma similar, buscando conseqüências lógicas, o interpretador responde todas as perguntas apresentadas acima. A segunda pergunta é: *Aran é feminino?*. A resposta é Não, pois não existem

fatos do tipo `mae(Aron,X)`. Para esta dedução o Prolog assume a **hipótese do mundo fechado**: tudo o que não está descrito nos fatos ou não é deduzível dos fatos é falso. A terceira pergunta é: *Quem é feminina?*. O sistema retorna `X=sara`, dizendo que *Sara* é quem é feminina. Mais adiante, pergunta-se *Quem é irmão de Lot?* `irmao(lot,X)` — o sistema dá duas respostas: *Melca* e *Jesca*. Ambos valores podem ser deduzidos para a variável `X`. E assim por diante.

1.3 Revisão

Exercício 1.1 *Com base nos esquemas abaixo e no texto do capítulo, defina os conceitos solicitados. Seja breve, utilize no máximo 20 palavras, incluindo pelo menos um exemplo.*

$$
\begin{array}{l}
\%\% \; fatos \;\&\; regras \\
\text{fatos} \left\{ \begin{array}{l} \texttt{pai(tare, abraao).} \\ \texttt{pai(tare, nacor).} \end{array} \right. \\
\%\% \\
\text{regras} \left\{ \begin{array}{l} \overbrace{\texttt{irmao(X, Y)}}^{\text{cabeca}} :- \overbrace{\texttt{pai(P, X), pai(P, Y), X\ == Y}}^{\text{corpo}}. \\ \texttt{fem(X)} :- \texttt{mae(X, Y).} \end{array} \right.
\end{array}
$$

$$
\begin{array}{l}
\%\% \; predicados \;\&\; cláusulas \\
\text{predicado} \left\{ \begin{array}{ll} \texttt{pai(tare, nacor).} & \}\text{cláusula} \\ \texttt{pai(tare, aran).} & \}\text{cláusula} \end{array} \right. \\
\text{predicado} \left\{ \begin{array}{ll} \texttt{fem(ana).} & \}\text{cláusula} \\ \texttt{fem(X) :- \textbackslash + masc(X).} & \}\text{cláusula} \\ \texttt{fem(X) :- mae(X, Y).} & \}\text{cláusula} \end{array} \right.
\end{array}
$$

1. Cláusula e predicado.

2. Regra, fato e pergunta.

Exercício 1.2 *Idem, defina os termos abaixo.*

$$
\begin{array}{l}
\%\% fórmulas \\
\overbrace{\overbrace{\overbrace{\texttt{irmao(X, Y)}}^{\text{atômica}} :- \overbrace{\texttt{pai(P, X)}}^{\text{atômica}}, \overbrace{\texttt{pai(P, Y)}}^{\text{atômica}}}^{\text{não atômica}}}^{\text{fórmula não atômica}}. \\
\\
\%\% \; termos \; simples \;\&\; compostos \\
\overbrace{\texttt{pessoa}(\overbrace{\texttt{joao} - \texttt{silva}}^{\text{termo/const}}, \underbrace{\texttt{ender}}_{\text{functor}} \underbrace{(\texttt{rua} - \texttt{abc}, \overbrace{\texttt{2700}}^{\text{termo/const}})}_{\text{argumentos}}, \overbrace{\texttt{CIC}}^{\text{termo/var}})}^{\text{termo composto}}.
\end{array}
$$

1. Variável.
   Solução: Uma variável representa um elemento não especificado do domínio do problema, inicia por letra maiúscula, por exemplo, `X, Maior`.

2. Constante.

3. Termo simples e termo composto.

4. Functor.
   Solução: Um functor é o símbolo que precede os argumentos num termo composto, por exemplo, em `pai(joao, ana)`, o functor é `pai`.

5. Argumento.

6. Operador.

7. Fórmula atômica e não atômica.

Exercício 1.3 *Idem, defina os termos abaixo.*

1. Símbolo funcional e símbolo predicativo.

2. Oração completa e oração incompleta.

3. Relação lógica entre indivíduos.

1.4 SWI-Polog

Neste livro, a partir deste capítulo, assumimos que o leitor tem disponível um sistema Prolog para rodar os programas e fazer os exercícios de programação. Recomendamos o uso do SWI-Prolog, porém existem dezenas de sistemas Prolog disponíveis no mercado (sistemas comerciais ou acadêmicos): Amzi!^{tm}, ALS^{tm}, Arity^{tm}, Eclipse^{tm}, LPA^{tm}, Quintus^{tm}, SICStus^{tm}, SWI^{tm}, XBS^{tm}. Praticamente todos estes sistemas rodam nas duas principais plataformas de sistemas operacionais: Windows^{tm} e Linux^{tm}.

O **SWI-Prolog** é desenvolvido pela Universidade de Amsterdã, é um software com distribuição livre para ensino e pesquisa. Ele usa poucos recursos da máquina, é rápido para compilação e possui documentação on-line (`http://www.swi-prolog.org/`). Ele implementa uma versão do Prolog ISO, que é o padrão adotado pelo livro, o que significa que os exemplos do livro podem ser executados neste Prolog sem necessidade de ajustes.

Na Figura 1.1 vemos um uso típico do SWI-Prolog. Ele é integrado a um editor compatível com o Emacs (editor UNIX), o qual só é recomendável para programadores que conhecem os comandos do Emacs. Programas Prolog são geralmente pequenos, podendo ser editados em qualquer editor de programas, como mostrado na Figura 1.1. É necessário que o editor enumere as linhas para identificar um erro após uma compilação.

Após instalado, o sistema SWI-Prolog, por default, se associa com todos os arquivos com extensão `'.pl'`. Assim, basta um duplo clique sobre um arquivo `'.pl'` para rodar o SWI-Prolog.

1.4.1 Entrando no SWI-Prolog

Podemos rodar o SWI-Prolog pelo modo usual, caminhando pelos menus, por exemplo: *Iniciar* $>$ *Programas* $>$ *SWI-Prolog*. **Nós preferimos duplo clique sobre um arquivo de programa `'.pl'` pois o interpretador reconhece o caminho até o local onde está o arquivo;** caso contrário é necessário indicar todo o caminho.

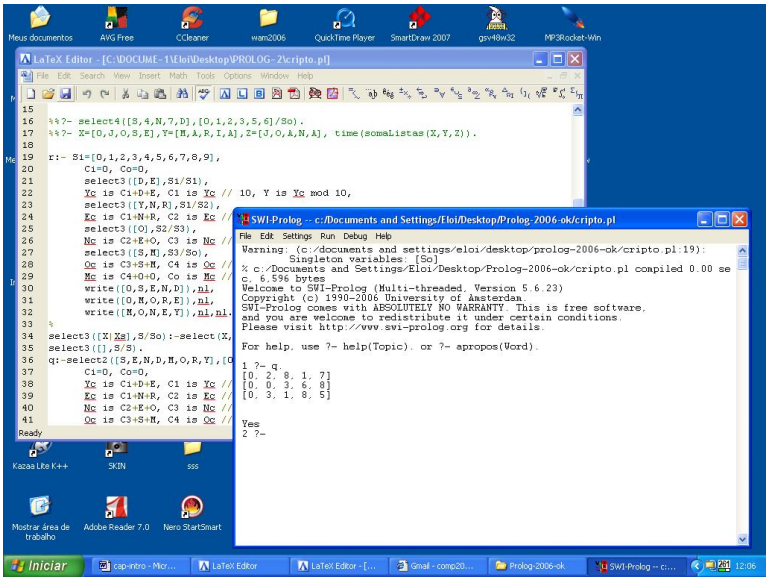


Figura 1.1: O SWI-Prolog: um editor e o interpretador de perguntas.

Num sistema Prolog, a maneira usual de rodar um programa é fazendo perguntas ao interpretador de comandos. Por exemplo, podemos perguntar se a=a e o sistema deve responder Sim (Yes). E, se perguntarmos se a=b, o sistema deve responder Não (No). Podemos também escrever o clássico: `write('Alo mundo'),nl`.

```
?-a=b.
   No
?-a=a.
   Yes
?-write('Alo mundo'),nl.
  Alo mundo
  Yes
```

No fim de cada pergunta devemos digitar um ponto final, seguido de *enter*. Para cada pergunta o interpretador sempre responde com Yes ou No.

1.4.2 Rodando um programa

O próximo passo é rodar um programa, isto é, carregar um conjunto de fatos e regras na memória para podermos perguntar sobre eles.

Convenção. Adotamos a seguinte convenção nos capítulos do livro: (1) Quando um texto em fonte courier (para programas) aparece dentro de uma caixa com os quatro lados, estaremos fazendo referência ao uso de um interpretador Prolog. Neste caso devem aparecer perguntas que iniciam com ?- (veja acima um exemplo). (2) Quando um texto em fonte courier aparecer delimitado por uma caixa sem as laterais e com as linhas enumeradas, se trata de um programa. Propositalmente, estes textos de programas, em fonte courier, não são acentuados.

Existem duas formas de carregar um programa fonte: duplo clique sobre o arquivo com extensão `'.pl'` ou usando o predicado `consult`. Com estas explicações, já estamos em condições de carregar o primeiro programa e em seguida fazer as primeiras perguntas. Digite em qualquer editor o programa que segue e salve-o como `'pai.pl'`.

```
1 %% pai.pl
2 pai(pedro, maria).
3 pai(pedro, ana).
```

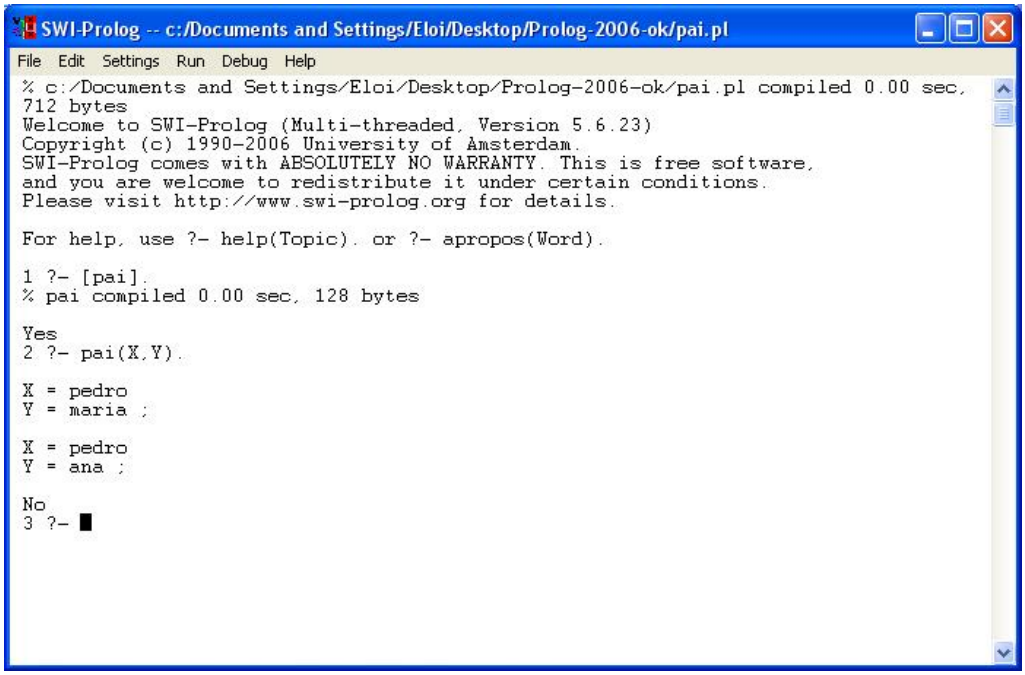


Figura 1.2: O interpretador de comandos do SWI-Prolog: o programa pai.pl está sendo consultado.

A Figura 1.2 mostra o interpretador de perguntas rodando (foi ativado clicando-se sobre o programa `'pai.pl'`). Um duplo clique sobre um arquivo de programa faz duas coisas: entra no Prolog e carrega o arquivo clicado.

Se já estamos dentro do Prolog, carregamos um programa perguntando `?-consult( arquivo)`. Este comando pode também ser simplificado na forma: `?-[arquivo]`. Assim, para carregar o arquivo `'pai.pl'` pergunta-se: `?-consult(pai)` ou `?-[pai]` (**No final de cada pergunta ou cláusula de programa devemos sempre digitar um ponto final.**). Agora podemos fazer perguntas sobre o programa já carregado. Por exemplo, perguntando `?-pai(pedro, ana)` o sistema responde Yes; e perguntando `?-pai(X,Y)` o sistema dá duas respostas. Para obter a segunda resposta digita-se ponto e virgula (;) depois da primeira resposta.

```
?- consult(pai).
% pai compiled 0.00 sec, 0 bytes
   Yes
2 ?- [pai].   %% mesmo que consult()
% pai compiled 0.00 sec, 0 bytes
   Yes
3 ?- listing(pai). %% lista o predicado pai
pai(pedro, maria).
pai(pedro, ana).
   Yes
4 ?- pai(pedro,maria).
   Yes
```

```
5 ?- pai(X,Y).
   X = pedro, Y = maria ; %% digite (;)
   X = pedro, Y = ana ;
  No
```

O processo de programação consiste em criar ou modificar o texto de um programa, salvá-lo e, a seguir, reconsultá-lo. No SWI-Prolog uma reconsulta é feita repetindo-se a pergunta `?-consult(aquivo)`. Em alguns sistemas só a primeira vez carregamos com `consult`, nas outras vezes devemos usar `reconsult`. Em resumo, trabalha-se com o editor de programas em uma janela e o SWI-Prolog em outra(s), repetindo o ciclo: editar, salvar, reconsultar e perguntar.

Uma parte trabalhosa da programação em Prolog é a digitação das perguntas. Para facilitar o trabalho, o SWI-Prolog guarda um histórico das perguntas que podem ser reexecutadas sem serem redigitadas, **basta caminhar com as setas up/down para resgatá-las**.

1.5 Erros de sintaxe

O Prolog possui uma sintaxe bem simples. Porém, para um principiante o diagnóstico sobre os erros de sintaxe é um tanto vago. Segue uma lista comentada de erros comuns.

Falta de ponto final

Um erro comum é a falta do ponto final, depois de uma pergunta; ou, num arquivo de programa, depois de uma cláusula.

Usar sempre apóstrofo com nomes de arquivos

Um dos primeiros erros é tentar usar aspas no lugar de apóstrofos nos comandos de `consult` e `reconsult` ou `['arquivo.pl']`[1]. A extensão `.pl` é opcional: basta escrever `[arquivo]`. Se o nome do arquivo é uma palavra que inicia com letra minúscula e sem extensão, não é necessário usar apóstrofos.

Usar aspas causa um problema, porque um conjunto de caracteres entre aspas corresponde a uma lista dos códigos ASCII dos caracteres, por exemplo, `?- "abc"= [97,98,99]` Portanto, `consult("abc")` é o mesmo que `consult([97,98,99])`. Sempre devemos usar apóstrofos: `?- ['abc']. ?-consult('abc').` Para especificar um caminho para um arquivo também use apóstrofos, por exemplo, `?- consult('C:/ prolog/ pai.pl').`

Variáveis iniciam por maiúscula e nomes de predicados por minúsculas

Para um iniciante é comum escrever variáveis começando com letra minúscula. Estes erros não são sintáticos. Por exemplo, se perguntarmos `?-pai(x,y)` o sistema responde No. Por outro lado, se escrevermos o nome de predicado começando em maiúscula, o SWI-Prolog mostra um erro. Note que aqui o erro não é indicado na posição correta - início do functor "Pai".

```
?- Pai(X,Y).
   ERROR: Syntax error: Operator expected
   ERROR: Pai(X,Y
```

[1] Em alguns sistemas Prolog a extensão é ".pro"; ou também podemos escolher a extensão default associada ao Prolog.

```
   ERROR: ** here **
   ERROR: ).
```

Não deixar brancos entre o nome do predicado e o abre parênteses

O nome do predicado deve ser colado ao abre parênteses. Note que, também, aqui o erro não é indicado na posição correta.

```
?- pai (X,Y) .
ERROR: Syntax error: Operator expected
ERROR: pai (X,Y
ERROR: ** here **
ERROR: ) .
```

Tentar definir um predicado predefinido

Se tentarmos carregar um programa que tem um predicado que é do sistema, por exemplo, um arquivo chamado `mm.pl` com `member(a,[a])`. O sistema acusa o erro abaixo:

```
?- [mm].
ERROR: (c:/prolog/mm.pl:3):
        No permission to modify static_procedure ‘member/2’
% pai compiled 0.00 sec, 16 bytes
```

Muitos dos predicados vistos neste livro-texto são predefinidos no SWI-Prolog, tais como: `append`, `member`, `select`, etc. Para usá-los devemos adicionar um sufixo, por exemplo, um dígito inteiro: `append1`.

Avaliar uma expressão aritmética com variáveis livres

Uma expressão aritmética, para ser avaliada, deve ter suas variáveis fechadas.

```
?- X is 1+ B. %% B não está instanciado
   ERROR: Arguments are not sufficiently instantiated
```

Usar operadores não definidos

Por exemplo, a operação de divisão inteira, para o ISO Prolog, não é `div`, mas sim `//` (no entanto, `div` é padrão no Prolog Edimburgo). Na tentativa de usar `div`, o SWI-Prolog responde como ilustrado abaixo.

```
?- X is 10 div 2.
ERROR: Syntax error: Operator expected
ERROR: X is 10
ERROR: ** here **
ERROR: div 2 .
```

No próximo capítulo, apresentamos uma fundamentação teórica para este tipo de raciocínio computacional baseado em deduções lógicas.

CAPÍTULO 2

Teoria da Programação em Lógica

Neste capítulo, apresentamos uma visão geral dos fundamentos teóricos da Linguagem Prolog, isto é, a teoria lógica subjacente ao Prolog. Partindo da lógica de proposições, chega-se ao procedimento de resolução com unificação. A apresentação é informal, sem provas de teoremas. Para um estudo detalhado deste tema são indicadas algumas referências no final do capítulo.

Este capítulo é opcional para cursos rápidos. Isto é, pode-se alcançar uma proficiência aceitável em Prolog, sem conhecer os seus fundamentos teóricos. No entanto, quanto mais o aluno domina os conceitos deste capítulo mais segurança e firmeza terá no desenvolvimento de programas em Prolog.

Em computação estudamos dois principais sistemas lógicos. A lógica de **proposições** (ou sentencial) estuda fórmulas do tipo $A \vee \neg B \rightarrow C$ e a lógica de **predicados** (ou lógica de primeira ordem) estuda fórmulas com ocorrências dos quantificadores universal e existencial, por exemplo: $\forall X \forall Y \exists P(pai(P, X) \wedge pai(P, Y) \wedge (X \neq Y) \rightarrow irmao(X, Y))$. O Prolog implementa um modelo lógico chamado de lógica de cláusulas definidas que está entre a lógica das proposições e a lógica dos predicados.

2.1 Lógica proposicional

O formalismo lógico mais simples é a lógica proposicional (também conhecida como cálculo proposicional ou álgebra Booleana).

Uma mulher não pode voar. *Onça Pequena não pode voar.* *Portanto, Onça Pequena é uma mulher.*	Argumento I
Um pássaro pode voar. *Sabiá Pequeno é um pássaro.* *Portanto, Sabiá Pequeno pode voar.*	Argumento II

Orações completas podem ser traduzidas para fórmulas da lógica proposicional. Em linguagem natural podemos escrever **argumentos** como uma seqüência de proposições, na qual a última proposição é a conclusão e as outras são as premissas.

Como exemplos, temos os argumentos I e II acima. Mesmo sem conhecer um modelo no mundo real (*Quem é Onça Pequena e Sabiá Pequeno?*), podemos ver que a partir das duas premissas não dá para deduzir logicamente que *Onça Pequena* é uma mulher. Este argumento pode ser representado[1] com um esquema de prova na lógica de proposições, como segue:

$$\{A \rightarrow \neg B, C \rightarrow \neg B\} \vdash C \rightarrow A$$

sendo *A=uma mulher; B=pode voar; C=Onça Pequena;* e, o símbolo $\vdash$ denota uma dedução ou prova. Por outro lado, a conclusão sobre o *Sabiá Pequeno* é válida (pela regra *modus ponens*). Esta argumentação pode ser representada como uma prova:

$$\{A \rightarrow B, C \rightarrow A\} \vdash C \rightarrow B$$

sendo *A=um pássaro; B=pode voar; C=Sabiá Pequeno*.

O estudo da lógica de proposições visa ao desenvolvimento de sistemas de prova formal para validar mecanicamente argumentos como I e II. Estes sistemas de prova para proposições são também conhecidos como cálculo para proposições[2].

Elementos da lógica proposicional

As unidades básicas da lógica proposicional são as constantes (`true`, `false`) e os identificadores (letras maiúsculas) que denotam proposições (orações completas) que são verdadeiras ou falsas.

As unidades básicas são combinadas em fórmulas pelos operadores lógicos; seguem os principais operadores lógicos: conjunção ($\wedge$, `&`), disjunção ($\vee$, `v`), negação ($\neg$, `~`), implicação ($\rightarrow$, `->`) e o condicional ($\leftarrow$, `<-`). A implicação e o condicional podem ser rescritos a partir dos outros, por exemplo, $A \rightarrow B$ equivale a $\neg A \vee B$.

[1] Esta representação deixa a desejar; em lógica dos predicados pode-se escrever uma versão mais fiel para este argumento.

[2] Um sistema bem conhecido é o Sistema de Dedução Natural, que consiste num conjunto de regras de inclusão e eliminação de conectivos, que são usadas para se fazer uma prova.

Em lógica de proposições, cada identificador pode assumir somente o valor verdadeiro ou falso, portanto, o **significado** dos operadores pode ser definido por tabelas-verdade, como exemplificado abaixo.

A	*B*	`~`*A*	*A* `&` *B*	*A* `v` *B*	*A* `->` *B*
true	*true*	*false*	*true*	*true*	*true*
true	*false*	*false*	*false*	*true*	*false*
false	*true*	*true*	*false*	*true*	*true*
false	*false*	*true*	*false*	*false*	*true*

Uma **interpretação** define um valor para cada um dos identificadores de uma fórmula. Por exemplo, a fórmula `A & B` é verdadeira para a interpretação { A=true, B=true }, mas é falsa para todas as outras interpretações. Algumas fórmulas são válidas para qualquer interpretação, por exemplo, `A v ~A`. E outras nunca são válidas, por exemplo, `A & ~A`.

Existem duas principais abordagens para se validar uma fórmula:

- o uso de tabelas-verdade;
- um sistema de cálculo ou dedução;

Para Programação em Lógica (PL), interessa-nos a segunda abordagem, na qual são usadas regras de igualdade e dedução para se reduzir algebricamente uma fórmula num valor verdade. Em PL uma destas regras se destaca pelo seu papel num sistema de resolução (para lógica dos predicados). Esta regra é a **regra de eliminação**, que pode ser enunciada como

`(A v B) & (~A v C)` $\vdash$ `(B v C)` ou como
`(A & (~A v B))` $\vdash$ `B` ou como
`(A & (A -> B))` $\vdash$ `B`

sendo que o símbolo $\vdash$ denota uma dedução ou prova.

Esta regra é chamada regra de eliminação porque, combinando duas subfórmulas, ela elimina o termo comum, que nestes enunciados é o `A`.

Exercício 2.1 *Usando a regra de eliminação prove que* `K` *é verdadeiro, a partir do conjunto de fórmulas dado abaixo:*

```
{F, H->I, H->J, J->K, F->G, F->H}
```

Exercício 2.2 *Usando a regra de eliminação prove que* `L` *é verdadeiro, a partir do conjunto de fórmulas dado abaixo:*

```
{F, H->I, H->J, H & J->L, F->G, F->H}
```

Uma dedução acontece em um metanível da linguagem. Nos exercícios acima, o `A` e o `B` da regra de eliminação `(A & (A -> B))` $\vdash$ `B` são substituídos por identificadores das fórmulas: `F,G,H,I,J,K,L`.

2.2 Lógica de cláusulas (para proposições)

Os exemplos de argumentação I e II são descritos como três orações: duas premissas e uma conclusão. Cada um destes argumentos pode ser representado numa única fórmula lógica: as premissas são ligadas por uma conjunção; e entre as premissas e a conclusão pode-se usar uma implicação. Com isso, temos que uma argumentação é válida se a sua fórmula é verdadeira.

Pelas leis da lógica de proposições, toda fórmula pode ser transformada numa **forma normal conjuntiva** que é uma conjunção de um conjunto de cláusulas. Cada cláusula é uma disjunção formada por identificadores com ou sem negação, também chamados de literais positivos e negativos, respectivamente.

Segue um exemplo de rescrita de uma fórmula como um conjunto de cláusulas na forma normal conjuntiva:

```
1    ((A v B) & (~A v C)) -> D
2  = ~((A v B) & (~A v C)) v D      %% definição de ->
3  = (~( A v B) v ~(~A v C)) v D    %% leis de morgam
4  = ((~A & ~B) v (A & ~C)) v D     %%   "
5  = (~A v A v D) &                 %% distribuição v
6    (~A v ~C v D) &
7    (~B v A v D) &
8    (~B v ~C v D)
9  = (A v D v ~A     ) &          %% forma
10   (    D v ~A v ~C) &          %% normal
11   (A v D v ~B     ) &
12   (    D v ~B v ~C)
```

Como resultado, a fórmula original é equivalente a quatro **cláusulas** ligadas por conjunções. Cada cláusula é uma disjunção com literais positivos e negativos. Se cada fórmula da lógica de proposições pode ser representada por uma ou mais cláusulas da lógica de cláusulas, então temos uma equivalência entre estes dois formalismos lógicos. Tudo o que pode ser expresso em lógica de proposições pode também ser expresso em lógica de cláusulas e vice-versa.

Uma forma de **notação abreviada** para cláusulas (na forma normal conjuntiva) foi proposta por Kowalski, como segue:

$A_1 \vee ... \vee A_m \vee \neg B_1 \vee ... \vee \neg B_n$ é representada como
$A_1, ..., A_m \leftarrow B_1, ...B_n$

Nesta notação abreviada, temos uma leitura mais simples: removeram-se todas as negações, os operadores de conjunção e disjunção; no lugar usam-se vírgulas.

Dada esta notação abreviada, temos a seguinte classificação para cláusulas:

- se `m>1` — `A1,A2,...<- ...` — Existem várias conseqüências ou conclusões. Estas são cláusulas **indefinidas**, pois a conclusão é uma disjunção: `A1 v A2 v ...`;
- se `m<=1` — `A <- ...` — Estas são as cláusulas de **Horn**. Existem as subclasses que seguem:
  - se `m=1,n>0` — `A <- B1,...Bn` — São as cláusulas **definidas** com uma só conseqüência ou conclusão, também chamadas de **regras** condicionais;
  - se `m=1,n=0` — `A <-` — São as cláusulas definidas incondicionais, também chamadas de **fatos**;

- se `m=0,n>0` — `<-B1,...Bn` — São as negações puras, também chamadas de **perguntas** ou cláusulas-objetivo;
- se `m=0,n=0` — `<-` — É a cláusula vazia.

Exercício 2.3 *Classifique as fórmulas abaixo, como cláusulas do tipo: 1) indefinida; 2) definida/regra; 3) definida/fato; 4) definida/pergunta.*

```
a)    (A1 v A2 v... ) <- (B1 & B2 & ... )
b)     A  <- (~B & C & D)
c)     false <- (~A & ~B & C & D)
d)     A v B v ~C v ~D <- true
e)     A v false <- B & true
f)     false <- true
g)     <- true
h)     false <-
i)     não chove <- faz_sol
j)     <- chove & faz_sol
```

Uma cláusula representa a generalização de um argumento: `conclusão <- premissas`. Ao mesmo tempo, uma cláusula é também uma generalização da definição do condicional ou implicação, como segue:

```
   A1 v A2 v... v  ~B1 v ~B2 v...
= (A1 v A2 v... ) v ~(B1 & B2 & ... )
= (A1 v A2 v... ) <- (B1 & B2 & ... )
```

Também é possível passar termos de um lado para o outro do condicional negando-se o termo movimentado. É a **regra de transposição de literais**, por exemplo:

```
   (A v B) <- (C & D)
 =      A <- (~B & C & D)
 = false <- (~A & ~B & C & D)
 = ...
 = (A v B v ~C v ~D) <- true
```

Algumas igualdades servem para clarificar o significado do condicional, como as que seguem:

```
  (A<-B)
= (A v false <- B & true)
= ( false <- ~A & B & true)
= ( false v A v ~B <- true)

  (false <- true)
= (<- true)
= (false <-)
=  <-
```

Um exemplo mais concreto ilustra o uso das transposições de literais, escrevendo variantes do mesmo condicional:

```
1    não chove <- faz_sol
2  = false <- chove & faz_sol
3  = não chove v não faz_sol <- true   %% "faz_sol" v "chove" <- true
4  = não faz_sol <- chove
```

Uma cláusula `A<-B` pode ser lida como "se B então A" ou "A se B": por exemplo, a regra `chove <- não faz_sol` é lida como "se chove então não faz sol" ou "chove se não faz sol". Assim sendo, uma cláusula é uma representação generalizada para um argumento: `conclusões <- premissas`.

2.2.1 Provas por refutação (para proposições)

Para a subclasse das cláusulas definidas a regra de eliminação pode ser descrita como:

```
1 A <- B, C, ...
2 D <- A, E, ...
3 --------------------
4 D <- B, C, ... E, ...
```

Aqui, o literal comum (`A`) é eliminado; este literal comum numa das cláusulas deve estar na premissa e na outra na conclusão. Esta regra é suficiente para provar se uma conclusão é deduzível (ou não) de um conjunto de cláusulas proposicionais.

Seja P um conjunto de cláusulas e Q uma cláusula a ser provada. Provar $(P \vdash Q)$ é provar que Q é derivado de P; que é equivalente a provar `P -> Q`; que é o mesmo que provar `P & ~Q -> false`. Esta última forma é a estratégia de prova por absurdo: junta-se a conclusão negada com as premissas para se chegar a uma falsidade. Esta prova é também chamada de refutação.

Queremos provar que `Q` é derivável (que é verdadeiro) a partir do seguinte conjunto de cláusulas definidas: `{ Q<-B,D | Q<-C,A | C<- | A<-D | D<- }`. Segue um exemplo de prova:

```
Q <- B, D    %1
Q <- C, A    %2
C <-         %3  Premissas {1..5}
A <- D       %4
D <-         %5
<-Q          %6  Pergunta {6}
<-B,D        %7 = 1x6
??? falhou B<-...

<-Q          %6  Pergunta {6}
<-C,A        %7 = 2x6
<-A          %8 = 7x3
<-D          %9 = 8x4
<-           %10= 9x5
```

Num sistema de refutação, a pergunta em forma de cláusula é representada por `<-Q`. A pergunta tem um literal comum com a cláusula %1; o literal `Q`. A regra de eliminação aplicada na pergunta e na cláusula %1 resulta na nova pergunta `<-B,D` %7. Agora, como `B` não acontece como conclusão de uma cláusula, sabemos que este caminho não leva a uma prova.

Então retornamos à pergunta inicial `<-Q` %6 e, agora, usamos a cláusula %2. Com a regra de eliminação obtemos a nova pergunta `<-C,A` %7. Agora, esta pergunta com a cláusula %3 resulta

em `<-A %8`, que com %4 resulta em `<-D %9`, que com %5 resulta na cláusula vazia, `<- %10`. Esta seqüência de passos é uma prova. A partir da pergunta inicial, em cada passo, é gerada uma nova cláusula-pergunta, até se chegar à cláusula vazia. Foram tentados dois caminhos, o segundo teve sucesso.

Esta prova se chama prova de **refutação** porque a pergunta `<-Q` na notação de cláusulas equivale à negação `~Q`. Chegar à cláusula vazia equivale a chegar a falso: `false =(X & ~X) =(X<- & <-X) = <-`.

2.3 Lógica de primeira ordem (ou predicados)

Onça Pequena pode amar.	
Todo ser que pode amar é humano.	Argumento III
Existe alguém que é humano?	

Uma prova por refutação pode responder perguntas do tipo *Existe alguém? Quem é?* Numa refutação nega-se a conclusão e chega-se a um absurdo. Assim, no argumento III, por um lado temos que *ninguém é humano* (negação da pergunta) e por outro lado deduzimos que *Onça Pequena é humana*, o que é um absurdo. Esta prova pode ser vista como *encontrar um contra exemplo*: *Ninguém é humano* vs. *Onça Pequena é humana*.

Para fazermos este tipo de raciocínio precisamos trabalhar com variáveis, que recebem valores não lógicos, por exemplo, indivíduos do universo do discurso: *Quem é o X? X=Onça Pequena.*

A lógica de proposições é um subcojunto de um sistema lógico mais geral chamado lógica de predicados (ou de primeira ordem). Esta estende a lógica de proposições em dois sentidos:

- uso de predicados parametrizados com indivíduos do discurso;
- uso de variáveis quantificadas, definidas para os indivíduos do discurso.

Em lógica de primeira ordem, as fórmulas são definidas a partir de predicados. Um predicado é definido a partir de um **símbolo predicativo**, que descreve relações e/ou qualidades dos indivíduos do discurso. Predicados podem também ser definidos para indivíduos arbitrários com o uso de variáveis. Seguem alguns exemplos: `ama(joao,maria)`, `ama(X,Y)` e `humano(joao)`. Estes exemplos são lidos como: *João ama Maria*, *X ama Y* e *João é humano*.

Os quantificadores existencial ($\exists$) e universal ($\forall$) permitem descrever com maior especificidade referências aos indivíduos do domínio, por exemplo, especificando se um predicado é válido para todos os indivíduos ou apenas um subconjunto deles.

Em lógica de primeira ordem podemos representar o argumento III por três fórmulas (duas premissas e uma conclusão):

premissa 1: $\forall X \exists Y (ama(X,Y) \rightarrow humano(X))$
premissa 2: $\exists Z (ama(oncapequena, Z))$
conclusão: $\exists W humano(W)$

Nosso objetivo é saber como representar argumentos como estes numa lógica de cláusulas baseada em predicados com parâmetros.

2.3.1 Cláusulas de Horn com predicados

Segue um programa descrito por um conjunto de cláusulas de primeira ordem ou cláusulas de Horn para predicados.

```
%% programa PAI %%
1. pai(tare, abraao) <-
2. pai(tare, nacor ) <-
3. mae(sara, isaac ) <-
4.            fem(X) <- mae(X,Y)
5.        irmao(X,Y) <- pai(P,X),pai(P,Y), X\==Y
```

O que diferencia este programa de um programa em cláusulas para proposições é o uso de predicados (com argumentos) no lugar dos identificadores proposicionais. Este programa, como um conjunto de cinco cláusulas, é equivalente a um texto em lógica de predicados, com cinco predicados, como segue (as fórmulas possuem a mesma numeração de linhas das cláusulas):

	Fórmulas lógicas
1.	$true \rightarrow pai(tare, abraao)$.
2.	$true \rightarrow pai(tare, nacor)$.
3.	$true \rightarrow mae(sara, isaac)$.
	$\forall X \forall Y (X = sara \wedge Y = isaac \rightarrow mae(X, Y))$.
5.	$\forall X \forall Z (mae(X, Z) \rightarrow fem(X))$.
6.	$\forall X \forall Y \forall P (pai(P, X) \wedge pai(P, Y) \wedge X \neq Y) \rightarrow irmao(X, Y)$.

Vemos que cada cláusula é representada por uma fórmula em lógica dos predicados. Além disso, todas as variáveis usadas nas cláusulas são ligadas a um quantificador universal ($\forall$). Na linha número 3, temos duas representações em lógica dos predicados para uma mesma cláusula. Este exemplo de representação alternativa é válido para todas as três primeiras cláusulas.

Para a lógica de cláusulas de Horn temos um resultado que diz: dada uma cláusula com uma ou mais variáveis, podemos criar uma nova instância da cláusula em que as variáveis são substituídas por indivíduos do universo de discurso. Este resultado é conhecido como **axioma da especificação geral**. Segue um exemplo:

```
1  fem(X)<-mae(X,Y)      {X/sara, Y/isaac}
2  ----------------------------------------
3  fem(sara)<- mae(sara,isaac)
```

Neste exemplo, `fem(sara) <- mae(sara,isaac)` é uma instância da regra `fem(X) <- mae(X,Y)` gerada pela substituição {`X/sara, Y/isaac`}.

Se toda a cláusula é universalmente quantificada, então necessariamente ela está definida para todos os indivíduos do universo do discurso. Porém, uma cláusula não necessariamente é verdadeira para todos os indivíduos do discurso. Por exemplo, a cláusula abaixo só é verdadeira para X=sara e Y=isaac.

$\forall X \forall Y (X = sara \wedge Y = isaac \rightarrow mae(X, Y))$.

2.3.2 O significado de um programa

Um programa é um conjunto de cláusulas. O *significado* de um programa é tudo o que pode ser deduzido dele. No programa PAI, abaixo, tudo o que pode ser deduzido é `irmao(abraao,nacor)` e `fem(sara)`.

```
%% programa PAI %%
pai(tare, abraao) <-
pai(tare, nacor ) <-
mae(sara, isaac ) <-
    fem(X) <- mae(X,Y)
irmao(X,Y) <- pai(P,X),pai(P,Y), X\==Y
```

Pelo axioma da especificação geral podemos rescrever o programa PAI num outro programa equivalente, como segue:

```
%% Programa PAI %%
pai(tare, abraao) <-
pai(tare, nacor ) <-
mae(sara, isaac ) <-
    fem(X) <- mae(X,Y)
irmao(X,Y) <- pai(P,X),pai(P,Y), X\==Y
-------------------------------------
fem(sara) <- mae(sara,isaac)
irmao(abraao, nacor)<- pai(tare,abraao), pai(tare,nacor),abraao\==nacor
```

Nesta versão, criamos uma instância para a regra `fem/1` e outra para o `irmao/2`. Agora, removendo as cláusulas com variáveis temos:

```
pai(tare, abraao) <-
pai(tare, nacor ) <-
mae(sara, isaac ) <-
fem(sara) <- mae(sara,isaac)
irmao(abraao, nacor)<- pai(tare,abraao), pai(tare,nacor),abraao\==nacor
```

Aqui não temos mais variáveis livres nas cláusulas. Uma cláusula sem variáveis livres é chamada de **cláusula fechada**. Este conjunto de cláusulas fechadas é equivalente a um programa em lógica de cláusulas de proposições, como segue:

```
P <-       % P=pai(tare, abraao)
Q <-       % Q=pai(tare, nacor)
M <-       % M=mae(sara, isaac)
F <- M     % F=fem(sara)
I <- P, Q  % I=irmao(abraao,nacor)
```

Pelo método de refutação é fácil ver que podemos deduzir `<-F` e `<-I`, que correspondem às únicas verdades deduzíveis do programa PAI: `irmao(abraao,nacor)` e `fem(sara)`.

O processo de resolução para cláusulas de primeira ordem é baseado nesta idéia. Ele consiste em um processo de substituição que gera novas instâncias de cláusulas a serem resolvidas pela regra de eliminação.

Exercício 2.4 *Prove por refutação* `<-F` *e* `<-I`*, onde* `F=fem(sara)` *e* `I=irmao(abraao,nacor)`.

2.3.3 Universo de Herbrand

Para o programa PAI, acima, de um modo inteligente, criamos apenas as duas instâncias de cláusulas necessárias para se gerar o resultado esperado do programa. No entanto, para tornar automático o processo de prova temos de usar uma abordagem que vale para qualquer programa.

O conjunto de todas as constantes mencionadas em um programa é chamado de **domínio de Herbrand** – são todos os indivíduos do discurso.

Na transformação de um programa que possui cláusulas com variáveis para um programa que possui somente cláusulas sem variáveis devemos, sistematicamente, criar todas as possíveis instâncias fechadas de uma cláusula e então remover a cláusula com variáveis. O que significa criar instâncias para todas as possíveis combinações de valores do domínio de Herbrand.

Por exemplo, para o programa PAI o domínio de Herbrand é `tare, abraao, nacor, isaac, sara`. Como este domínio é finito, podemos criar todas as instâncias para as cláusulas `fem/1` e `irmao/2`. Seguem alguns exemplos de instâncias:

```
fem(sara) <- mae(sara,isaac)
fem(abraao) <- mae(abraao, isaac)
fem(tare)  <- mae(tare, isaac)
...
irmao(abraao, nacor)<- pai(tare,abraao), pai(tare,nacor),abraao\==nacor
irmao(sara, tare)<- pai(nacor, sara), pai(nacor, tare), sara\==tare
...
```

Neste processo criam-se várias instâncias de cláusulas que não são usadas na *computação* de uma dedução. De qualquer modo, elas não interferem no processo de refutação. Por exemplo, mesmo existindo uma instância `fem(abraao) <- mae(abraao, isaac)` ela só poderá ser efetivamente usada numa prova se temos a cláusula fato `mae(abraao, isaac)<-` , o que não é verdade. Este processo geral é bem ineficiente pois gera uma montanha de instâncias sem utilidade.

O **universo de Herbrand** é o menor conjunto que contém todas as cláusulas fechadas que podem ser criadas para um programa. O programa PAI possui um universo de Herbrand finito. Quando o domínio de Herbrand é finito, o universo de Herbrand também é finito. Por outro lado, é comum termos domínios de Herbrand infinitos. Vejamos o programa abaixo.

```
1 %% programa NATURAL (só positivos) %%
2 natural(0)<-
3 natural(s(N))<- natural(N)
```

Deste programa podemos deduzir a resposta para a pergunta *O 2 é natural?*, `<-natural (s(s(0)))`. Este programa gera os números naturais a partir de um esquema indutivo:

- *0 é um natural;*
- *se N é um natural, então o sucessor de N, s(N) também é um natural.*

A lógica de predicados permite também o uso de símbolos funcionais, como parâmetros de predicados. Um **símbolo funcional** representa uma função que, quando avaliada, resulta em um indivíduo[3] do universo do discurso. Como exemplos de símbolos funcionais temos: o `sucessor(5)` que resulta em 6; 1+3 que resulta em 4; `idade(jose)` que resulta em 35.

Existe uma diferença entre um símbolo predicativo e um símbolo funcional:

[3]Um indivíduo é um objeto qualquer do discurso: pessoas, constantes, números, etc.

- o símbolo funcional representa um indivíduo e é uma oração incompleta (sem verbo);
- o símbolo predicativo representa uma assertiva ou relação e é uma oração completa (com verbo).

Por exemplo, o símbolo funcional `idade(jose)` é lido como: *A idade de José*. O símbolo predicativo (ou predicado apenas) `idade(jose,23)` é lido como: *A idade de José é 23*.

No programa NATURAL, temos o símbolo funcional "s"e a constante "0". Quando temos um símbolo funcional, o domínio de Herbrand compreende todas as possíveis instâncias do símbolo funcional aplicado às constantes e/ou aos símbolos funcionais. Para o programa NATURAL o domínio de Herbrand é { 0, s(0),s(s(0)), s(s(s(0))),... }, que corresponde ao conjunto infinito dos naturais positivos {0, 1, 2, 3, ...}.

Neste caso, no programa NATURAL, para eliminarmos a cláusula com a variável N, teremos um número infinito de cláusulas. Assim, para o programa NATURAL, o domínio e o universo de Herbrand são infinitos. Segue um esboço das instâncias das cláusulas do universo de Herbrand:

```
natural(s(0)) <- natural(0)
natural(s(s(0))) <- natural(s(s(0)))
natural(s(s(s(0)))) <- natural(s(s(0)))
...
```

Não podemos escrever na memória de um computador um novo programa equivalente ao NATURAL com infinitas cláusulas fechadas. No caso deste programa, não apenas o universo de Herbrand é infinito, mas também o seu significado (que é tudo o que pode ser deduzido a partir dele – todo o conjunto dos naturais positivos).

O **teorema de Herbrand** é um resultado que soluciona o problema de se tratar infinitas instâncias de cláusulas: *Dado um programa e uma pergunta. Se a pergunta é uma dedução do programa num finito número de passos então ela pode ser deduzida de um subconjunto finito de instâncias das cláusulas do programa (do Universo de Herbrand).*

Este teorema diz que uma solução finita, quando existe, está num subconjunto finito de instâncias. Se pedirmos para o programa NATURAL gerar todos os naturais, o processo não terá uma solução finita. Em termos práticos, não podemos pegar todos os naturais, pois não temos como guardar um conjunto infinito. Sempre temos que trabalhar com um subconjunto finito. Às vezes queremos um objeto abstrato, como um tipo de dados que represente o conjunto todo, por exemplo, um tipo abstrato de dados chamado INTEGER – neste caso, ele é apenas uma representação potencial. Numa realização desta representação em uma linguagem de programação, impomos um limite finito: por exemplo, os naturais representados em 32 bits.

2.4 O procedimento de resolução

O teorema de Hebrand diz que uma solução finita está num subconjunto finito do universo de Herbrand. Ainda assim, temos o problema de saber como calcular este subconjunto finito.

A **resolução** generaliza o processo de refutação da lógica proposicional permitindo predicados parametrizáveis. Com a resolução, as cláusulas de um programa são manipuladas por um processo de unificação (associado à regra de eliminação) que gera de modo sistemático e controlado as novas instâncias de cláusulas. Cada uma destas instâncias é sempre resolvida com a pergunta mais recente. Assim, só são geradas as instâncias úteis para provar a pergunta.

Segue um exemplo: provar por resolução a pergunta `<- natural(s(s(0)))`, a partir do programa NATURAL.

```
1. natural(0) <-
2. natural(s(N)) <- natural(N)
3. <- natural(s(s(0)))  %% pergunta inicial
                        %% tenta 3x1 natural(0)=natural(s(s(0))) FALHA
                        %% tenta 3x2 natural(s(N))=natural(s(s(0))),
                        %% unifica N/s(0) e gera a instância 2'
2'. natural(s(s(0)))<-natural(s(0))
4. <- natural(s(0))     %% resolve 3x2'
                        %% tenta 4x1 natural(0)=natural(s(0)) FALHA
                        %% tenta 4x2 natural(s(N))=natural(s(0)),
                        %% unifica N/0 e gera 2''
2''. natural(s(0))<-natural(0)
5. <- natural(0)        %% resolve 4x2''
6. <-                   %% resolve 5x1
```

A **unificação** é um processo usado para encontrar uma substituição que gera uma instância de uma cláusula para um passo da resolução. No exemplo acima, quando tentamos resolver a pergunta (3) com a cláusula (2) temos `natural(s(N)) = natural(s(s(0))` que resulta na substituição `{N/s(0)}`. Com esta substituição é gerada a instância (2'). Para gerar uma instância aplica-se a substituição em todos os predicados da cláusula. A instância gerada (2') é resolvida com a pergunta (3) gerando a nova pergunta (4).

Um passo de resolução consiste em:

- escolher uma cláusula que sua cabeça unifica com um literal (por exemplo, o primeiro) da pergunta;
- calcular a substituição;
- usar a substituição para gerar a instância da cláusula escolhida;
- resolver a instância com a pergunta, usando a regra de eliminação.

Note que, de forma inteligente, foram criadas só duas instâncias de cláusulas para responder à pergunta `<-natural(s(s(0)))`.

Um exemplo mais complexo de unificação é: `predic(a, W, X, f(f(X))) = predic(Z, g(Y), g(Z), f(Y))` que resulta na substituição `{Z/a, W/g(f(g(a))), X/g(a), Y/f(g(a))}`, que aplicada ao mesmo tempo nos dois predicados produz o unificador: `predic( a, g(f(g(a))), g(a), f(f(g(a))) )`. Abaixo ilustramos, informalmente, o cálculo desta unificação:

```
predic(a , W     ,X      ,f(f(X)))
predic(Z , g(Y)  ,g(Z)   ,f( Y ) )
-------------------------------------
       Z/a,                              %% passo 1, 1ro parametro
predic(a , W     ,X      ,f(f(X)))
predic(a , g(Y)  ,g(a)   ,f( Y ) )
-------------------------------------
       Z/a, W/g(Y)                       %% passo 2
predic(a , g(Y)   ,X      ,f(f(X)))
predic(a , g(Y)   ,g(a)   ,f( Y ) )
-------------------------------------
      Z/a, W/g(Y) ,X/g(a),               %% passo 3
```

```
predic(a , g(Y)   ,g(a)   ,f(f(g(a))) )
predic(a , g(Y)   ,g(a)   ,f( Y )     )
-------------------------------------
       Z/a, W/g(Y) ,X/g(a), Y/f(g(a))     %% passo 4
predic(a , g(f(g(a))) ,g(a) ,f(f(g(a))) )
predic(a , g(f(g(a))) ,g(a) ,f(f(g(a))) )

       {Z/a, W/g(Y) ,X/g(a), Y/f(g(a)) }       %% unificador
     = {Z/a, W/g(f(g(a))) ,X/g(a), Y/f(g(a))}  %% unificador simplificado
```

Neste processo de unificação a cada passo unifica-se um dos parâmetros do predicado. No primeiro passo unificou-se o parâmetro {Z/a}. Logo em seguida, todas as ocorrências de Z nos predicados são substituídas por a. O resultado de uma unificação é uma substituição chamada de unificador. No final do processo, simplificamos o unificador aplicando as substituições do unificador nele mesmo.

Para efeito de generalização do processo de unificação definimos que a unificação de duas variáveis sempre acontece, renomeando-se uma delas. Por exemplo, `pred(X)=pred(Y)` unifica com a substituição {X/Y} E, também, definimos que a unificação de duas constantes iguais é válida resultando na substituição vazia {}, por exemplo, `pred(a)=pred(a)`. Por outro lado, a tentativa de unificação de duas constantes diferentes sempre resulta em erro, por exemplo, `pred(a)=pred(b)`.

Exercício 2.5 *Calcule a substituição resultante da unificação (unificador) para cada um dos termos que seguem.*

```
1. p(a,X) = p(Y,b)
2. p(X,X) = p(Y,Z)
3. p(t(X,t(X,b))) = p(t(a,Z))
4. p(X,f(Y)) = p(f(Y),X)
5. p(X,f(X)) = p(f(Z),f(Z))
```

2.4.1 Estratégias de resolução

Dada uma pergunta, por exemplo, `<-P1,P2,...,Pn`, existem diversas estratégias para se escolher o termo (ou literal) a ser resolvido no próximo passo: por exemplo, podemos escolher `P1`, ou `Pn`, ou outro `Pi` qualquer. O Prolog adota a estratégia de escolher sempre o Pi mais à esquerda, neste exemplo, o P1.

Por outro lado, em cada passo devemos também escolher uma cláusula que pode ser resolvida com a pergunta. Neste caso, o Prolog adota a estratégia de escolha compatível com a ordem de escrita, de cima para baixo.

Às vezes podemos gerar múltiplas respostas. Neste caso, a escolha de cláusulas alternativas ocorre de cima para baixo. Segue um exemplo, para a pergunta `<-pai(X,Y)` que tem duas respostas: `X=tare, Y=abraao` e `X=tare, Y=aran`.

```
1. pai(tare, abraao) <-
2. pai(tare, aran ) <-
3. mae(sara, isaac ) <-
4. fem(X) <- mae(X,Y)
5. irmao(X,Y) <- pai(P,X),pai(P,Y), X\==Y
```

```
6. <-pai(X,Y)
7. <-          % 6x1 X=tare, Y=abraao
8. <-pai(X,Y)
9. <-          % 6x2 X=tare, Y=aran
```

Em Prolog, esta estratégia de escolha do termo da pergunta e da cláusula a ser resolvida com a pergunta, da esquerda para a direita e de cima para baixo, torna o processo de resolução adequado para executar uma cláusula `C1<-P1,P2,..Pn` vista como um procedimento, como exemplificado abaixo:

```
1 procedure C1;
2     begin
3        P1;
4        P2;
5        ...
6        Pn;
7      end
```

Aqui nasceu a visão procedimental para a lógica de cláusulas definidas proposta por Kowalski.

2.5 Fórmulas lógicas x Cláusulas

Na lógica de cláusulas, cada cláusula é uma fórmula universalmente quantificada. Esta restrição, em alguns casos, dificulta a leitura da fórmula lógica, por exemplo `mae(sara, isaac)<-` pode ser lida em lógica como:
$Q1 \equiv \exists X \exists Y (X = sara \wedge Y = isaac \wedge mae(X, Y))$
$Q2 \equiv \forall X \forall Y (X = sara \wedge Y = isaac \rightarrow mae(X, Y))$
$Q3 \equiv true \rightarrow mae(sara, isaac)$

Estas três fórmulas são equivalentes quando assumimos o domínio $\{X = sara$ e $Y = isaac\}$, que são os valores definidos na cláusula.

Um raciocínio similar acontece na fórmula lógica para a cláusula que define o predicado `tio/2`, que fica mais clara com o quantificador existencial,
$Q4 \equiv \forall T \forall X \exists P(pai(P, X) \wedge irmao(P, T) \rightarrow tio(T, X))$
$Q5 \equiv \forall T \forall X \forall P(pai(P, X) \wedge irmao(P, T) \rightarrow tio(T, X))$

Estas duas versões $Q4,\ Q5$ são equivalentes por existir um só pai para um mesmo filho, no domínio de discurso.

2.5.1 Transformação de fórmulas em cláusulas

Todo programa em lógica de cláusulas é traduzido para um conjunto de fórmulas lógicas de primeira ordem, simplesmente acrescentado-se o quantificador universal em cada variável. Cabe a pergunta no sentido inverso: dado um conjunto de fórmulas, ele pode ser traduzido para um programa Prolog? A resposta é não completamente, só em parte.

Vimos na lógica de proposições como transformar um fórmula proposicional na forma normal conjuntiva. E sabemos que fórmulas universalmente quantificadas possuem representação como cláusulas. O problema está principalmente na remoção do quantificador existencial.

Dada uma fórmula lógica X, são válidas as equivalências:
$\neg \exists X p(X) \equiv \forall X \neg p(X)$ e
$\neg \forall X p(X) \equiv \exists X \neg p(X)$.

Porém, estas regras só podem ser usadas quando, no contexto, temos apenas um quantificador, o existencial ou o universal. Em muitas fórmulas os dois quantificadores acontecem ao mesmo tempo, são adjacentes: tentando eliminar um existencial introduzimos outro: $\neg\forall X\exists Y p(X,Y) \equiv \exists X\forall Y\neg p(X,Y)$

Existe um resultado que permite eliminar um quantificador existencial ($\exists$) em um escopo de um quantificador universal $\forall$. Dada uma fórmula, por exemplo, "$\forall X\exists Y\ maior(Y,X)$", podemos remover o $\exists$ introduzindo uma **função de Skolem**: "$\forall X\ maior(f(X),X)$". Aqui $Y = f(X)$ é uma função de Skolem. Semanticamente, se para cada X existe sempre um número maior que X, então podemos assumir que ele é calculado por uma função de X, por exemplo, $f(X) = X + 1$.

Casos mais complexos acontecem num contexto com diversas ocorrências de variáveis, por exemplo: "$\forall X\forall Y\exists Z\ predicado(Z,Y,X)$". Neste caso a fórmula sem o existencial é "$\forall X\forall Y predicado(f(Y,X),Y,X)$".

Concluindo, pode-se traduzir fórmulas para cláusulas, mas na presença de quantificadores existenciais são introduzidas funções "estranhas"ao domínio do problema, chamadas de funções de Skolem.

2.5.2 Transformações para cláusulas

Um programa é um conjunto de cláusulas, por exemplo, $P = \{C1, .., Cn\}$. Este conjunto de cláusulas equivale a uma conjunção lógica, na forma $C1 \wedge C2 \wedge ...Cn$. Sabemos ainda que cada cláusula é uma condicional, na forma `cabeca<-corpo`.

Seja um programa formado por duas cláusulas com mesma cabeça {`a<-b; a<-c`}, podemos reescrevê-lo num programa equivalente com uma só cláusula {`a<-b v c`}, ver exercícios. O contrário também vale. Se temos uma disjunção no corpo de uma cláusula, podemos separá-la em duas.

A cabeça de uma cláusula definida pode ter apenas um predicado e este deve ser positivo. Logo, os exemplos abaixo não são de cláusulas definidas:

1. `nao chove <- faz_sol.`
2. `preto v branco <- nao colorido.`

A primeira cláusula é indefinida porque a conclusão é negativa. Se transpomos os literais, ela continua sendo negativa: `nao faz_sol <- chove`. Daí, uma solução pode ser a rescrita para uma cláusula "semanticamente"similar `faz_sol <- nao chove`; é apenas similar, pois, quando não chove, não necessariamente está fazendo sol, poderia estar apenas nublado.

A segunda cláusula é indefinida, temos uma disjunção na cabeça. Para torná-la definida é suficiente transpor um dos termos com o sinal trocado `preto <- nao branco & nao colorido`.

Exercício 2.6 *Prove que* {`a<-b, a<-c`} *é equivalente a* {`a<-b v c`}.

Dica: Como duas cláusulas são ligadas com `&`, basta provar que: `(av~b)&(av~c)=av~(bvc)`.

Exercício 2.7 *De forma similar, o que acontece quando unimos duas cláusulas com o mesmo corpo, por exemplo* {`a<-x, b<-x`}.

Solução: Examinando { `nao trabalho<-chove, nao faz_sol<-chove` } e também `(av~x)&(bv~x)=(a&b)v~x`, conclui-se que equivale a uma cláusula `(a&b)<-x`.

Exercício 2.8 *Transforme as cláusulas abaixo em cláusulas definidas.*

1. `nao chove<- faz_sol`

2. `chove v faz_sol <- true`

3. `fat(X,F) v nao fat(X-1,F1) v  nao fat(F=F1*X)) <- true`

4. `homem(X) v mulher(X) <- adulto(X)`

Exercício 2.9 *Defina, de forma clara e concisa, os termos que seguem:*

1. Lógica de proposições.

2. Operador lógico condicional (`<-`).

3. Lógica de predicados.

4. Lógica de primeira ordem.

5. Fórmula lógica e cláusula.

6. Lógica de cláusulas, Lógica de cláusulas indefinidas e Lógica de cláusulas definidas.

7. Regra, fato e pergunta (em termos de cláusulas).

8. Refutação.

9. Substituição, Resolução e Unificação.

Exercício 2.10 *Defina os termos* universo de Herbrand *e* domínio de Herbrand. *Então enuncie o* Teorema de Herbrand.

Exercício 2.11 *No processo de resolução, o Prolog adota uma estratégia específica para escolha das cláusulas a serem unificadas no corpo da pergunta, de cima para baixo e da esquerda para à direita. Qual a importância disso para o sucesso do Prolog?*

2.6 Conclusão

Vimos informalmente os principais conceitos da teoria do Prolog, que é o processo de resolução com unificação. Outro objetivo deste capítulo foi mostrar intuitivamente a relação entre os sistemas lógicos e a lógica subjacente ao Prolog.

A lógica de cláusulas definidas, apesar das limitações na representação de certas fórmulas da lógica de primeira ordem ou da lógica de cláusulas não definidas, tem um grande poder expressivo quando comparada às linguagens de computação convencionais. A restrição de permitir somente um predicado positivo como cabeça de uma cláusula é que possibilita um processamento eficiente das cláusulas no processo de resolução.

Representação de conhecimento vs. cláusulas

Um sistema de Inteligência Artificial (IA) Simbólica se caracteriza por possuir uma base de conhecimento, similar a base de regras e fatos de um programa Prolog. Independente do sistema de IA Simbólica, todas as linguagens utilizadas para representar conhecimento se classificam nas seguintes categorias (de menor expressividade para maior expressividade):

1. Lógica de proposições (ou de zero ordem):
   `P & Q -> R; calor & ~chuva -> ir_praia;`

2. Lógica de atributos-valores:
   `tempo=quente & chuva=não -> ação=ir_praia;`
   `tempo(quente), chuva(não) -> ação(ir_praia);`

3. Lógica de cláusulas definidas:
   `irmao(X,Y) <- pai(P,X),pai(P,Y), X\==Y;`

4. Lógica de predicados (ou de primeira ordem):
   $\forall X \forall Y \forall P(pai(P,X) \wedge pai(P,Y) \wedge X \neq Y) \rightarrow irmao(X,Y)$;

5. Lógica de ordem superior (ou de segunda ordem):
   $\forall P(\forall X \forall Y(P(ana,X) \wedge P(ana,Y) \wedge X \neq Y) \rightarrow irmao(X,Y))$.

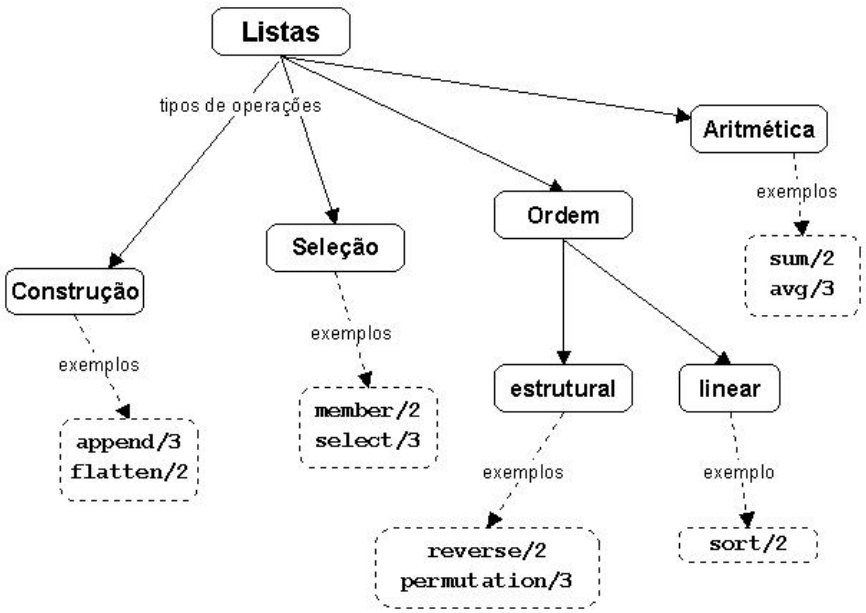
De um modo geral todas estas regras são cláusulas definidas. Na prática, o que varia de uma linguagem para outra é a forma sintática, por exemplo, usa-se muito sintaxe de regras `if-then` ou `if-then-else`. O Prolog trabalha com cláusulas definidas que é um subconjunto da lógica de primeira ordem. Na lógica de primeira ordem as variáveis devem ter como valores apenas indivíduos do domínio, por exemplo, as pessoas da família bíblica. Já na lógica de segunda ordem, as variáveis podem ter como valores nomes de predicados, por exemplo, a variável `P` acima; isto equivale a passar predicados como parâmetros na linguagem Prolog, o que também é permitido apesar de não ter eficiência na execução (pois a cláusula não pode ser compilada).

2.6.1 Transformações para cláusulas

O material deste capítulo pode ser complementado e aprofundado, em livros tais como [4] e [13]. O primeiro livro [4] detalha, em centenas de páginas, todos os temas relacionados com a teoria da Programação em Lógica (PL), que aqui são sintetizados em uma dezena de páginas. O segundo livro [13] é um texto de PL conciso e rico em exemplos e exercícios.

CAPÍTULO 3

Programação com Listas



Este capítulo é básico e essencial para qualquer curso de Prolog. Ele apresenta um estudo detalhado sobre programação em lógica pura, com o tipo de dados lista, que é fundamental para o Prolog. Algumas técnicas básicas de programação são introduzidas no decorrer do capítulo, entre elas, programação recursiva, predicados reversíveis, simplificar regras e técnica de acumuladores.

Neste capítulo, examinamos a definição de diversos predicados que **são predefinidos** em muitos sistemas Prolog, como o SWI-Prolog. Entre outros temos os predicados `append`, `member`, `select`. No momento de programá-los usamos um sufixo numérico ou um nome em português, por exemplo, `append1`, `membro`, `select1`, pois a tentativa de redefinir um predicado causa um erro do tipo *No permission to modify static procedure 'select/3'*. No final do capítulo apresentamos a relação de predicados para manipulação de listas que estão predefinidos no SWI-Prolog.

3.1 Operações sobre o tipo lista

O tipo de dados lista é uma das estruturas fundamentais para a PL (e também em linguagens funcionais). Na árvore do início do capítulo, esboçamos uma classificação para as operações (métodos) sobre o tipo lista, compreendendo quatro principais classes: *construção, seleção, ordem e aritmética*.

Na classe *construção* temos os construtores primitivos `[]`, `[ | ]` e outras operações que têm listas como saída, entre elas o famoso `append`, que concatena duas listas. Na classe de *seleção* temos os predicados que selecionam elementos ou sublistas de listas. Uma das operações de seleção mais usadas é o `member`. A classe de *ordem* é subdividida em duas. A subclasse *estrutural* compreende operações como o `reverse`, que inverte uma lista. Já na subclasse *linear*, temos as operações que ordenam os elementos de uma lista seguindo critérios como a ordem aritmética ou lexicográfica. Por fim, a classe das operações *aritméticas* com listas compreende algum tipo de cálculo sobre os valores das listas, por exemplo, contagem, somatório, produto, etc. Vale lembrar que muitas operações envolvem mais de uma classe.

A organização deste capítulo segue esta estrutura de classes.

3.2 O tipo de dados lista

Em Prolog, listas são estruturas com representação interna recursiva. Uma lista é definida por dois construtores:

- o construtor `[]`, que representa a lista vazia;
- o construtor `[X|Xs]`, que representa uma lista com o elemento `X` na cabeça e com a lista `Xs` como cauda.

Estes são os construtores básicos para recursivamente se construir listas de qualquer comprimento. Por exemplo, `[a|[b|[c|[]]]]` é uma lista com os elementos `a,b,c`. Para evitar esse excesso de colchetes aninhados, melhorando a leitura e escrita, o Prolog admite algumas simplificações, como segue:

```
%%Sintaxe basica    %%Sintaxe simplificada
  []                  []
  [a| []]             [a]
  [a|[b| []]]         [a,b]
  [a|[b|[c| []]]]     [a,b,c]
  [a|Xs]              [a|Xs]
  [a|[b|Xs]]          [a,b|Xs]
```

Listas são manipuladas pela unificação. Por exemplo, para remover a cabeça da lista `[a,b,c]` basta unificá-la com uma lista que tenha uma variável na cabeça:

```
?- [X|Xs]=[a,b,c].
   X=a, Xs=[b,c] Yes
```

Vale notar que a cabeça de uma lista é sempre um elemento, já a cauda é uma estrutura de lista. Assim, para manipular os elementos de uma lista devemos sempre trabalhar na cabeça da lista. Podemos pegar vários elementos da cabeça. Abaixo, por exemplo, pegam-se os três primeiros elementos de uma lista.

```
?- [X,Y,Z|_] = [1,3,5,7,9,11,13,15].
    X=1, Y=3, Z=5 Yes
?- [A,B,C|_]=[1,[a,b],f(5),4,f(bb),[6,t],7].
    A=1, B=[a,b], C=f(5) Yes
```

Aqui, também vemos que uma lista pode conter elementos dos mais variados tipos, como inteiros, letras, átomos, outras listas e functores.

Estas operações de pegar vários elementos de uma lista são baseadas na operação primitiva, que pega apenas um, por exemplo, abaixo pegamos os dois primeiros elementos em `X` e `Y`, usando apenas a construção primitiva de lista `[ | ]`:

```
?- L=[1,3,5,7],L=[X|Xs],Xs=[Y|_].
    X=1, Y=3, L=[1,3,5,7], Xs = [3,5,7] Yes
?- L=[1,3,5,7],L=[X|[Y|_]].
    X=1, Y=3, L=[1,3,5,7]  Yes
```

3.3 Programação recursiva

A seguir, veremos uma versão lógica para o predicado `compr`. A versão aritmética[1] deste predicado é apresentada no capítulo sobre aritmética. No Prolog, este predicado é predefinido como `lenght/2`, como no teste que segue.

```
?- length([a,b],L).
   L = 2
```

Para estudo de algoritmos e estruturas de dados devemos inicialmente desenvolver versões lógicas dos programas em Programação em Lógica (PL) pura. Estas versões posteriormente podem ser relaxadas, buscando um efeito mais procedural (ou funcional), por exemplo, substituindo-se expressões aritméticas por valores calculados.

A definição de operações sobre listas faz referência à recursividade da estrutura de dados. Assim, o predicado comprimento `compr/2` é definido por duas regras, uma para a lista com cabeça e cauda e a outra para a lista vazia. A primeira é chamada regra recursiva e a segunda regra base, que define quando termina a recursividade.

- **regra recursiva:** o comprimento da lista `[X|Xs]` é 1 mais o comprimento da cauda `Xs`;
- **regra base:** o comprimento da cauda `[]` é 0.

Esta especificação é escrita como um predicado Prolog definido por duas regras:

```
1 compr(L,T):-L=[X|Xs], compr(Xs,T1),T=1+T1.
2 compr(L,T):-L=[], T=0.
```

Uma lição básica de programação é **saber simplificar**. A idéia é simplificar as regras de um predicado, com o objetivo de torná-lo mais econômico, de leitura mais fácil. Neste sentido, PL é uma boa linguagem, pois ela define formalmente o conceito de substituição que é usado no

[1] Retorna um valor calculado.

processo de simplificação. Dada uma regra, podemos encontrar instâncias mais simples e com a mesma semântica.

Vamos examinar o exemplo do predicado `compr/2`. A primeira dificuldade na programação de um predicado é encontrar uma versão inicial, que funcione como esperado, isto é, que seja uma especificação correta para o problema. A versão inicial do `compr/2` é correta. Então podemos perguntar: Existe um predicado equivalente e de leitura mais simples?

Examinemos como simplificar a primeira das duas regras em quatro passos:

1. Seja a regra inicial
   ```
   compr(L,T):-L=[X|Xs], compr(Xs,T1), T=1+T1.
   ```

2. Aplicamos a substituição `{L/[X|Xs]}` para gerar uma nova instância da regra
   ```
   compr([X|Xs],T):-[X|Xs]=[X|Xs], compr(Xs,T1), T=1+T1.
   compr([X|Xs],T):- true, compr(Xs,T1), T=1+T1.
   compr([X|Xs],T):- compr(Xs,T1), T=1+T1.
   ```

3. De forma similar com a substituição `{T/1+T1}` resulta em:
   ```
   compr([X|Xs],1+T1):- compr(Xs,T1).
   ```
   Agora, temos só a variável T1 e não temos T – sendo que uma variável como C[2] é mais simples que T1, substituímos `{T1/C}`:
   ```
   compr([X|Xs],1+C):- compr(Xs,C).
   ```

4. Por fim, quando numa regra uma variável tem ocorrência única, podemos torná-la anônima, que é o caso da variável `X`: `{_/X}`
   ```
   compr([_|Xs],1+C):- compr(Xs,C).
   ```

Exercício 3.1 *No passo 4, por que dizemos que C é mais simples que T1? Justifique.*

Para entendermos o efeito dessa simplificação, fazemos uma leitura, para cada um dos passos, partindo da regra inicial:

1. *o compr de uma lista L é T se L é igual a uma lista com cabeça X e cauda Xs, e o compr de Xs é T1, e T é igual a 1 mais T1.*

2. *o compr da lista com cabeça X e cauda Xs é T se o compr da cauda é T1, e T é igual T1 mais 1.*

3. *o compr da lista com cabeça X e cauda Xs é 1 mais C se o compr da cauda é C.*

4. *o compr da lista com cauda Xs é 1 mais C se o compr da cauda é C.*

Em cada um destes passos obtivemos uma leitura mais simples, com menos palavras. Simplificar um predicado é sinônimo de torná-lo mais **silencioso**, porém mantendo o seu significado.

Segue a versão final do predicado, no qual a cláusula `compr(L,T):- L=[],T=0` foi rescrita sem o uso das variáveis.

```
1 compr([_|Xs],1+T):-compr(Xs,T).
2 compr([],0).
```

[2] Usamos C denotando comprimento, mas poderia ser qualquer letra, inclusive a letra T.

```
?- compr([a,B,c],T).
 T=1+1+1+0
?- compr([],X).
 X=0
```

Na verdade, para este predicado existe ainda uma leitura mais silenciosa, que é: *o comprimento da lista com cauda Xs é 1 mais o comprimento da cauda*. Esta leitura corresponde a uma versão funcional da regra: `compr([_|Xs]) -> 1+compr(Xs)` que não pode ser programada como um predicado lógico.

Para simplificar regras, evitando dar nomes a indivíduos que não se relacionam com ninguém, são usadas variáveis anônimas, que têm significado próprio, por exemplo, `?- [A,A]=[3,X].  A=X=3` porém em `?- [_,_]=[3,X],X=4.  X=4 Yes`. No primeiro caso o A é usado para passar o valor 3 para X, porque, se temos dois As, ambos assumem o mesmo valor; com variáveis anônimas isto não acontece, cada ocorrência é uma instância distinta.

Exercício 3.2 *Simplifique a cláusula abaixo:*

`pega3(E,L):-L=[B|A],A=[Y|Ys],Ys=[Z|D],Z=E`

3.3.1 Usando um acumulador

Muitos problemas de programação recursiva podem ser resolvidos por diferentes estratégias. Quando o resultado de uma computação é calculado passo a passo (por exemplo, somar ou contar uma unidade), podemos usar a técnica do acumulador.

Esta técnica simula uma computação numa linguagem procedural. Por exemplo, um programa para calcular o comprimento de uma lista L em Pascal pode ser:

```
1 acc:=0;
2 while not eh_vazia(L)
3     begin
4        L := cauda(L);
5        acc := acc+1;
6     end
7 resultado := acc;
```

O papel do acumulador `acc` é armazenar o resultado intermediário durante o cálculo. Quando termina o laço de processamento, a variável `resultado` recebe o valor do acumulador.

Abaixo, temos uma versão Prolog do predicado *comprimento* usando um acumulador. Definimos um predicado auxiliar `compr3`, com um parâmetro a mais, para o acumulador. O predicado principal chama o auxiliar, inicializando o acumulador com 0 (equivale à linha antes do `while`). Na regra base do predicado auxiliar `compr3`, o valor do acumulador é passado para predicado principal (equivale a regra após `while`).

```
1 comprAcc(L,C):-compr3(L,0,C).
2 compr3([X|Xs],Acc,C):-compr3(Xs,Acc+1,C).
3 compr3([],Acc,Acc).
```

Para estudo, comparamos os dois predicados *comprimento* em Prolog. A principal diferença está na estratégia de como é calculado o valor — sem acumulador vs. com acumulador: (1) sem acumulador, o valor é calculado de trás para frente — mais interno para externo; (2) com acumulador, ao contrário — do mais externo para o interno. O trace mostra em cada um dos passos da execução o que entra `call` e o que sai `exit`.

```
?-trace(compr,[call,exit]).
%         compr/2: [call, exit] Yes
[debug]  ?- compr([a,b,c],X).
 T Call: (6) compr([a, b, c], _G367)
  T Call: (7) compr([b, c], _G422)
   T Call: (8) compr([c], _G425)
    T Call: (9) compr([], _G428)
    T Exit: (9) compr([], 0)
   T Exit: (8) compr([c], 0+1)
  T Exit: (7) compr([b, c], 0+1+1)
 T Exit: (6) compr([a, b, c], 0+1+1+1)
X = 0+1+1+1
%
?-trace([compr3,comprAcc],[call,exit]).
%         compr3/3: [call, exit]
%         comprAcc/2: [call, exit]  Yes
 [debug]  ?- comprAcc([a,b,c],X).
 T Call: (7) comprAcc([a, b, c], _G391)
 T Call: (8) compr3([a, b, c], 0, _G391)
 T Call: (9) compr3([b, c], 0+1, _G391)
 T Call: (10) compr3([c], 0+1+1, _G391)
 T Call: (11) compr3([], 0+1+1+1, _G391)
 T Exit: (11) compr3([], 0+1+1+1, 0+1+1+1)
 T Exit: (10) compr3([c], 0+1+1, 0+1+1+1)
 T Exit: (9) compr3([b, c], 0+1, 0+1+1+1)
 T Exit: (8) compr3([a, b, c], 0, 0+1+1+1)
 T Exit: (7) comprAcc([a, b, c], 0+1+1+1)
X = 0+1+1+1
```

No cenário acima usamos o comando `trace/2` para entrar no modo debug, controlando a exibição só para as portas `call` e `exit`.

Exercício 3.3 *Usando trace, compare o predicado* `compr/2` *com o* `compr1/2` *abaixo, que é o* `compr` *com as linhas trocadas de ordem. O que acontece na execução? Qual executa mais passos?*

```
1 compr1([],0).
2 compr1([_|Xs],1+T) :- compr1(Xs,T).
```

Exercício 3.4 *Compare a especificação do programa* `compr/2` *com o* `sum/2` *abaixo. Rode* `sum/2` *para listas de inteiros. O que difere na especificação deles?*

```
1 sum([],0).
2 sum([X|Xs],X+S) :- sum(Xs,S).
```

3.4 Seleção em Listas

Nesta seção, apresentaremos o predicado `member/2`, que é predefinido nos sistemas Prolog. Normalmente, não podemos redefinir um predicado predefinido no sistema; podemos criar um outro com um novo nome, por exemplo, `member1` ou `membro`.

3.4.1 Membros de uma lista

O predicado `membro/2` verifica se um elemento é membro de uma lista. Usam-se duas regras:

- (base) um elemento é membro de uma lista, se ele é a cabeça da lista;
- (recursiva) um elemento é membro de uma lista, se ele for membro da cauda da lista.

Seguem estas duas regras codificadas e já simplificadas em Prolog:

```
1  membro(X,[X|_]).
2  membro(X,[_|Xs]):-membro(X,Xs).
```

```
?- membro(a,[b,c]).
   No.
?- membro(b,[a,b,c]).
   Yes
?- membro(a,X).
   X=[a] Yes
?- L=[a,B,c],membro(x,L).
   L=[a,x,c], B=x   Yes
?- membro(X,Y).
   X=_, Y=[X|_]; Yes
```

Como vemos, `membro/2` é reversível. Se os dois parâmetros são termos fechados (preenchidos com constantes), o predicado funciona como um teste. Se nos parâmetros existem variáveis, o Prolog tentará inferir o valor delas. Este predicado tem vários usos:

- testar se um elemento está na lista;
- incluir um elemento na lista, quando a lista é uma variável ou quando a lista contém uma variável;
- retornar um elemento da lista (na pergunta, o elemento é uma variável).

Relacionado com este último uso, vale reforçar que ele retorna todos os elementos de um conjunto, um de cada vez, por retrocesso:

```
?- membro(X,[a,b,c]).
X = a ;
X = b ;
X = c ;
No
```

Exercício 3.5 *Rode o trace para* `membro(X,[a,b,c])`*?.*

Segue uma versão mais simples, chamada membro1/2, que somente testa se um elemento é membro de uma lista. Este predicado é definido por duas regras mutuamente exclusivas. A primeira para X==Y e a segunda negando a condição da primeira.

```
1    membro1(X,[Y| _]):- X ==Y.
2    membro1(X,[Y|Xs]):- X\==Y,membro1(X,Xs).
```

```
?- membro1(b,[a,b,c]).
   Yes
?- membro1(a,X).
   No
?- L=[a,B,c],membro1(x,L).
   No
```

3.4.2 Explorando o significado de um predicado

A diferença entre membro e membro1/2 está no significado dos operadores de igualdade == e unificação =. Apresentamos uma técnica para se conhecer mais sobre operadores e predicados.

Um sistema Prolog, por ser um interpretador de comandos, nos oferece a facilidade de explorar, de imediato e interativamente, uma operação ou estrutura de dados. Submetendo-se pequenas perguntas e analisando-se as respostas,... reiterando com pequenas variações, até descobrir o uso de uma construção. Este processo pode e deve ser usado com os predicados que são novos para o programador — ou, às vezes, também para relembrar um significado de um predicado já conhecido. Veja alguns exemplos:

```
?- 1==1.
   yes
?- 'a'==a.
   yes
?- a==B.
   No
?- a\==B.
   Yes
?- p(a,b)=p(a,X).
   X = b  Yes.
?- p(b)=p(c,D).
   No
?- p(b)\=p(c,D).
   D=_  Yes
...
?-p(X)=p(X+1).
 X = ... +... +1+1+1+1+1+1+1+1+1 (SWI Prolog)
 Yes
?- p(X)==p(X+1).
  No
```

É importante diferenciar a unificação = da igualdade ==. A unificação de dois termos os torna iguais, se casam. Uma vez casados é como um casamento sem separação pelo resto da vida útil das variáveis, na instância da cláusula[3]. Na verdade, diversas variáveis podem casar com um único valor, vejamos:

[3]Uma mesma variável em diferentes instâncias pode assumir diferentes valores, ver capítulo 2.

```
?- A=B, B=Z, C=A, C=5.
   A=B=C=Z=5   Yes
```

O teste de igualdade simplesmente verifica se dois termos são iguais, e cada um mantém sua individualidade.

3.5 Construção de listas

Uma das operações sobre listas mais usadas é a concatenação, também conhecida como append/3, no Prolog. Esta operação concatena duas listas dadas em uma terceira:

- (base) concatenar uma lista vazia [] com uma lista Bs. O resultado é a lista Bs;
- (recursiva) concatenar a lista [A|As] com a lista Bs resulta na lista [A|AsBs], sendo AsBs uma lista formada pela concatenação de As com Bs.

Em Prolog, esta especificação resulta num compacto e flexível predicado, como mostra a execução abaixo. Se damos dois parâmetros, é devolvido o terceiro. Chamamos de append1, pois o append já é predefinido no Prolog.

```
1 append1([],Bs,Bs).
2 append1([A|As],Bs,[A|AsBs]):- append1(As,Bs,AsBs).
```

```
?- append1([a,b],[c,d],X).
   X=[a,b,c,d].
?- append1(X,[c,d],[a,b,c,d]).
   X=[a,b]
?-append1([a,b],X,[a,b,c,d]).
   X=[c,d].
```

O append é um dos predicados de manipulação de listas mais útil e reusado para definir novos predicados. O predicado member/2, por exemplo, pode ser definido a partir dele (ver abaixo, membro/2). X é membro de uma lista L, se está no meio da lista. Outros dois predicados similares ao member são prefix e sufix que devolvem todos os prefixos ou sufixos de uma lista. Finalmente, temos o predicado sublist/2 que testa se uma lista é sublista de outra e/ou, também, gera todas as sublistas.

Aqui nestes exemplos, usamos o append predefinido no Prolog.

```
1 membro(X,L):-append(_,[X|_],L).
2 prefix(P,L):-append(P,_,L).
3 suffix(S,L):-append(_,S,L).
4 sublist(S,L):-prefix(P,L),suffix(S,P).
5
```

```
?- prefix(P,[a,b,c]).
P = [] ;
P = [a] ;
P = [a, b] ;
```

```
P = [a, b, c]
?- suffix(S,[a,b,c]).
S = [a, b, c] ;
S = [b, c] ;
S = [c] ;
S = []
?- sublist(L,[a,b]),L\=[].
L = [a] ;
L = [a, b] ;
L = [b]
```

Antes de mudarmos de assunto, temos outra operação parente do append que é `select/3`. Na verdade, ela tem a mesma estrutura recursiva de `member/2`, com a diferença que, dada uma lista, ela remove um elemento desta, devolvendo-a sem o elemento. Definimos um `select1`, pois o `select` é predefinido no Prolog.

```
1 select1(X,[X|Xs],Xs).
2 select1(X,[Y|Xs],[Y|Zs]):-select1(X,Xs,Zs).
```

```
?- L=[a,b,c,d], select1(b,L,Lo).
L = [a,b,c,d] ,  Lo = [a,c,d] Yes

?- L=[a,b,c,d], select1(X,L,Lo).
L = [a,b,c,d] , X = a ,Lo = [b,c,d] ;
L = [a,b,c,d] , X = b ,Lo = [a,c,d] ;
L = [a,b,c,d] , X = c ,Lo = [a,b,d] ;
L = [a,b,c,d] , X = d ,Lo = [a,b,c] ; no
```

Este predicado, por retrocesso, seleciona sistematicamente cada um dos elementos de uma lista, partindo do primeiro ao último.

Exercício 3.6 *Defina o predicado* `ultimo/2`*, que devolve o último elemento de uma lista.*

Solução:

```
1 ultimo(L,U):-L=[X|Xs],Xs=[],U=X.
2 ultimo(L,U):-L=[X|Xs],Xs\=[],ultimo(Xs,U).
3 %%
4 ultimo([U],U).                    %% simplificado
5 ultimo([X|Xs],U):-ultimo(Xs,U).
```

E, usando append/3: `ultimo(L,U):-append(_,[U],L).`

Exercício 3.7 *Escreva em linguagem natural as regras que definem o predicado último. Uma regra base e uma regra recursiva.*

Exercício 3.8 *Escreva em linguagem natural à leitura do predicado último, programado com* append.

Solução: O último elemento U de uma lista L é calculado como: *faça a lista L ser a concatenação de duas listas, sendo a segunda de apenas um elemento U, então U é o último elemento.*

Exercício 3.9 *Defina um predicado* `remU/2` *que devolve uma lista sem o último elemento; com e sem o uso do* `append`.

Solução:

```
remUa(L,Lu):-append(Lu,[_],L).
%%
remU(L,Lu):-L=[X|Xs],Xs=[],Lu=[].
remU(L,Lu):-L=[X|Xs],Xs\=[],Lu=[X|Xu],remU(Xs,Xu).
%%
remU([_],[]).                    %% simplificado
remU([X|Xs],[X|Xu]):-remU(Xs,Xu).
```

Exercício 3.10 *Escreva em linguagem natural as regras que definem o predicado* remove último.*Para o predicado que usa* `append` *e para o predicado recursivo (regra base e recursiva).*

Exercício 3.11 *Defina um predicado* `contiguos/1`, *que testa se uma lista tem dois elementos contíguos iguais. Com e sem o uso do* `append/3`. *Por exemplo* `?-contiguos ([a,b,c,c,e]). Yes`.

Exercício 3.12 *Escreva em linguagem natural uma leitura para cada uma das versões do predicado* `contiguos/1`.

Exercício 3.13 *Defina um predicado* `dupl/1`, *que é verdadeiro, se a lista tem elementos duplicados. Use o* `member`.

Exercício 3.14 *Escreva em linguagem natural uma leitura para o predicado* `dupl/1`.

Exercício 3.15 *Defina um predicado* `trocaPU/2` *que devolve uma lista em que o primeiro e último elementos são trocados de posição. Use* `append`.

Exercício 3.16 *Defina um predicado* `remDupl/2` *que remove os elementos duplicados de uma lista, use apenas o* `member` *e recursividade. Remova as cópias iniciais dos elementos da lista. Assim,* `?-remDupl([a,b,a],X). X=[b,a]. Yes`

Exercício 3.17 *Escreva em linguagem natural uma leitura para o predicado que remove os elementos duplicados.*

Exercício 3.18 *Defina um predicado* `remDupl/2` *que remove os elementos duplicados de uma lista, use apenas o* `member` *e o* `select/3`. *Remova as cópias finais dos elementos da lista. Assim,* `?-remDupl([a,b,a],X). X=[a,b]. Yes`

Exercício 3.19 *Defina o predicado* `flatten/2` *que devolve uma lista simples a partir de uma lista formada por listas aninhadas, mantendo a ordem de ocorrência dos elementos. Por exemplo,*

```
flatten1([1,[2],[],[2,[3,4],5],6],X).
 X = [1, 2, 2, 3, 4, 5, 6]
```

Solução:

```
1 flatten1([X|Xs],F):-flatten1(X,F1),flatten1(Xs,F2),append(F1,F2,F).
2 flatten1([],   []).
3 flatten1(X,   [X]):-X\=[], X\=[_|_].
```

Exercício 3.20 *Escreva em linguagem natural uma leitura para o predicado* `flatten`.

3.6 Trabalhando com ordem

Entre os elementos de uma lista, existem dois tipos de ordem. A ordem aritmética ou alfanumérica, usada para ordenar nomes ou valores numéricos. Este tipo de ordem leva em conta o valor ou o conteúdo de cada elemento. Por outro lado, a ordem estrutural dependente da posição física dos elementos na lista e não do valor de cada elemento. Este tipo de ordem é usado em problemas de permutação, por exemplo, permutar os elementos de uma lista. Abaixo, mostra-se a execução do predicado `permutation/2`, gerando as permutações da lista `[a,b,c]`.

```
?- permutation([a,b,c],P).
P = [a,b,c] ;
P = [a,c,b] ;
P = [b,a,c] ;
P = [b,c,a] ;
P = [c,a,b] ;
P = [c,b,a] ; No
```

Este predicado, `permutation`, usa o predicado `select/3` para extrair cada elemento de uma lista. Assim sendo, permutar uma lista consiste em:

- (base) se a lista é `[]`, então retorna-se `[]`;
- (recursiva) senão, dada uma lista, extrai-se um de seus elementos usando `select/3`, coloca-se como primeiro elemento da solução e chama-se recursivamente o predicado permutação para a lista sem o elemento.

```
1 permutation(Xs,[Z|Zs]):-select(Z,Xs,Ys),permutation(Ys,Zs).
2 permutation([],[]).
```

Um predicado parente do `permutation/2` é o `reverse/2`, que inverte a ordem dos elementos de uma lista. Podemos programá-lo em duas versões: uma ingênua, baseada no `append/3` que é deixado como exercício; e outra usando um acumulador.

```
1 reverse(L,R) :- reverse(L,[],R).
2 reverse([], R,R).
3 reverse([X|Xs],ACC,R) :- reverse(Xs,[X|ACC],R).
```

```
?- reverse([a,b,c],R).
   R=[c,b,a]
```

Exercício 3.21 *Usando* `append/3`*, defina o* `reverse/2`*. Codifique as regras:*

- (base) o reverso de uma lista vazia é a lista vazia;
- (recursiva) o reverso de uma lista é o reverso da cauda concatenado com a cabeça.

Exercício 3.22 *Defina o predicado* `palindrome/1`*, que é verdadeiro se a lista é um palíndromo, por exemplo,* `[a,b,c,d,c,b,a]`*. Faça duas versões, uma usando* `reverse` *e uma recursiva usando* `append`*.*

Exercício 3.23 *Defina um predicado* `metIguais/1`*, que é verdadeiro se uma lista é formada por duas metades iguais. Use o* `append`*. Seguem dois exemplos de uso.*

```
?-metIguais([a,b,c, a,b,c]).
   Yes
?-metIguais([a,b,c, a,b,d]).
   No
```

3.6.1 Ordem alfanumérica

A ordem alfanumérica é usada para ordenar números e cadeias de caracteres, por exemplo, ordenar nomes na ordem alfabética. O predicado `isOrdered/1` é verdadeiro se uma lista de inteiros está em ordem crescente.

```
1 isOrdered([]).
2 isOrdered([_]).
3 isOrdered([X,Y|XYs]):-X=<Y,isOrdered([Y|XYs]).
```

```
?- isOrdered([1,5,5,11]).
Yes
?- isOrdered([2,1]).
No
```

Exercício 3.24 *Sem usar acumulador, defina um predicado* `maxL(M,L)` *que retorna o valor máximo de uma lista de valores numéricos? (seleção e ordem)*

Solução:

```
1    maxL([X],X).
2    maxL([X|Xs],X):-maxL(Xs,M),X >M.
3    maxL([X|Xs],M):-maxL(Xs,M),X=<M.
4 %%
5    maxL([X],X).                                  %% ou usando ;
6    maxL([X|Xs],M):-maxL(Xs,M1),(X>M1,M=X; X=<M1,M=M1).
```

Exercício 3.25 *Usando um acumulador, defina um predicado* `maxL(M,L)`. *Inicie o acumulador com o valor da cabeça da lista de entrada.*

Exercício 3.26 *Usando* `permutation` *e* `isOrdered` *defina um algoritmo de ordenação* `sortx/2`, *tal que:*

```
?-sortx([1,11,5,2], X).
  X=[1, 2, 5, 11] Yes
```

3.7 Listas para teoria dos conjuntos

Muitos problemas de especificação de software envolvem conjuntos finitos, por exemplo, o conjunto das vogais. Um conjunto finito pode ser modelado por uma lista, sem valores repetidos. Neste contexto, o predicado `member/2` equivale à operação `pertence/2` (se um elemento pertence a um conjunto).

Outra operação básica é `subconjunto/2`. A operação `sublist/2` não pode ser usada para implementar a operação subconjunto, porque uma sublista é uma seqüência contígua de elementos.

Segue a definição de `pertence/2`, `subconjunto/2` e `interseção/3`.

```
1 pertence(X,C):-member(X,C).
2 subConjunto([X|Xs],Y):- select(X,Y,Ys), subConjunto(Xs,Ys).
3 subConjunto([],Y).
4 intersecao([],X,[]).
5 intersecao([X|Xs],Y,[X|Is]):-    member(X,Y),  intersecao(Xs,Y,Is).
6 intersecao([X|Xs],Y,    Is):- \+ member(X,Y),  intersecao(Xs,Y,Is).
```

```
?-subConjunto(X,[a,b]).
X = [a, b] ;
X = [a] ;
X = [b, a] ;
X = [b] ;
X = []
intersecao([a,b,c],[c,a,d,b],Y).
Y = [a, b, c]
```

Existem modos mais eficientes para se escrever vários dos predicados apresentados neste capítulo. Por exemplo, o predicado `intersecao` pode ser melhorado com o uso do operador de corte, a ser estudado nos próximos capítulos. O objetivo deste capítulo é enfatizar a PL pura e não a eficiência de predicados escritos em Prolog.

Exercício 3.27 *Qual o problema do programa abaixo se comparado com a versão apresentada com o* `select/3`*?*

```
subConjunto([],Y).
subConjunto([X|Xs],Y):- pertence(X,Y), subconjunto(X,Y).
```

Exercício 3.28 *Defina o predicado* `uniao/3`, *de forma que no resultado não haja elementos repetidos. Faça duas soluções, uma usando* `remDupl/2` *e a outra, recursiva, usando* `member/2` *(similar ao predicado* `intersecao/3`*).*

Exercício 3.29 *Escreva um predicado* pertence3/3 *em que o terceiro parâmetro é true ou false. Como segue.*

```
?- pertence(a, [c,a,b],X). X=true Yes
```

Exercício 3.30 *Rescreva* interseção/3 *sem usar a negação de* member/1*, fazendo uso do predicado* pertence3/3 *e uma disjunção (;). A idéia é computar o pertence uma única vez e usar um* flag *do resultado para escolha da cláusula alternativa.*

Exercício 3.31 *Faça um predicado* insOrd/3*, que insere um elemento numa lista mantendo–a ordenada. Faça duas regras: uma base e uma recursiva.*

```
?-insOrd(4,[2,3,5,7],L).
  L=[2,3,4,5,7] Yes
```

Exercício 3.32 *Faça o predicado de ordenação pelo método inserção direta,* insDir/2*. Use o predicado* insOrd/3*. Enquanto a lista dada de entrada não for vazia, pegue a cabeça e insira na lista ordenada, que é um acumulador (inicializado como lista vazia).*

Exercício 3.33 *Faça um predicado que* particiona/3 *uma lista em duas, de tamanho igual se o número de elementos for par, senão uma delas terá um elemento a mais. Tire dois elementos de uma lista (se possível) e ponha cada um em uma lista resultado.*

```
?- particiona([a,b,c,1,2,3,4],A,B).
   A=[a,c,2,4],
   B=[b,1,3]  Yes
```

Exercício 3.34 *Faça o predicado* merge/3*, que junta duas listas ordenadas em uma terceira, mantendo a ordem. Como segue:*

```
?- merge([a,b,b,k,z], [c,m,n,o], X).
   X=[a,b,b,c,k,,m,n,o,z], yes
```

Exercício 3.35 *Faça um predicado de ordenação* mergeSort/2*. Use o* particiona/3 *e o* merge/3*. Dada uma lista, particiona-se a lista em duas, chama-se o mergeSort para ordenar cada partição e com o merge juntamos os dois sub-resultados no resultado final. Considere a condição de parada, uma lista vazia ou com um elemento.*

```
?-mergeSort([1,11,5,2], X).
  X=[1, 2, 5, 11] Yes
```

Exercício 3.36 *Faça um predicado* zipper/3 *tal que:*

```
?- zipper([a,b,c,d], [1,2,3,4], X).
   X=[a-1, b-2, c-3, d-4], yes
```

Exercício 3.37 *Faça um predicado* enesimo/3 *tal que:*

```
?- enesimo(3,[a,b,c,d], X).
   X=c, yes
```

Exercício 3.38 *Faça um predicado* remove/3 *tal que:*

```
?- remove(a,[a,b,a,d,x,a], X).
   X=[b,d,x] yes
```

3.8 Refatoração: Otimização de predicados

Neste capítulo vimos que existem diferentes técnicas para se fazer uma mesma solução, por exemplo, com e sem acumulador. Agora vamos explorar a técnica de refatoração (reescrever um código que já está funcionando buscando clareza, reuso ou performance). São discutidas duas técnicas: performance linear e recursão na cauda.

Refatoração buscando performance linear. Um predicado para manipular listas deve ter, se possível, uma performance linear em relação ao comprimento da lista. Vamos mostrar como testar e indicar o caminho para obter esta performance linear.

```
1  maxL0([X],X).
2  maxL0([X|Xs],X):-maxL0(Xs,M),X >M,!.
3  maxL0([X|Xs],M):-maxL0(Xs,M),X=<M,!.
4  %%
5  maxL00([X],X).
6  maxL00([X|Xs],M):-maxL00(Xs,M1),(X>M1,M=X,!; X=<M1,M=M1).
7  %%
8  maxL([X|Xs],M):-maxL(Xs,X,M).
9  maxL([],M,M).
10 maxL([X|Xs],ACC,M):- ACC>X,!,maxL(Xs,ACC,M).
11 maxL([X|Xs],_,M):- maxL(Xs,X,M).
12 %%
13 gL(L,N):-!,findall(X,(between(1,N,I),X is random(1000)),L).
```

Aqui temos três versões para o predicado que retorna o valor máximo de uma lista. E temos um predicado que gera uma lista de valores randômicos de tamanho N, para testar as versões do predicado; este predicado gL é explicado só no final desta seção.

Uma unidade de medida de tempo de predicados é o número de inferências: corresponde a um CALL ou REDO no modelo execução da máquina abstrata do Prolog. O número de inferências é uma medida de tempo, independente de cpu, complementar ao tempo em segundos da cpu. Associada a esta medida temos a unidade chamada Lips (logical inferences per second), que define a velocidade do Prolog numa dada máquina, dividindo o número de inferências pelo tempo em segundos.

A primeira versão, `maxL0`, tem um comportamento exponencial: vai crescendo exponencialmente em relação ao comprimento da lista. Pois para uma lista de 10 elementos executa com 416 inferências; para 20 salta para 467.004; para 30 dispara para 126 milhões; com os tempos de 0.0seg, 0.14seg e 39.69 segundos. Projetando estes valores num gráfico temos uma curva exponencial. Um predicado com comportamento exponencial não serve, deve ser reescrito, refatorado.

A segunda versão, `maxL00`, já tem um comportamento linear em relação ao tamanho da lista. Em média de 4,9 inferências por elemento. E a terceira, `maxL` que usa um acumulador tem o melhor tempo, com uma média de 2,0 inferências por elemento.

```
?- gL(L,10), time(maxL0(L,M)).
   416 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
?- gL(L,20), time(maxL0(L,M)).
   467,004 inferences, 0.13 CPU in 0.14 seconds (88% CPU, 3736032 Lips)
?- gL(L,30), time(maxL0(L,M)).
   126,353,408 inferences, 34.97 CPU in 39.69 seconds (88% CPU, 3613324 Lips)
%%
```

```
?- gL(L,100), time(maxL00(L,M)).
   490 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
?- gL(L,10000), time(maxL00(L,M)).
   49,990 inferences, 0.02 CPU in 0.02 seconds (98% CPU, 3199360 Lips)
%%
?- gL(L,100), time(maxL(L,M)).
   204 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
?- gL(L,10000), time(maxL(L,M)).
   20,022 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
```

Recursão na cauda. Normalmente a versão de um programa com acumulador pode ser recursiva na cauda. Isto é, a cabeça da cláusula é chamada no final do corpo, daquela mesma cláusula; a chamada é a última parte do corpo. Das três versões acima só o `maxL` é recursivo na cauda.

Um predicado recursivo na cauda utiliza menos recursos de memória para executar, tendo conseqüências positivas na redução do tempo. Veja abaixo os tempos em segundos quando aumentamos o tamanho da lista para 100 mil e depois para 1 milhão de elementos. Para 100 mil o `maxL` recursivo na cauda levou 0.5 segundos, enquanto que o `maxL0` não recursivo na cauda levou 0.15 segundo. Já, para 1 milhão, o `maxL` levou 0.50 segundos, porém o `maxL0` não conseguiu rodar pois estourou a pilha de execução.

```
?- gL(L,100000), time(maxL(L,M)).
% 200,106 inferences, 0.05 CPU in 0.05 seconds (94% CPU, 4268928 Lips)
gL(L,1000000), time(maxL(L,M)).
% 2,000,996 inferences, 0.50 CPU in 0.50 seconds (100% CPU, 4001992 Lips)
%%
?- gL(L,100000), time(maxL00(L,M)).
% 499,986 inferences, 0.14 CPU in 0.15 seconds (93% CPU, 3555456 Lips)
?- gL(L,1000000), time(maxL0(L,M)).
ERROR: Out of local stack
```

Em resumo um predicado recursivo na cauda consome menos recursos e é mais robusto. Portanto, sempre que um predicado é executado um grande número de vezes ou que faz parte de uma biblioteca importante, ele deve ser reescrito com recursão na cauda e deve passar no teste de performance linear.

Mais sobre inferências. As inferências são contabilizadas somente para os predicados que são executados na máquina abstrata do Prolog. Portanto, quando estamos chamando um predicado de uma biblioteca em C, o número de inferências não representará toda a realidade. Segue um código que é usado para ilustrar estes pontos.

```
1 rsort(X,S):-sort(X,S1),reverse(S1,S).
2 rmsort(X,S):-msort(X,S1),reverse(S1,S).
3 %%
4 front(N,L,Lo):-length(Lo,N),append(Lo,_,L).
5  last(N,L,Lo):-length(Lo,N),append(_,Lo,L).
```

No SWI-Prolog utilizamos muito dois predicados de ordenação de listas: `sort` e `msort` Ambos ordenam uma lista na ordem crescente, o primeiro remove os elementos duplicados e o segundo não remove. Acima codificamos duas versões que retornam o resultado em ordem decrescente, simplesmente invertendo a lista. Veja as execuções abaixo.

```
?- sort([a,a,a,b,c,d,d,d,d],S).
   S = [a, b, c, d]
?- rsort([a,a,a,b,c,d,d,d,d],S).
   S = [d, c, b, a]
?- msort([a,a,a,b,c,d,d,d,d],S).
   S = [a, a, a, b, c, d, d, d, d]
?- rmsort([a,a,a,b,c,d,d,d,d],S).
   S = [d, d, d, d, c, b, a, a, a]
```

Os predicados `sort` e `msort` são implementados numa biblioteca em C, portanto quando chamamos eles o número de inferências é apenas uma. Já o `rsort` usa o `reverse` da biblioteca do Prolog, portanto as inferências são contadas: uma inferência para cada elemento de uma lista. O mesmo vale para o `msort`.

```
?- gL(L,100000), time(sort(L,M)).
   1 inferences, 0.08 CPU in 0.08 seconds (98% CPU, 13 Lips)
?- gL(L,100000), time(rsort(L,M)),length(M,LM).
   1,004 inferences, 0.06 CPU in 0.06 seconds (99% CPU, 16064 Lips)
   LM = 1000
%%
?- gL(L,100000), time(msort(L,M)),length(M,LM).
   1 inferences, 0.25 CPU in 0.25 seconds (100% CPU, 4 Lips)
   L = [399, 335, 758, 172, 194, 503, 747, 234, 915|...]
   M = [0, 0, 0, 0, 0, 0, 0, 0, 0|...]
   LM = 100000
?- gL(L,100000), time(rmsort(L,M)),length(M,LM).
   100,004 inferences, 0.45 CPU in 0.45 seconds (100% CPU, 220698 Lips)
   L = [259, 560, 928, 397, 887, 348, 373, 527, 191|...]
   M = [999, 999, 999, 999, 999, 999, 999, 999, 999|...]
   LM = 100000
```

As perguntas abaixo mostram a execução do `front` e `last` que pegam respectivamente N elementos iniciais ou finais de uma lista. Estes predicados usam o `append`. Vemos que pegar elementos na frente da lista é bem mais eficiente. Logo para otimizar programas devemos sempre que possível trabalhar a partir do início de uma lista.

```
?- gL(L,10000),time(front(1000,L,Lo)).
   1,004 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
?- gL(L,10000),time(last(1000,L,Lo)).
   9,004 inferences, 0.45 CPU in 0.47 seconds (97% CPU, 19871 Lips)
```

Por fim temos o predicado gL que depende dos predicados `findall` e `between`. Abaixo temos o predicado `forall( between(1,5,I), ...)` que, com dois parâmetros, imita um predicado `for` de uma linguagem procedural. O `findall(EXP,PRED,LISTA)` é similar ao `forall`, porém tem três parâmetros; o primeiro é uma EXPressão que é coletada em LISTA a cada vez que o PREDicado é executado. O `between(INIC,FIM,I)` retorna por retrocesso todos os valores da faixa declarada. Seguem alguns testes.

```
?- between(1,3,X).
   X = 1 ; X = 2 ; X = 3 ; No
?- forall(between(1,5,I),(write(I*I),write(' '))).
   1*1 2*2 3*3 4*4 5*5
?- findall(Y*Y,(between(1,5,Y)),L).
   L = [1*1, 2*2, 3*3, 4*4, 5*5]
```

O funcionamento interno destes predicados é apresentado no capítulo sobre estruturas de controle. Além disso, outras variantes do `findall` são apresentadas no capítulo sobre estruturas de dados.

3.9 Predicados para listas do SWI-Prolog

Existe uma convenção para definir a passagem de parâmetros em protótipos de predicados em Prolog. Por exemplo, em `reverse(+List1, -List2)`, o (+) indica parâmetro de entrada e o (−) parâmetro de saída.

```
?- reverse([a,b],X).
   X = [b, a] ; No
?- reverse(X,[a,b]).
   X = [b, a] ;
   ERROR: Out of global stack
?- reverse(X,Y).
   X = [], Y = [] ;
   X = [_G228], Y = [_G228] ;
   X = [_G228, _G234], Y = [_G234, _G228] ;
...
```

Como vemos acima, podemos usar o predicado indevidamente e mesmo assim em muitos casos ele funciona. O uso correto, pela especificação do protótipo, é o da primeira pergunta. Outra anotação de passagem de parâmetros é o (?), que define que o parâmetro pode ser tanto de entrada como de saída, como `length(?List, ?Int)`.

```
?- length([a,b],T).
   T = 2 Yes
?- length(L,2).
   L = [_G216, _G219] Yes
?- length(L,I).
   L = [], I = 0 ;
   L = [_G231], I = 1 ;
...
```

Segue a relação dos protótipos dos predicados de manipulação de listas, predefinidos no SWI-Prolog:

- `is_list(+Term)` retorna sucesso, se Term é uma lista, vazia ou com elementos. Se Term for uma variável, o predicado falha. Este teste é feito na primeira cláusula do predicado, que é definido abaixo.

```
1    is_list(X) :-var(X),!,fail.
2    is_list([]).
3    is_list([_|T]) :-is_list(T).
```

- `memberchk(?Elem, +List)` é similar ao `member`, mas retorna somente uma solução. List deve ser dada como entrada.
- `delete(+List1, ?Elem, ?List2)` remove todas as ocorrências do elemento Elem de List1 retornando List2. List1 deve ser dada como entrada.
- `nth0(?Index, ?List, ?Elem)` retorna o enésimo elemento de uma lista. A contagem começa no 0.
- `nth1(?Index, ?List, ?Elem)` idem, a contagem começa no 1.
- os predicados abaixo, são os mesmos estudados neste capítulo:
  `append(?List1, ?List2, ?List3)`
  `member(?Elem, ?List)`
  `select(?Elem, ?List, ?Rest)`
  `last(?Elem, ?List) %% ultimo`
  `reverse(+List1, -List2)`
  `flatten(+List1, -List2)`
  `length(?List, ?Int) %% compr`
  `sumlist(?List, ?Int) %% soma`
  `merge(+List1, +List2, -List3)`

Ainda existe o predicado `maplist` que mapeia um predicado aritmético, no exemplo, `plus/3`, para todos os elementos de uma lista, retornando uma nova lista. Segue um teste.

```
?- plus(4,5,X).
   X =9
?- maplist(plus(2),[1,2,3],L).
   L = [3, 4, 5]
```

O funcionamento interno deste predicado é apresentado no capítulo sobre estruturas de controle.

3.10 *Programação procedural vs lógica (Opcional)

Esta seção é destinada a programadores experientes em programação imperativa, por exemplo, Pascal e C++.

Aqui fazemos um paralelo entre a programação procedural e lógica, aprofundando a discussão sobre a codificação de predicados usando a técnica dos acumuladores. Abaixo temos a versão em Prolog do programa que calcula o comprimento de uma lista, com e sem acumulador.

```
1 % sem acumulador
2 compr([],0).
3 compr([X|Xs],C):-compr(Xs,C1), C is C1+1.
4
```

```
% com acumulador
comprAcc(L,C):-compr3(L,0,C).
compr3([X|Xs],Acc,C):-Acc1 is Acc+1, compr3(Xs,Acc1,C).
compr3([],Acc,Acc).
```

Abaixo, codificamos estes predicados como procedimentos em Pascal. Inicialmente definimos um tipo `Lista` e as funções `cauda` e `eh_vazia` para podermos manipular a lista. Em seguida, temos três versões destes predicados. A primeira é para calcular o comprimento sem usar um acumulador, que é bem próxima da versão em Prolog, a diferença está no açúcar sintático que é necessário no Pascal. A segunda corresponde ao predicado com acumulador

mas sem recursividade. Aqui a variável `acc` é local ao procedimento, e é atualizada a cada iteração. A iteração é executada por um `while`; após o `while` retornamos o valor acumulado. A terceira é programada de forma recursiva. Até no Pascal, ao usarmos recursividade, temos que passar o acumulador como um parâmetro. Esta versão Pascal também é bem próxima do predicado Prolog, salvo pelo açúcar sintático.

```
Type  Lista = record cab:info; pcauda:pLista; end;
      pLista = pointer to Lista;
function cauda(L:pLista):pLista; begin cauda := L^.pcauda; end;
function eh_vazia(L:pLista):bool; begin eh_vazia := L=nil; end;
% sem acumulador recursiva
procedure compr(L:pLista; var resultC:int);
begin
  if   eh_vazia(L) then result:=0
  else begin compr(cauda(L),C1); resultC:=C1+1; end;
end;
% com while iterativa
procedure comprAccI(L:tLista; var result:int);
begin
acc:=0;
while not eh_vazia(L)
    begin  L := cauda(L); acc := acc+1; end
result := acc;
end;
% com acumulador recursiva
procedure comprAcc(L:tLista; var result:int);

begin  camprAcc0(L,0,result); end;
procedure comprAcc0(L:tLista; acc:int; var result:int);
begin
  if   eh_vazia(L) then result:=acc
  else begin acc1 := acc+1; comprAcc0(cauda(L),acc1, result); end;
end;
```

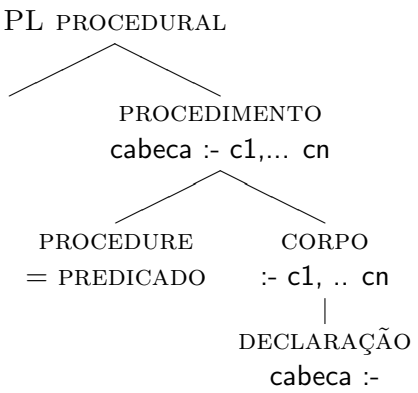
Podemos agora resumir as diferenças e similaridades entre os códigos Prolog e Pascal, em várias dimensões:

- **predicado vs. procedure:** cada predicado é definido como um procedimento imperativo;
- **cláusulas vs.** `if`**:** cada cláusula de um predicado corresponde a um caminho de um comando condicional (`if`);

- **parâmetros vs. acumuladores:** num programa iterativo, a variável que acumula um resultado corresponde a um parâmetro num predicado Prolog (ou numa procedure recursiva);
- **itens de dados:** listas em Prolog são estruturas de dados com ponteiros em Pascal; variáveis simples e constantes são os mesmos objetos em ambas as linguagens;
- **seqüência, seleção e repetição:** seqüências de comandos em Pascal são separadas por (;), em Prolog por (,); um comando condicional (`if`) é codificado em Prolog por diferentes cláusulas de um mesmo predicado; as repetições (`while`) são codificadas por recursividade;
- **atribuição vs unificação:** A atribuição `result:=acc` em Pascal corresponde a uma combinação de passagem de parâmetros e unificação em Prolog: a cláusula `compr3([],Acc,Acc)` pode ser rescrita como `compr3([],Acc,Result) :- Result = Acc`.

CAPÍTULO 4

Programando o fluxo de controle



Este capítulo apresenta os principais predicados para programar o fluxo de controle de um procedimento. O entendimento completo destes predicados é necessário para se ter o domínio da programação em Prolog.

Sugerimos especial atenção para os tópicos:

- O uso do operador de corte;
- Negação por falha;
- Programação recursiva com entrada e saída.

4.1 Interpretação procedimental para cláusulas

A interpretação procedimental para as cláusulas de Horn foi desenvolvida por Kowalski [12]. Ele mostrou que uma cláusula (ou uma implicação lógica) pode ser lida como um procedimento. Por exemplo, a cláusula `A if B1 and B2 and ... Bn` é lida e executada como um procedimento em uma linguagem de programação recursiva, em que `A` é a cabeça do procedimento e os `Bi(s)` são o seu corpo. Assim, o programa[1].

```
prog if ler(Dado) and calcular(Dado, Result) and imprimir(Result)
```

é lido como: para executar `prog` executa-se `ler(Dado)`, então executa-se `calcular(Dado, Result)` e então executa-se `imprimir(Result)`. Em Prolog este programa pode ser escrito como:

```
1  prog :- ler(Dado),
2          calcular(Dado, Result),
3          imprimir(Result).
4  calcular(Dado,Quadr):- Quadr is Dado*Dado.
5       ler(Dado      ):- write('Digite um valor:'), read(Dado).
6  imprimir(Result    ):- write('  o quadrado eh:'), write(Result),nl.
```

```
?- prog.
   Digite um valor: 4 <enter>
     o quadrado eh: 16
```

Esta interpretação de cláusulas como procedimentos é consistente com processo de resolução, em que a unificação é quem faz a manipulação dos dados: atribuição de valores, construção de estruturas de dados, seleção de dados e passagem de parâmetros. Porém, programas pensados e desenvolvidos sob esta interpretação perdem duas das características de programas em PL puros:

- mudar a ordem dos predicados do corpo da cláusula sem alterar o significado do procedimento;
- reversibilidade: dar uma entrada e receber a resposta; e dar a resposta e receber a entrada.

Em PL pura podemos trocar a ordem seqüencial dos predicados no corpo de uma cláusula, por exemplo, os dois predicados abaixo são equivalentes:

```
tio1(T,X):-irmao(T,P),pai(P,X).
tio2(T,X):-pai(P,X),irmao(T,P).
```

O mesmo não é válido para programas procedurais. O programa `prog`, acima exemplificado, executa uma seqüência correta de chamadas a subprocedimentos como `ler, calcular` e `imprimir`. Por outro lado, `prog2`, abaixo, não pode calcular um resultado de um `Dado` que ainda não foi lido.

```
prog2 :- calcular(Dado, Result),
         ler(Dado),
         imprimir(Result).
```

[1] Ou unidade de programa, que pode ser um procedimento ou um predicado, dependendo do paradigma de programação.

4.2 Predicados de controle

O Prolog possui vários predicados de controle, explorados principalmente em programas procedurais.

- Predicados `fail/0`, `true/0` e `repeat/0`
- Operador de corte `!/0`
- Comandos condicionais `If-Then` e `If-Then-Else (->/2) (;/2)`
- Negação por falha `\+/1`
- Metapredicados `call/1` e `once/1`

4.2.1 Os predicados fail/0, true/0 e repeat/0

O predicado `fail` é usado para forçar o retrocesso. Por exemplo, no programa abaixo queremos listar todos os participantes da relação `pai/2`.

```
1 pai(tare, abraao).
2 pai(tare, nacor).
3 pai(aran, lot).
4 escreve :- pai(X,Y), write(X),write(' eh pai de '),write(Y),nl,fail.
5 escreve1:- pai(X,Y), write(X),write(' eh pai de '),write(Y),nl,fail.
6 escreve1:- true.
```

Neste programa o predicado `escreve/0` chama `pai(X,Y)`, que retorna os valores para `X` e `Y` que são escritos; após a escrita, o predicado `fail` força o retrocesso. Por retrocesso, este processo se repete, para cada um dos fatos `pai/2`. Quando `pai(X,Y)` for falso, o predicado `escreve` termina com No.

Aqui, o `escreve` fez todo o trabalho e deveria então retornar verdadeiro. Na versão `escreve1/0` consertamos este problema, acrescentando uma cláusula com `true`, assim sempre termina com Yes. Esta cláusula é também equivalente ao fato `escreve1`.

Os predicados `true` e `fail` possuem significados opostos e complementares[2].

```
?- escreve.
tare eh pai de abraao
tare eh pai de nacor
aran eh pai de lot
NO
?- escreve1.
tare eh pai de abraao
tare eh pai de nacor
aran eh pai de lot
YES
```

[2]Na verdade, seria mais correto nomeá-los `true/false` ou `succeed/fail`.

O predicado repeat

O predicado `repeat` é usado para forçar uma repetição que termina só quando a cláusula como um todo é verdadeira. Por exemplo, `"?- repeat, get0(Char),Char=32,!"` repete a leitura no teclado até encontrar um branco (valor ASCII 32). Se `Char=32` falha, a execução retorna até o `repeat`, e então volta a reexecutar a leitura `get0/1`.

O predicado `repeat`, que já é predefinido no Prolog, é programado como segue:

```
1 repeat :- repeat.
2 repeat.
```

4.2.2 O operador de corte (!)

Os fatos e regras do Prolog têm o poder expressivo de relações, em que um objeto é relacionado com vários. Porém em muitos casos de computação queremos implementar funções determinísticas onde cada entrada é associada a uma única saída.

Um predicado é não determinístico quando gera múltiplas respostas. Quando gera uma só resposta, ele é determinístico. O operador de corte pode ser usado para tornar um predicado determinístico. Quando um predicado já é determinístico, ele pode ser otimizado com o uso do operador de corte.

- **O retrocesso sobre o corte termina a execução do predicado com** `fail`.

Abaixo, mostramos duas perguntas, a primeira, sem corte, gera todos os valores por retrocesso e a segunda, com corte, gera só o primeiro valor.

```
?-pai(X,Y), write(pai(X,Y)),nl,fail.
 pai(tare, abraao)
 pai(tare, nacor)
 pai(aran, lot)
No
?- pai(X,Y),!,fail.
  pai(tare, abraao)
 No
```

Basicamente a semântica do operador de corte é: *a execução pode passar por cima de mim* `(!)`, *mas não pode retroceder por cima de mim*. Uma tentativa de retrocesso sobre um corte instantaneamente termina a execução do predicado, com falha.

O corte controla o retrocesso. Em Prolog, o retrocesso acontece de dois modos:

- a execução retrocede, numa mesma cláusula, até o ponto em que encontra um predicado com várias cláusulas alternativas (ou disjunções);
- ou a execução retrocede toda a cláusula e tenta a cláusula seguinte, dentro de um mesmo predicado.

O primeiro caso foi ilustrado acima, em que a execução retorna até a chamada do predicado `pai(X,Y)` que é reexecutado, cada vez trazendo uma nova resposta. O segundo é ilustrado a seguir, em `max0`. Compare-o com `max`.

```
1 max(X,Y,M):- X>Y, M=X.
2 max(X,Y,M):- X<Y, M=Y.
3 max(X,Y,M):- X=Y, M=X.
4 %
5 max0(X,Y,M):- X>Y,!, M=X.
6 max0(X,Y,M):- X<Y,!, M=Y.
7 max0(X,Y,M):- X=Y,!, M=X.
8 %
9 max00(X,Y,M):- (X>Y,!,M=X; X<Y,!,M=Y; X=Y,!,M=X).
```

O efeito do corte sobre o código fonte do predicado pode ser visto como um comando procedural **gotoFIM**, que, na tentativa de retrocesso, envia o controle para uma ultima cláusula que falha, como segue

max0(X,Y,M):- X>Y,!**(gotoFIM←)**, M=X.
max0(X,Y,M):- X<Y,!**(gotoFIM←)**, M=Y.
max0(X,Y,M):- X=Y,!**(gotoFIM←)**, M=X.
gotoFIM: max0(_ ,_ ,_) :- fail.

Quando sabemos que um predicado retorna um único resultado, como o `max/3` acima, podemos incluir o corte depois da condição de teste de entrada na cláusula, visando otimizar a execução. Em `max0`, quando uma condição é satisfeita, o corte elimina todas as tentativas de execução das regras seguintes, otimizando-o. O significado do corte neste contexto é *se a execução retroceder por cima de mim* (`!`)*, encerre a execução do predicado*.

O escopo do corte, i.e., onde ele controla o fluxo de execução, é o predicado em que ele é definido. O predicado *pai* que o chamou não é afetado. O corte em `max0` não muda o significado do programa, apenas otimiza o seu fluxo de execução cortando regras que já sabemos que não trazem novos resultados.

O operador de corte na última linha do predicado `max0` não tem nenhum efeito: não corta outras possibilidades de retrocesso, já que não tem uma quarta regra. Mesmo assim, quando usamos corte em um predicado com várias regras é bom colocar o corte em todas elas numa mesma posição, como exemplificado em `max0` — o código fica mais padrão (mais reto — tudo igual).

Por fim, o `max00` tem o mesmo comportamento de `max0` porém está escrito como uma única cláusula com disjunções.

```
?- trace.
Yes
[trace]  ?- max(4,3,M).
   Call: (7) max(4, 3, _G329) ? creep
     Call: (8) 4>3 ? creep
     Exit: (8) 4>3 ? creep
   Exit: (7) max(4, 3, 4) ? creep
M = 4 ;  %% digite ;
   Redo: (7) max(4, 3, _G329) ? creep
     Call: (8) 4<3 ? creep
     Fail: (8) 4<3 ? creep
   Redo: (7) max(4, 3, _G329) ? creep
     Call: (8) 4=3 ? creep
     Fail: (8) 4=3 ? creep
```

```
   Fail: (7) max(4, 3, _G329) ? creep
No
[trace] ?- max0(4,3,M).
   Call: (7) max0(4, 3, _G335) ? creep
     Call: (8) 4>3 ? creep
     Exit: (8) 4>3 ? creep
     Call: (8) _G335=4 ? creep
     Exit: (8) 4=4 ? creep
   Exit: (7) max0(4, 3, 4) ? creep
M = 4 ;
   Fail: (7) max0(4, 3, _G335) ? creep
No
```

Como vemos, em `max` houve duas tentativas de reexecutar (redo) as cláusulas, enquanto em `max0` não houve tentativas de reexecutar cláusulas.

```
1 max1(X,Y,X):- X>Y,!.
2 max1(X,Y,Y).
3 %
4 max2(X,Y,X):- X>Y.
5 max2(X,Y,Y).
```

```
?- max1(3,2,M).
  M=3
?- max2(3,2,M).
  M=3 ;
  M=2
```

Outro uso do corte é ilustrado em `max1`. Aqui, o corte é usado para garantir que, quando `X>Y`, a execução da segunda regra não acontece. Portanto, sem o corte, como mostrado em `max2`, o predicado retorna dois resultados, sendo um deles incorreto.

O uso do corte é classificado como:

- *corte verde*, quando a sua remoção não afeta o significado do programa, por exemplo, em `max0`;
- *corte vermelho*, quando a remoção a sua remoção altera o comportamento do programa, por exemplo, em `max1`.

Em termos de execução, programas com corte vermelho geralmente são mais eficientes, por exemplo, em `max1` não é testada a condição para `Y>=X`. Mesmo assim, recomenda-se evitar, quando possível, o uso de corte vermelho.

Em muitos usos, o corte muda o significado lógico de um predicado. Por exemplo,

```
1 p:-a,b.      p1:-a,!,b.
2 P:-c.        p1:-c.
```

Aqui, `p` é igual a `(a&b)v c` enquanto `p1` é igual a `(a&b)v (~a&c)`. Isto é, em `p`, se `a` for verdadeiro e `b` falso, o `c` é executado; mas em `p1` o `c` é executado só quando `a` for falso.

Exercício 4.1 *O que está errado no programa abaixo? Rode-o com trace, para* ? `max(3,4,M)` *e* ?- `max(4,3,M)`.

```
max(X,Y,M):-!, X>Y, M=X.
max(X,Y,M):-!, X<=Y, M=Y.
```

Exercício 4.2 *O que acontece com o predicado* `p`*, abaixo, quando o* `b` *é executado?*
`a. b. p:-!,a. p:-b.`

Exercício 4.3 *Por que o corte é necessário em* `fat/1`*, abaixo? Teste-o com* ?- `fat(3,X), fail`. *Modifique-o como um corte do tipo verde.*

```
fat(X,1):-X<1,!.
fat(X,F):-fat(X-1,F1), F is F1*X.
```

4.2.3 If-then-else (_ −> _ ; _)

Segue mais uma versão para o máximo de dois números, `max5/3`, agora usando um comando condicional, em vez de duas regras alternativas. A forma geral do comando condicional é `Bool->Then;Else`. Por exemplo, `X>Y->M=X;M=Y` é lido como *se* `X>Y` *então* `M=X` *senão* `M=Y`. Esta versão é equivalente a uma versão com corte vermelho `max6/3`.

```
max5(X,Y,M):-( X>Y -> M=X; M=Y).
max6(X,Y,M):-( X>Y,!, M=X; M=Y).
```

Um comando condicional deve vir entre parênteses, pois o operador `(;)` tem maior precedência que o `(->)`. Pode-se usar somente `(Bool->Then)`, por exemplo,

```
?- (true->write(a)),write(b).
  ab  Yes
?- (fail->write(a)),write(b).
  No
```

Quando a condição falha, os predicados que seguem o `(Bool->Then)` são ignorados.

4.2.4 Negação por falha

O Prolog implementa uma forma simples de negação, conhecida como negação por falha. No Prolog ISO o símbolo de negação por falha é \+ que é lido como "not".

A negação do Prolog funciona como esperado, sempre que a pergunta é um termo fechado, por exemplo:

```
namora(jose,ana).
casal(joao,maria).
casal(pedro,antonia).
casado(X):-casal(X,_);casal(_,X).
solteiro(X):- \+ casado(X).
```

```
?- solteiro(joao).
   No
?- solteiro(jose).
   Yes
```

O Prolog assume a *hipótese do mundo fechado*, para responder `?- solteiro(jose)` ele chama `casado(jose)` que falha. Pela lógica, ele assume `\+ casado(jose) = \+ fail = true`. Assim, ele assume que todos os casados estão declarados como fatos no problema; e todos que não estão declarados casados não o são.

Existem dois problemas com a negação por falha do Prolog:

- Se perguntarmos sobre algum fato não especificado, ele assume como falso. Se perguntarmos `?- solteiro(abacaxi)` ele responde Sim, porque não tem a informação que abacaxi é fruta e não pessoa.
- Se perguntarmos `?- solteiro(X)` ele responde Não. Mas, logicamente, ele deveria responder `X=jose` e `X=ana` que são os enamorados que estão descritos no domínio deste problema.

```
?- solteiro(abacaxi).
   Yes
?- solteiro(X).
   No
```

A solução para estes dois problemas é a especificação de um predicado (como um tipo de dados) que gera todos os valores válidos para a variável `X` (no caso o conjunto das pessoas). Este predicado é usado antes de chamar a negação. Isto funciona porque **a negação responde de forma correta a perguntas fechadas**. Segue esta solução:

```
1 pessoa(X):-member(X,[jose,ana,joao,maria,pedro,antonia]).
2 namora(jose,ana).
3 casal(joao,maria).
4 casal(pedro,antonia).
5 casado(X):-casal(X,_);casal(_,X).
6 solteiro(X):- pessoa(X),\+ casado(X).
```

```
?- solteiro(X).
  X = jose ;
  X = ana ;
  No
```

Exercício 4.4 *Defina o predicado* `pessoa/1`*, para o programa acima, usando só fatos.*

Uma pergunta, quando falha, não instancia as variáveis. Seja `p/1` um predicado, `\+ p(X)` ou `p(X)` falha. Portanto, a pergunta `?- \+ p(X)` não instanciará a variável `X`, independente do resultado. Com esse fato podemos usar uma dupla negação para saber se um predicado é verdadeiro sem instanciar as suas variáveis.

```
1 p(a).
```

```
?- p(X).
  X=a Yes
?- \+ \+ p(X).
  X= _G232 Yes
```

O predicado \+, **que define a negação por falha**, que já é predefinido no Prolog, é definido como:

```
1 :-op(900, fy,  \+  ).
2 %
3 \+ X  :- call(X), !, fail.
4 \+ X  :- true.
```

4.2.5 Meta chamada e resposta única

O predicado `call/1` é usado em metaprogramação, quando numa variável é passada uma regra para ser executada. Por exemplo, o predicado `once/1` é definido como
`once(X):- call(X),!.`

Interpretadores mais flexíveis permitem escrever este predicado sem o `call/1` explícito:
`once(X):- X,!.`

O predicado `once/1` devolve a primeira resposta para uma pergunta com múltiplas respostas:

```
?-listing(p/1).
p(a).
p(b).
?- p(X).
  X = a ;
  X = b ; No
?- once(p(X)).
  X = a ; No
```

Existem no Prolog, também, predicados similares aos comandos de repetição de uma linguagem procedural: `forall`, `findall` e `between`. O `forall(CONTROLE, DO)` imita um comando for; o parâmetro `CONTROLE` indica quantas vezes ele deve repetir, o executando o segundo parâmetro. O `between(INIC,FIM,I)` retorna por retrocesso todos os valores da faixa declarada. A combinação `forall` e `between` imita um comando `for`.

```
?- between(1,3,X).
   X = 1 ; X = 2 ; X = 3 ; No
?- forall(member(X,[1,2,3]),write(X)).
   123
?- forall(between(1,5,I),(write(I*I),write(' '))).
   1*1 2*2 3*3 4*4 5*5

?- forall(between(10,20,I),write(I:' ')).
10: 11: 12: 13: 14: 15: 16: 17: 18: 19: 20:
```

O `findall(EXP,PRED,LISTA)` é similar ao `forall`, mas tem três parâmetros; o primeiro é uma EXPressão que é coletada em LISTA a cada vez que o PREDicado é executado. Ele é importante pois no Prolog listas é a estrutura de dados mais utilizada.

Outro predicado que executa uma repetição sobre uma lista é o `maplist`. Ele mapeia um predicado, no exemplo, `plus/3`, para todos os elementos de uma lista, retornando uma nova lista. Seguem alguns testes.

```
?- findall(Y*Y,(between(1,5,Y)),L).
   L = [1*1, 2*2, 3*3, 4*4, 5*5]
?- plus(4,5,X).
   X =9
?- maplist(plus(2),[1,2,3],L).
   L = [3, 4, 5]
```

Segue abaixo uma possível codificação para os predicados `between` e `forall`. Já a programação do `findall` é bem complexa, portanto, não será mostrada. O `forall` utiliza o `fail` para forçar a repetição até o predicado de controle também falhar. Mas no final tem mais uma cláusula para termina com true.

O `between` devolve por retrocesso todos os elementos do intervalo, e no final termina com fail. É um dos poucos predicados que queremos que termine com fail, pois ele deve abortar a execução do comando que está sendo repetido no `forall`.

Por fim, cabe dizer que estes predicados são independentes, isto é podem ser utilizados em conjunto ou não e podem ser combinados com quaisquer outros.

```
1 forall0(F,D):-F,D,fail.
2 forall0(_,_).
3 between0(N,M,_):-N>M,!,fail.
4 between0(N,M,N).
5 between0(N,M,I):-between0(N,M-1,Io),I is Io+1.
6 maplist0(F,[X|Xs],[Xo|Xso]):- F=..FL, append(FL,[X,Xo],FL1),F1=..FL1,F1,maplist0(F,Xs,Xso).
7 maplist0(_,[],[]).
```

O `maplist` também é simples, mas utiliza um comando que decompões e recompões o predicado. Por exemplo, `plus(2,3,Xo)`=.[plus,3,2,Xo].. Assim podemos acrescentar novos parâmetros para um predicado, antes de executá-lo.

```
?- X=2,F=plus(3),F=..FL, append(FL,[X,Xo],FL1),F1=..FL1.
   FL = [plus, 3]
   FL1 = [plus, 3, 2, Xo]
   F1 = plus(3, 2, Xo)
?- X=2,F=plus(3),F=..FL, append(FL,[X,Xo],FL1),F1=..FL1,F1.
   Xo = 5
   F1 = plus(3, 2, 5)
```

4.3 Recursividade com entrada e saída

Prolog possui três comandos básicos de entrada e saída: `read/1`, `write/1`, `nl/0`. Estes comandos são ilustrados abaixo.

```
?- read(A),read(B),write((A+B)/2),nl.
  33. <enter>
  13. <enter>
```

```
 (33+13)/2
?- write('Digite A:'),read(A),  \c
   write('Digite B:'),read(B),nl, write((A+B)/2). <enter>
 Digite A: 33. <enter>
 Digite B: 13. <enter>
   (33+13)/2
```

4.3.1 Mudando a ordem de predicados

Para exercitar a programação recursiva com entrada e saída, apresentamos uma série de exercícios que envolvem recursividade e escrita. O predicado a/0 conta no modo ascendente e escreve cada valor da conta em ordem crescente.

```
1 a:-a(0).
2 a(X):- X>10,!.
3 a(X):- write(X),write(' '), X1 is X+1,a(X1).
```

```
  ?-a.
   0 1 2 3 4 5 6 7 8 9 10
```

Se mudarmos de posição o write, mudaremos a ordem de escrita na execução do programa. O predicado b/0 conta em ordem crescente e escreve a conta em ordem decrescente.

```
1 b:-b(0).
2 b(X):- X>10,!.
3 b(X):- X1 is X+1, b(X1), write(X),write(' ').
```

```
  ?-b.
   10 9 8 7 6 5 4 3 2 1 0
```

Podemos juntar os dois, a() e b() num programa.

```
  ?-a(0),b(0).
   0 1 2 3 4 5 6 7 8 9 10 10 9 8 7 6 5 4 3 2 1 0
```

Exercício 4.5 *Faça um predicado que gere a pirâmide abaixo. Use o predicado* wN/1.

```
1 wN(0):-write(0),!.
2 wN(N):-write(N),N1 is N-1, wN(N1),write(N).
```

```
  ?- xxx(3).
   0
  101
 21012
3210123
```

Exercício 4.6 *Faça um programa que gere a figura abaixo.*

```
     ?- xxxi(3).
3210123
 21012
  101
   0
```

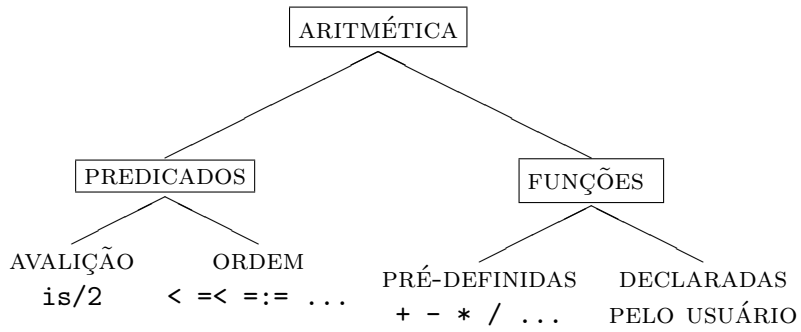
Exercício 4.7 *Faça um programa que gere a figura abaixo.*

```
     ?- xxxX(3).
   0
  101
 21012
3210123
 21012
  101
   0
```

Outros comandos de entrada e saída, em especial para arquivos, foram apresentados no capítulo sobre a sintaxe do Prolog.

CAPÍTULO 5

Programação aritmética



Este capítulo apresenta programas e exercícios sobre temas que envolvem a computação aritmética.

São exploradas as seguintes técnicas de programação: uso de acumuladores, mover a recursividade para a cauda, uso de diferença de listas e refinamento de programas: declarativo para procedural.

5.1 Avaliando expressões aritméticas

Em princípio, em Prolog uma expressão aritmética não é avaliada. Por exemplo, se perguntamos

```
?- X = 1+3.
   X = 1+3 Yes
```

o interpretador responde sem avaliar a expressão. Para o Prolog, a expressão 1+3 corresponde a um functor + com dois argumentos.

```
   ?- X = 1+3, write_canonical(X).
      +(1, 3)  Yes
   ?- X = 1+3, display(X).
      +(1, 3)  Yes
```

Quando precisamos um resultado de uma expressão, usamos o operador `is/2` que força a avaliação da expressão.

```
?- Y is 1+5.
    Y=6  Yes
?- X = 1+3*65/2, Z is X.
    X = 1+3*65/2, Z = 98.5 Yes
```

O predicado `is/2` exige que todas as variáveis do lado direito sejam fechadas (casadas com um valor). Por exemplo, à pergunta `?-X is A+3`, em que A é uma variável não fechada, o interpretador responde com uma mensagem de erro.

```
?- X is A+3.
  ERROR: Arguments are not sufficiently instantiated
```

No lado esquerdo da atribuição, podemos ter também constantes. Neste caso, o lado direito é avaliado e, se for igual ao lado esquerdo, o predicado é verdadeiro.

```
?- 3 is 3.
        Yes
?- X=3, X is X.
    X=3 Yes
?- 3 is 2+1.   Yes
?- A=2, 3 is A+1.
   A=2    Yes
?- A=2, A is A/2+A/2.
   A=2    Yes
```

O Prolog possui um conjunto de predicados de comparação que funcionam de forma similar ao `is/2`, entre eles temos `=:=`, `<`, `>`, etc. Os predicados aritméticos avaliam as expressões envolvidas, por exemplo, compare o predicado de igualdade aritmética `(=:=)` com a unificação `(=)`:

```
?- X=4, 1+3+2 < X+3.
   X = 4  Yes
?- 3+1<4.
   No
?- 3+5 =:= 5+3.
   Yes
?- 3+5 = 5+3.




   No
```

5.2 Processamento aritmético

Predicados lógicos puros são (ou podem ser escritos como) reversíveis. A reversibilidade é obtida pela unificação, que é bidirecional. No entanto, a unificação não avalia expressões. O problema é que às vezes queremos reduzir (ou calcular) as expressões.

Segue uma versão lógica do programa fatorial, no qual o resultado é devolvido na forma de uma expressão aritmética:

```
fat(X,   1):-X<=1.
fat(X,X*F1):-X> 1,fat(X-1,F1).
```

```
?- fat(4,X).
X = 4* ((4-1)* ((4-1-1)* ((4-1-1-1)*1)))
```

Abaixo, apresentamos uma versão aritmética do fatorial, que é comparada à versão em Pascal. O `is/2` do Prolog corresponde a uma atribuição := do Pascal. Em termos de execução, estes dois programas são similares: para chamar recursivamente `fatorial(X1,F1)`, atribui-se primeiro o valor para `X1`, e para calcular `F`, espera-se o resultado da chamada recursiva.

```
fatorial(X,F):- X=<1, F is 1.
fatorial(X,F):- X>1, X1 is X-1, fatorial(X1,F1), F is X*F1.
%
procedure fatorial(X:int; var F:int); %% em Pascal
begin
  if X<=1 then F:=1;
  if X> 1 then begin X1:=X-1; fatorial(X1,F1); F:=X*F1; end;
end;
```

```
   ?- fatorial(4,F).
      F = 24
```

5.3 Recursividade e iteração

Em Prolog a iteração, de modo geral, é realizada pela recursividade. Programas com recursividade na cauda (tail recursive) são automaticamente otimizados pelos compiladores. São transformados em programas iterativos, com relação aos recursos computacionais usados na sua execução. Portanto, devemos saber como transformar um programa recursivo em um programa com recursividade na cauda. Seguem dois exemplos de programas iterativos, com recursividade na cauda, baseados em acumulador: o cálculo do fatorial (a segunda versão) e a soma dos ímpares até um determinado valor.

```
fat(X,1):-X=<1,!.
fat(X,F):-    !,X1 is X-1, fat(X1,F1), F is F1*X.
%
fati(X,F):-fatAcc(X,1/F).
fatAcc(X,ACC/ACC):-X=<1,!.
fatAcc(X,ACC/F  ):-    !,ACC1 is X*ACC, X1 is X-1,fatAcc(X1,ACC1/F).
%
somaImpar(X,S):-somaAcc(X,0/S).
somaAcc(X,ACC/ACC):-X=<1,!.
somaAcc(X,ACC/S  ):-    !,ACC1 is X+ACC, X1 is X-2,somaAcc(X1,ACC1/S).
```

```
?- fati(3,X).
   X = 6
?- somaImpar(7,S).
   S = 16
```

Na primeira versão `fat/2`, depois da chamada recursiva no corpo, ainda temos uma computação `F is F1*X`, o que impede o compilador de liberar o espaço do programa antes de cada chamada recursiva. Este problema é resolvido na segunda versão `fati/2` com acumulador. Em `fatAcc/2` implementamos o acumulador no segundo parâmetro com o functor `/`, denotando uma ligação entre a variável acumulador e a variável F que recebe o valor final do fatorial. Se substituirmos o operador `/` por uma vírgula o resultado será o mesmo.

5.3.1 Somando os ímpares

Para efeito de comparação apresentamos um outro programa similar ao fatorial, que é a soma dos números ímpares até um dado valor. Os programas `somaImpar` e `fati` baseados em acumulador são bem próximos, o que muda é:

- **inicialização:** o elemento neutro da multiplicação para o fatorial é 1 e para a soma é zero;
- **acumulador:** usa-se a multiplicação para o fatorial (`X*ACC`) e a adição para a soma dos ímpares (`X+ACC`);
- **contagem:** no fatorial é menos um e na soma dos ímpares é menos dois.

Ambos algoritmos, do fatorial e da soma dos ímpares, fazem uma contagem ascendente. No entanto, podemos reescrevê-los fazendo uma contagem ascendente, como exemplificado abaixo para a soma dos ímpares.

```
1 somaImpar2(M,S):- somaAcc2(0/M,0/S).
2 somaAcc2(X/M,   S/S):-X>M,!.
3 somaAcc2(X/M, ACC/S):-ACC1 is ACC+X, X1 is X+2, somaAcc2(X/M,ACC1/S).
```

```
?- somaImpar(7,S).
   S=16
?- somaImpar2(91,S).
   S=184
```

Em `somaAcc2`, usamos um parâmetro `X/M`, denotando que existe uma ligação entre estas duas variáveis. A variável X é iterativamente incrementada e para quando `X>M`. Aqui também poderíamos substituir `X/M` por dois argumentos separados por vírgula.

Em resumo, programas iterativos possuem estruturas similares, assim, pode-se usar um mesmo esquema padrão para programar algoritmos similares. Um esquema compreende a estratégia de solução, por exemplo, usando acumulador e contando ascendentemente.

Exercício 5.1 *Verifique até que valor máximo o seu computador roda o fatorial. Tente para valores como 10 e 20. Teste as duas versões do fatorial,* `fat/2` *e* `fati/2`*.*

Exercício 5.2 *Usando um acumulador, e somente as operações* `(+)(-)(*)`*, calcule X elevado a Y. Assuma X e Y inteiros.*

5.3.2 Valores aleatórios (randômicos)

Um sistema Prolog possui, geralmente, uma função geradora de valores aleatórios. Um valor inteiro entre 0 e N é gerado a cada vez que a função é chamada. Com esta função podemos, por exemplo, programar a geração de uma lista de valores aleatórios. Seguem alguns exemplos da função `random/1` e do predicado `lista_rand/3`:

```
1 lista_rand(   [],V,N):-N<1,!.
2 lista_rand([R|L],V,N):-N1 is N-1, R is random(V), lista_rand(L,V,N1).
```

```
?- X is random(4).
X=3
?- X is random(4).
X=0
?-lista_rand(L,10,5).
L=[3,9,0,1,7]
```

Exercício 5.3 *Um predicado como o* `lista_rand` *serve para avaliar a qualidade do gerador de números aleatórios de um sistema Prolog. Rode diversas vezes este predicado para listas com até centenas de valores, aumentando e diminuindo a faixa de valores. Agora, avalie a qualidade do gerador. Um bom gerador tem pelo menos: cobertura, se gera todos os valores dentro da faixa; e boa distribuição, se gera quantidades uniformes para cada valor; caso contrário, é viciado ou tendencioso em direção a certos valores.*

5.4 Refinamento de programas

Geradores de números são principalmente usados para resolver equações. Por exemplo, se sabemos que a equação $x^2 - 12x + 35 = 0$ tem alguma solução nos valores ímpares menores que 10, podemos usar a estratégia de *geração e teste* para encontrar os valores de `X`, como segue.

```
?- member(X,[1,3,5,7,9]),X**2-12*X+35=:=0.
  X = 5 ;  X = 7 ; No
```

5.4.1 Criptografia aritmética

Um estudo de caso mais interessante para geradores de números é o da criptografia aritmética. Dados três nomes, escritos como listas de variáveis, a idéia é encontrar um valor para cada letra, de modo que a soma seja válida. O exemplo clássico é SEND+MORE = MONEY, mas podem ser quaisquer nomes, como JOSE+MARIA=JOANA. Segue uma solução para o primeiro:

```
 [0, S, E, N, D]     [0, 9, 5, 6, 7]   09567
+[0, M, O, R, E]    +[0, 1, 0, 8, 5]  +01085
=[M, O, N, E, Y]    =[1, 0, 6, 5, 2]  =10652
S = 9, E = 5, N = 6, D = 7, M = 1, O = 0, R = 8, Y = 2
```

Existem soluções triviais como todas as variáveis com valor 0. Para tornar o problema mais interessante, assumimos que cada variável deve possuir um valor diferente. Os nomes SEND e MORE foram completados com um zero na frente, deixando as listas com o mesmo comprimento. Existem várias soluções para esta equação; acima, apresentamos apenas uma delas. Existem, também, algumas combinações de palavras que não possuem solução.

5.4.2 Versão inicial

A solução geral do problema consiste em distribuir aleatoriamente um dígito do conjunto $\{0..9\}$ para cada variável e tentar fazer o cálculo. Se a soma der certo, aquela distribuição é uma solução.

Para selecionar um valor para cada variável usa-se o `select2` que é definido a partir do predicado `select`, apresentado no capítulo sobre listas.

```
?- select2([S,E,N,D],[0,1,7,8,9]).
  S = 0, E = 1, N = 7, D = 8 ;
  S = 0, E = 1, N = 7, D = 9 ;
  S = 0, E = 1, N = 8, D = 7 ;
  ...
```

Segue a solução. O `select2` funciona por retrocesso a cada vez selecionando novos valores para as variáveis.

```
1  q:-select2([S,E,N,D,M,O,R,Y],[0,1,2,3,4,5,6,7,8,9]),
2        Ci=0, Co=0,
3        Yc is Ci+D+E, C1 is Yc // 10, Y is Yc mod 10,
4        Ec is C1+N+R, C2 is Ec // 10, E is Ec mod 10,
5        Nc is C2+E+O, C3 is Nc // 10, N is Nc mod 10,
6        Oc is C3+S+M, C4 is Oc // 10, O is Oc mod 10,
7        Mc is C4+0+0, Co is Mc // 10, M is Mc mod 10,
8        write([0,S,E,N,D]),nl,
9        write([0,M,O,R,E]),nl,
10       write([M,O,N,E,Y]),nl,nl.
11 select2([X|Xs],S):-select(X,S,So),select2(Xs,So).
12 select2([],_).
```

O cálculo é feito a partir da coluna de letras mais à direita: `D+E=Y`. Em cada linha é calculado o vai-um. Ele é calculado pela divisão inteira (//). O valor do dígito é calculado pelo resto da divisão inteira (mod). Por exemplo, na soma de 7+5=12 temos um vai-um (C1=1) que é usado como vem-um na igualdade seguinte C1+N+R.

```
 ?- C is 12 // 10.
    C=1
 ?- D is 12 mod 10.
    D=2
```

O programa q gera todas as soluções pelo retrocesso, forçado pelo fail no final do predicado. Chamamos o predicado `time` para coletar o número de inferências e o tempo necessário para encontrar a primeira solução.

```
?- time(q).
[0, 2, 8, 1, 7]
[0, 0, 3, 6, 8]
[0, 3, 1, 8, 5]
% 4,406,311 inferences, 3.17 CPU in 3.25 seconds
```

5.4.3 Uma solução mais eficiente

A solução apresentada já funciona. Existe uma solução mais eficiente? Sim existe. Um problema da solução anterior é que selecionamos um conjunto de valores para as variáveis, antes de começarmos a computação das equações. Caso uma equação falhe, abandonam-se todos os cálculos e seleciona-se um novo conjunto de valores. Continuamos este processo até encontrarmos um conjunto-solução.

Selecionando-se os valores por demanda, podemos aproveitar os valores calculados: isto é, primeiro calculamos um dígito para uma variável, e então o tiramos da lista dos valores. Para isso precisamos outro predicado de seleção que retorna os dígitos não utilizados: `select3(ListaVars, Si/So)` (i=in, o=out). A diferença entre `Si` e `So` são os valores consumidos pela `ListaVars`. Como segue:

```
?- Si=[0,7,8,9],select3([S,7,N],Si/So).
   S = 0, N = 8, So = [9] ;
   S = 0, N = 9, So = [8] ;
 ...
```

Aqui, o select3 trabalha também com valores constantes. Eles são removidos da lista dos valores não usados.

Segue a versão que seleciona os valores por demanda, usando o predicado `select3`. As chamadas do `select3` são encadeadas em relação à lista de valores disponíveis, `So/S1/S2/S3/So`. É um bom exemplo de uso de diferença de listas, tema que será apresentado no capítulo sobre estruturas de dados.

```
1  r:- Si=[0,1,2,3,4,5,6,7,8,9],
2        Ci=0, Co=0,
3        select3([D,E],Si/S1),
4        Yc is Ci+D+E, C1 is Yc // 10, Y is Yc mod 10,
5        select3([Y,N,R],S1/S2),
6        Ec is C1+N+R, C2 is Ec // 10, E is Ec mod 10,
7        select3([O],S2/S3),
8        Nc is C2+E+O, C3 is Nc // 10, N is Nc mod 10,
9        select3([S,M],S3/So),
10       Oc is C3+S+M, C4 is Oc // 10, O is Oc mod 10,
11       Mc is C4+0+0, Co is Mc // 10, M is Mc mod 10,
12       write([0,S,E,N,D]),nl,
13       write([0,M,O,R,E]),nl,
14       write([M,O,N,E,Y]),nl,nl.
15 select3([X|Xs],S/So):-select(X,S,S1),select3(Xs,S1/So).
16 select3([],S/S).
```

Para chamar este programa, digita-se `?-r`. Ele é sensivelmente mais rápido que a sua versão anterior.

```
 ?- time(r).
[0, 7, 5, 3, 1]
[0, 0, 8, 2, 5]
[0, 8, 3, 5, 6]
% 1,751 inferences, 0.00 CPU in 0.01 seconds
```

Agora podemos comparar o tempo desta solução com a versão anterior. O número de inferências é uma métrica mais precisa e apropriada para comparar dois programas Prolog: mede o número de instruções da máquina abstrata dom Prolog. Esta segunda versão é muito mais eficiente: de mais de 4 milhões de inferências (3.25 seg) para apenas 1751 (0.01 seg).

Exercício 5.4 *Faça uma solução generalizada usando o predicado* `select4` *abaixo.*

```
select4([X|Xs],S/So):-var(X),select(X,S,S1),select4(Xs,S1/So).
select4([X|Xs],S/So):-nonvar(X),            select4(Xs, S/So).
select4([],S/S).
```

```
?- select4([S,4,N,7,D],[0,1,2,3,5,6]/So).
S = 0, N = 1, D = 2, So = [3, 5, 6] ;
S = 0, N = 1, D = 3, So = [2, 5, 6] ;
...
```

Neste predicado, os valores constantes são considerados já selecionados. O `select3` para esta mesma pergunta falha. Para saber se um valor é constante ou variável, foram usados os predicados predefinidos `var` e `nonvar`.

Solução: Segue abaixo uma solução generalizada.

```
somaListas(Xs,Ys,Zs):- somaListaDig(Xs,Ys,Zs,0/0,[0,1,2,3,4,5,6,7,8,9]/_),
                       write(Xs),nl,write(Ys),nl,write(Zs),nl,nl.
somaListaDig([X|Xs],[Y|Ys], [Z|Zs], Ci/Co, Si/So ):-
     somaListaDig(Xs,Ys,Zs, Ci/C1, Si/S1),
     somaDig(X,Y,Z,  C1/Co, S1/So ).
somaListaDig([], [], [], 0/0, S/S).
somaDig(X, Y, Z, Ci/Co, Si/So):-
     select4([X,Y,Z], Si/So),
     S is X+Y+Ci, Co is S // 10, Z is S mod 10.
```

O núcleo desta solução está em `somaDig/5`, que implementa a soma de dois dígitos (uma das equações da versão anterior). Uma diferença está no `select4` que aqui é chamado para as três variáveis envolvidas, já que não sabemos qual delas é variável ou constante. O `somaListaDig` encadeia as chamadas do `somaDig`. O predicado principal `somaListas` inicializa o vem/vai-um com zeros e a diferença de listas com o conjunto dos dígitos disponíveis, com $\{0..9\}$.

Finalmente, podemos testar quaisquer combinações de palavras, em particular JOSE + MARIA = JOANA.

```
?- X=[0,J,O,S,E],Y=[M,A,R,I,A],Z=[J,O,A,N,A], somaListas(X,Y,Z).
[0, 8, 1, 4, 0]
[7, 3, 2, 5, 3]
[8, 1, 3, 9, 3]
X = [0, 8, 1, 4, 0], J = 8, O = 1, S = 4, E = 0
Y = [7, 3, 2, 5, 3], M = 7, A = 3, R = 2, I = 5
Z = [8, 1, 3, 9, 3], N = 9 ;
```

Este programa pode ser usado para somar listas de valores constantes, como segue. Note que em teoria não existe um limite para o comprimento da lista. (Veja também os exercícios abaixo.)

```
?-somaListas([0,1,2,4,5],[0,3,5,6,7],[X,Y,Z,W,T]).
[0, 1, 2, 4, 5]
[0, 3, 5, 6, 7]
[0, 4, 8, 1, 2]
X = 0, Y = 4, Z = 8, W = 1, T = 2;
No
```

Exercício 5.5 *Encontre dois exemplos de listas que não podem ser somadas - sem solução. Sugestão: Force duas variáveis em duas colunas diferentes com valores conflitantes.*

Exercício 5.6 Projeto: *Torne o programa do exercício anterior eficiente: devendo processar listas com mais de mil dígitos. É possível passar a recursividade para a cauda? Justifique. Sugestão: gere listas de números aleatórios para testar o predicado desenvolvido (use o* `findall` *apresentado na próxima seção).*

5.5 Gerar histograma para dados estatísticos

Com o objetivo de integrar os conhecimentos sobre listas, estruturas de controle e programação aritmética esta seção mostra a codificação de funções da estatística com um exemplo de geração de histogramas. Mostramos como gerar valores dentro de uma função de distribuição da estatística. Mostramos também como criar um histograma para uma área ou uma população.

Segue abaixo o programa `stat.pl`. Ele inicia com várias assertivas declarando funções aritméticas. Um predicado pode ser visto como uma função se seu último parâmetro retorna um valor: por exemplo, o predicado `ranf/1` é visto como a função `ranf/0`; o predicado `normal/3` é visto como a função `normal/2`. A vantagem de declarar um predicado como uma função aritmética é poder usá-lo no meio de uma expressão aritmética, deixando o código do programa mais claro e limpo. Por fim, cabe dizer que esta forma de declarar funções ainda não está no Prolog ISO.

```
1  %% stat.pl
2  :- arithmetic_function(ranf/0). % vale no SWI-Prolog
3  :- arithmetic_function(normal/2).
4  :- arithmetic_function(normal_df/3).
5  :- arithmetic_function(expo_df/2).
6  ranf(X) :- N=65536, X is random(N)/N.
7  normal(Avg,Sd,N) :- box_muller(Avg,Sd,N). %% credito:Tim Menzies
8  box_muller(M,S,N) :- w(W0,X),W is sqrt((-2.0 * log(W0))/W0),
9                       Y1 is X * W, N is round(M + Y1*S).
10 w(W,X) :- X1 is 2.0 * ranf - 1,
11           X2 is 2.0 * ranf - 1,
12           W0 is X1*X1 + X2*X2, (W0>=1.0->w(W,X); W0=W,X=X1).
13 normal_df(Avg,Sd,X,AreaY):-
14          Y is exp(-1*((X-Avg)**2)/(2*((Sd)**2))),
15          AreaY is (Y/(sqrt(2*(pi))*Sd)).
16 expo_df(Avg,X,AreaY):- AreaY is (exp(-X/Avg)/Avg).
17 do_df1 :- findall(Y-X,(between(1,10,Y),X is normal_df(5,1,Y)),LB),
18        zipper(LB,L,B),sort(B,Bs),last(Bs,Max),
19        maplist(mult(30/Max),LB,LL),
20        format('\n---| ~w*~w |---', [between(1,10,'Y'),normal_df(5,1,'Y')]),
```

```
21        wDist(30/Max, LL).
22  do_df2 :- findall(Yo-X,(between(0,10,Y),(Yo is Y/10,X is expo_df(0.2,Yo))),LB),
23          zipper(LB,L,B),maxL(B,Max),
24          format('\n---| ~w*~w |---', [between(0,10,'Y'), expo_df(0.2,'Y/10')]),
25          maplist(mult(30/Max),LB,LL),
26          wDist(30/Max,LL).
27  do1 :- N is 10^4, forall(member(F, [normal(50,7),normal(100,3) ]), do2(N,F)).
28  do2(N,F) :- format('\n---| ~w * ~w |-------', [N,F]),
29          findall(X,(between(1,N,_),X is F),L0),
30          bins(L0,LB),zipper(LB,L,B),maxL(B,Max),
31          maplist(mult(30/Max),LB,LL),
32          wDist(30/Max,LL).
33  mult(N,Y-X,Y-Xo) :- Xo is round(X*N).
34  maxL([L|Ls],M):-maxL(Ls,L,M).
35  maxL([L|Ls],ACC,M):- (ACC>L->maxL(Ls,ACC,M);maxL(Ls,L,M)).
36  maxL([],M,M).
37  zipper([(X-Y)|XYs],[X|Xs],[Y|Ys]):-!,zipper(XYs,Xs,Ys).
38  zipper([],[],[]).
39  bins(L0,L) :- msort(L0,L1), bins(L1,[],L).
40  bins([H|T],[H-N0|Rest],Out) :- !, N is N0 + 1, bins(T,[H-N|Rest],Out).
41  bins([H|T],In,Out) :- bins(T,[H-1|In],Out).
42  bins([],X,X).
43  %% escreve histograma
44  wDist(F,L) :- wDist(5,11,5,F,L).
45  wDist(W1,W2,W3,F,L) :- writeln('item frequency'),length(L,LN),
46     forall(between(1,LN,I),(nth1(I,L,(X-N)), wHist([W1,W2],[X,N/F,N])) ).
47  wHist([N,F],[Nd,Fd,Hd]):-
48      sformat(NR,'%~wL',[N]),writef(NR,[Nd]),  %força var no str
49      sformat(FL,'%~wR',[F]),sformat(Fs,'~2f',Fd),writef(FL,[Fs]),
50      writef(' %r\n',['~',Hd]).
51  %% ?- wHist([3,10],[3,7,7]),wHist([3,10],[5,13,13]).
```

O predicado `maplist` que mapeia um predicado aritmético, no exemplo, `plus/3`, para todos os elementos de uma lista, retornando uma nova lista. Seguem alguns testes.

```
?- plus(4,5,X).
   X =9
?- maplist(plus(2),[1,2,3],L).
   L = [3, 4, 5]
```

No Prolog existe a função `random/1` que retorna um valor entre 0 e N. A partir dela, criamos a função `ranf` que retorna um valor `float` entre 0 e 1. Depois, criamos a função `normal` que gera indivíduos dentro de uma curva normal definida pela média e pelo desvio padrão; o método usado na geração é chamado de *Box Muller* nos textos especializados. O predicado `bins` é usado para coletar a freqüência de cada indivíduo. Retorna pares de `X-Y` de indivíduo-freqüência. O predicado `findall` foi apresentado no capítulo sobre listas: ele coleta numa lista todas as soluções. Se gerarmos uma população grande, a média e o desvio padrão da população gerada deve ser bem próxima dos valores passados como parâmetros. Seguem exemplos de testes.

```
?- bins([a,b,b,a,b,c],L).
   L = [c-1, b-3, a-2]
```

```
?- findall(X,(between(1,10,_),X is normal(10,2)),L0).
   L0 = [7, 11, 11, 14, 15, 10, 7, 5, 8|...]
?- findall(X,(between(1,100,_),X is normal(5,1)),L0), bins(L0,F).
   L0 = [5, 6, 4, 5, 6, 6, 5, 5, 5|...]
   F = [7-4, 6-23, 5-46, 4-22, 3-4, 2-1]
```

As funções de densidade `df` descrevem a área abaixo da curva de distribuição de uma população; elas resumem a população em termos estatísticos. Para a curva normal os valores retornados são entre 0 e 1.

Abaixo testamos as funções densidade. Para mostrar estes valores entre 0 e 1 num histograma, é necessário multiplicá-los por um valor que fique numa faixa de valores inteiro positivos; para isso usamos o `maplist/3`.

No segundo teste geramos distribuição para uma média de 5 e desvio padrão de um; geramos dez valores de distribuição e multiplicamos o valor da distribuição por 100 retornando um valor inteiro. Para a média, 5, obtivemos o maior número de indivíduos, 40; depois, para o 4 e 6 foram 24. Podemos comparar esta saída com a saída acima, onde a média 5 teve 46 indivíduos; etc. Para os valores serem mais próximos é só executar uma pergunta que gera muitos indivíduos, por exemplo, 10 mil indivíduos.

```
?- findall(X,(between(1,10,Y),X is normal_df(5,2,Y)),L0).
   L0 = [0.0381774, 0.0915828, 0.171099, 0.248948, 0.282095, ...]
?- findall(Y-X,(between(1,10,Y),X is normal_df(5,1,Y)),L0),
   maplist(mult(100),L0,LM).
 L0 = [1-0.00013383, 2-0.00443185, 3-0.053991, 4-0.241971, 5-0.398942,...]
 LM = [1-0, 2-0, 3-5, 4-24, 5-40, 6-24, 7-5, 8-0, ...]
?- findall(Yo-X,(between(0,10,Y),(Yo is Y/10,X is expo_df(0.5,Yo))),L0),
   maplist(mult(10),L0,LL).
   L0 = [0-2.0, 0.1-1.63746, 0.2-1.34064, 0.3-1.09762, 0.4-0.898658, ...
   LL = [0-20, 0.1-16, 0.2-13, 0.3-11, 0.4-9, 0.5-7, 0.6-6, 0.7-5, ...]
```

Por fim podemos gerar um histograma para função de distribuição. Basicamente chama-se um predicado `wHist` que escreve uma linha do histograma. Este predicado utiliza o predicado `format` para três situações: para alinhar à esquerda o valor, alinhar à direita o valor da freqüência e escrever N vezes o caractere do histograma. Segue dois exemplos de chamadas para os histogramas.

```
?- wHist([3,10],[3,7,7]),wHist([3,10],[5,13,13]).
3      "7.00" ~~~~~~~
5     "13.00" ~

?- do_df2.
---| between(0, 10, Y) * expo_df(0.2, Y/10) |---item frequency
0         "5.00" ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
0.1       "3.00" ~~~~~~~~~~~~~~~~~
0.2       "1.83" ~~~~~~~~~~
0.3       "1.17" ~~~~~~~
0.4       "0.67" ~~~~
0.5       "0.33" ~~
0.6       "0.17" ~
0.7       "0.17" ~
0.8       "0.17" ~
```

```
...

?-do1.
---| 10000 * normal(100, 3) |-------item frequency
111       "0.00"
110       "0.00"
109       "0.00"
108      "45.17" ~
107      "90.33" ~~
106     "180.67" ~~~~
105     "316.17" ~~~~~~~
104     "542.00" ~~~~~~~~~~~~
103     "813.00" ~~~~~~~~~~~~~~~~~~
102    "1038.83" ~~~~~~~~~~~~~~~~~~~~~~
101    "1219.50" ~~~~~~~~~~~~~~~~~~~~~~~~~~
100    "1355.00" ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
...
```

5.5.1 Conclusão

Desenvolvemos um programa para resolver criptografia aritmética em três versões. A primeira versão apresenta a solução razoável. A segunda versão, que é bem procedural, apresenta a solução mais eficiente. A terceira apenas codifica a segunda solução numa forma generalizada.

Paralelamente, foram desenvolvidas variações de predicado de seleção, todas baseadas no `select`, apresentado no capítulo sobre listas. Este é um bom exemplo da flexibilidade de reuso do `select`.

Na seção final mostramos como declarar funções aritméticas e apresentamos vários exemplos de predicados de listas sendo utilizados na geração de histogramas. Mostramos também como utilizar os comandos de controle `forall`, `findall` e `between`.

CAPÍTULO 6

Estruturas de Dados

No Prolog, as constantes são os itens de dados básicos. Tipos de dados estruturados são representados como termos. Um termo representa uma estrutura de dados na forma de árvore. Por exemplo, o termo `1+3*4` e o termo `dados('Joao', 'OK', end('rua X', 23))` são representados como segue:

A expressão aritmética é uma árvore binária: cada nó tem no máximo dois filhos. A árvore para o termo `dados/3` tem um nó ternário e outro binário. Em Prolog, podemos representar nós de árvores com qualquer número de filhos.

Com base nesta noção de estruturas de dados, neste capítulo exploramos as técnicas de programação para representar e manipular dados e estruturas de dados diferentes de listas, tais como árvores e grafos.

6.1 Manipulação de Dados

O Prolog possui uma base de fatos (e regras) que pode ser usada para processar dados de uma forma bem próxima do **modelo relacional de banco de dados[9]**.

O modelo relacional tem por base tabelas formadas por linhas. Numa tabela, cada linha é formada por um mesmo número de colunas. Existem operações para criar e atualizar as tabelas (inserir e excluir linhas) e outras para recuperar a informação das tabelas.

Uma linguagem de consulta para estas tabelas é a álgebra relacional, que compreende as operações de:

- **seleção:** seleciona uma ou mais linhas de uma tabela;
- **projeção:** seleciona uma ou mais colunas de uma tabela;

- **junção:** junta duas tabelas, que possuem valores comuns para uma mesma coluna;
- **união:** junta as linhas de duas tabelas numa mesma tabela;
- **diferença:** dadas duas tabelas, remove da primeira as linhas da segunda; e,
- **produto cartesiano:** combina cada linha de uma tabela com todas as linhas de outra tabela.

Estas operações relacionais para manipular tabelas, formadas por regras e fatos, são facilmente implementadas no Prolog. Dadas duas relações `s` e `r`, abaixo, definimos alguns esquemas para implementá-las em Prolog.

```
% projeção de Xi,Xj em r
r_p_i_j(Xi,Xj) :- r(X1,...,Xn).

% selecao em r baseada na condição c
r_select_c(X1,...,Xn) :- r(X1,...,Xn), c(X1,...,Xn).

% r juncao com s (remove-se colunas repetidas)
r_juncao_s(X'1,...,X'j,Y'1,...,Y'k) :- r(X1,...,Xn), s(Y1,...,Ym).

% uniao de r e s
r_uniao_s(X1,...,Xn) :- r(X1,...,Xn) ; s(X1,...,Xn).

% r menos (diferenca) s
r_menos_s(X1,...,Xn) :- r(X1,...,Xn), \+ s(X1,...,Xn).

% produto cartesiano de r e s
r_x_s(X1,...,Xm,Y1,...,Yn) :- r(X1,...,Xm), s(Y1,...,Yn).

% intercessão de r e s
r_inter_s(X1,...,Xn) :- r(X1,...,Xn), s(X1,...,Xn).
```

Vamos explorar as três primeiras operações, as outras são deixadas como exercícios. Os exemplos são baseados nas tabelas a seguir:

```
%produto(NOME, FORNECEDOR, QTY, DATA).
produto(tomate, jose,     20, 20/10).
produto(feijao, joao,     10, 21/10).
produto(arroz,  antonio,  40, 20/10).
produto(sal,    maria,     3, 21/10).
produto(acucar, ana,      5,  20/10).
produto(copos,  jose,   1000, 20/10).
produto(pratos, maria,   100, 21/10).

%fornecedor(NOME, FONE, END).
fornecedor(jose, 1503, rua5).
fornecedor(antonio, 1402, rua21).
```

Uma projeção sobre a tabela `produto` para listar o nome dos produtos e fornecedores é mostrada abaixo.

```
?-produto(NOME, FORNEC,_,_).
NOME = tomate FORNEC = jose ;
NOME = feijao FORNEC = joao ;
NOME = arroz  FORNEC = antonio ;
...
```

Devemos ter o cuidado de, na consulta, sempre usar uma variável anônima (_) nas colunas que não são desejadas, pois uma pergunta sobre um predicado com um menor número de argumentos sempre falha.

Uma operação de seleção extrai diversas linhas, por exemplo, *todos os produtos com data = 21/10*, como segue.

```
?-produto(NOME, FORNEC,QTY,DT), DT=21/10.
NOME = feijao FORNEC = joao  QTY = 10 DT = 21/10 ;
NOME = sal    FORNEC = maria QTY = 3  DT = 21/10 ;
...
```

Podemos também combinar estas operações para fazer consultas mais especializadas. As duas consultas exemplificadas podem ser combinadas numa só, como segue:

```
?-produto(NOME, FORNEC,_,DT), DT=21/10.
```

A junção permite fazer consultas que combinam mais de uma tabela ao mesmo tempo, por exemplo, *liste o nome de cada produto associado ao nome e telefone do fornecedor*, como segue. Neste caso estamos também fazendo uma projeção porque listamos somente parte dos campos.

```
?- produto(NOME, FORNEC,_,_), fornecedor(FORNEC, FONE,_).
NOME = tomate FORNEC = jose    FONE = 1503 ;
NOME = arroz  FORNEC = antonio FONE = 1402 ;
NOME = copos  FORNEC = jose    FONE = 1503
```

É fácil programar visões como predicados, por exemplo, *todos os produtos fornecidos na data 21/10*.

```
produto1(N,F,QTY, DT):- produto(N, F,QTY,DT), DT=21/10
```

Exercício 6.1 *Faça um predicado para a operação de diferença, calculando a diferença da tabela* `produto` *com a tabela* `produto1` *definida na visão acima.*

Exercício 6.2 *Defina uma visão parcial da tabela produto,* `produto2`*, com (QTY < 50), e numa consulta ou predicado junte com a visão* `produto1`*.*

Exercício 6.3 *Defina um predicado que retorna o produto cartesiano entre as tabelas produtos e fornecedores.*

6.1.1 Metapredicados

Acima, exemplificamos as operações como consultas feitas ao interpretador Prolog. Para obtermos múltiplas respostas digitamos (;) após cada resposta. Já vimos que podemos usar recursos de controle de fluxo para obter múltiplas respostas, por exemplo, o uso do `fail`, como segue.

```
?- produto(NOME, FORN,_,_), write(prod(NOME,FORN)),nl,fail.
prod(tomate, jose)
prod(feijao, joao)
prod(arroz, antonio)
...
```

O Prolog possui outros predicados que são mais apropriados para se coletar um conjunto de respostas, que são:

- `findall/3`: coleta numa lista todas os possíveis valores para respostas, incluindo valores repetidos;
- `bagof/3`: similar ao `findall`, porém pode-se controlar quais variáveis devem ou não permanecer livres; a lista que retorna com valores (possivelmente repetidos) é também chamada de saco (bag);
- `setof/3`: similar ao `bagof`, porém coleta todas as repostas numa lista sem valores repetidos.

Abaixo, apresentamos alguns exemplos ilustrativos para comparar o uso destes predicados.

```
?- findall(prod(N,F), (produto(N, F,_,DT), DT=21/10), RESP).
N =_ F = _ DT = _
RESP = [prod(feijao, joao), prod(sal, maria), prod(pratos, maria)]

?- bagof(prod(N, F), (produto(N, F, _, DT), DT=21/10), RESP).
N = _ F = _ DT = 21/10
RESP = [prod(feijao, joao)] ;
N = _ F = _ DT = 21/10
RESP = [prod(sal, maria)] ;
N = _ F = _ DT = 21/10
RESP = [prod(pratos, maria)] ;
No

?- bagof(Y,X^C^D^produto(X, Y, C, D), RESP).
Y = X = C = D = _
RESP = [jose, joao, antonio, maria, ana, jose, maria]

?- setof(Y,X^C^D^produto(X, Y, C, D), RESP).
Y = X = C = D = _
RESP = [ana, antonio, joao, jose, maria]

?- setof(Y,X^C^produto(X, Y, C, D), RESP).
Y = X = C _   D = 20/10
RESP = [ana, antonio, jose]
```

A notação `X^C^D^produto(X, Y, C, D)` marca as variáveis `X`, `C` e `D` para permanecerem livres. Caso contrário o sistema unifica estas variáveis com o primeiro valor encontrado, como exemplificado na última pergunta em que o `D=20/10`. Um `bagof/3` com todas as variáveis marcadas como livres corresponde a um `findall/3`.

Um `setof/3` funciona como um `bagof`, porém no final o resultado é ordenado e os valores repetidos são removidos.

6.1.2 Incluindo e excluindo fatos e regras

No Prolog dinamicamente podemos **incluir** e **excluir** regras ou fatos. Uma **alteração** pode ser feita pela exclusão e inclusão da informação alterada.

Para incluir e excluir fatos e regras são utilizados os predicados que seguem:

- `:-dynamic(Functor/Aridade)` —(é uma assertiva) declara que o predicado `Functor/Aridade`. pode ser modificado durante a execução do programa.
- `assert(Clausula)` — armazena de modo permanente a cláusula passada como parâmetro. A cláusula será armazenada no fim da lista de cláusulas associadas ao predicado.
- `asserta(Clausula)` — idem `assert`; porém a cláusula será armazenada no início da lista de cláusulas associadas ao predicado.
- `retract(Clausula)` — remove uma cláusula da base de fatos. As cláusulas a serem removidas devem ser declaradas como dinâmicas.
- `abolish(Functor/Aridade)` — remove todas as cláusulas do predicado definido pelo functor e pela aridade. A declaração do predicado como do tipo dinâmico é juntamente removida.

Segue um exemplo que faz uso de alguns dos predicados descritos.

```
1  %% incluir aqui as tabelas produtos/4 e fornecedores/3
2  :-abolish(prod2/4).
3  :-dynamic(prod2/4).
4
5  copiaProd :- produto(X,Y,Z,W), assert(prod2(X,Y,Z,W)),fail.
6  copiaProd :- listing(prod2).
7
8  remProd2(X):-retract(prod2(X,_,_,_)), listing(prod2).
9
10 replProd2(X,Y):- retract(prod2(X,A,B,C)),
11                  assert(prod2(Y,A,B,C)), listing(prod2).
```

Acima, definimos duas assertivas. A primeira `:-abolish(prod2/4)` remove quaisquer fatos ou regras do predicado `prod2/4`. A segunda `:-dynamic(prod2/4)` declara que podemos modificar o predicado `prod2/4`. A tentativa de modificar um predicado que não é dinâmico causa uma exceção ou erro.

Abaixo, temos a execução de uma operação `copiaProd` que copia uma tabela e inclui uma linha de cada vez com o comando `assert`. Depois, usando o `retract` removemos a linha do produto *sal*. Em seguida, substituímos o nome do produto *açúcar* pelo nome *mascavo*, excluindo e reincluindo a linha do produto. Após cada operação é usado o predicado `listing(prod2)` para mostrar o efeito da sua execução.

```
?- copiaProd.
prod2(tomate, jose, 20, 20/10).
prod2(feijao, joao, 10, 21/10).
prod2(arroz, antonio, 40, 20/10).
prod2(sal, maria, 3, 21/10).
prod2(acucar, ana, 5, 20/10).
```

```
prod2(copos, jose, 1000, 20/10).
prod2(pratos, maria, 100, 21/10).
Yes
?- remProd2(sal).
prod2(tomate, jose, 20, 20/10).
prod2(feijao, joao, 10, 21/10).
prod2(arroz, antonio, 40, 20/10).
prod2(acucar, ana, 5, 20/10).
prod2(copos, jose, 1000, 20/10).
prod2(pratos, maria, 100, 21/10).
Yes
?- replProd2(acucar,mascavo).
prod2(tomate, jose, 20, 20/10).
prod2(feijao, joao, 10, 21/10).
prod2(arroz, antonio, 40, 20/10).
prod2(copos, jose, 1000, 20/10).
prod2(pratos, maria, 100, 21/10).
prod2(mascavo, ana, 5, 20/10).
```

Por fim, lembramos que o Prolog não é um gerenciador de BD, os fatos e regras são manipulados em memória e considerados parte do próprio programa.

Em teoria da informação existe uma hierarquia: ruído < dado < informação < conhecimento < sabedoria. Dados são objetos isolados, por exemplo, valores constantes. Fatos são relacionamentos entre objetos, associações entre dados e fatos expressam informação. Cada argumento de um fato é um objeto ou valor, por exemplo, o fato

`produto(NOME=tomate, FORNECEDOR=joão, QTY(k)=20, DATA=20/10)`

expressa uma informação, relacionando o produto com o fornecedor. Um predicado (com regras) pode processar um conjunto de fatos sobre fornecedores e produtos para saber sobre o estoque atual. Assim, predicados, de certo modo, codificam o conhecimento necessário para extrair uma determinada informação.

Do ponto de vista da ciência da computação, fatos codificam dados e informação; e regras codificam programas ou conhecimento. Os dados e a informação têm uma natureza estática. O conhecimento tem uma natureza dinâmica que é baseada em experimentos ou atividade prática[1]. Tipicamente, um dos efeitos de um experimento é um conjunto de novas informações.

No Prolog não existe uma diferença significativa na codificação de dados e programas: ambos são codificados como cláusulas. Porém, fatos são mais relacionados com dados e regras são mais relacionadas com programas.

6.2 Árvores binárias

Uma árvore organiza um conjunto de itens de dados numa estrutura hierárquica, com algum propósito computacional. Por exemplo, uma árvore binária pode ser usada para calcular o valor de uma expressão aritmética. Os nós internos são os operadores e os nós das folhas são os valores numéricos. O cálculo é feito percorrendo-se a árvore de baixo para cima.

Uma árvore binária pode ser definida indutivamente, a partir de dois construtores de nós:

[1] Um estudante só adquire o conhecimento **praticando** a informação do livro. Apenas a posse da informação decorada, por exemplo, não é sinônimo de conhecimento. Num primeiro estágio, a prática confirma a informação e num segundo estágio leva a uma convicção que acontece no nível do conhecimento. A informação está nos livros e nas máquinas, mas o conhecimento está no homem.

- o nó vazio corresponde a uma árvore vazia (que pode ser `[]`);
- um nó não vazio possui uma informação e dois nós-filho do tipo árvore (que pode ser `no(INFO, ESQUERDO, DIREITO)`).

Abaixo, temos um exemplo de árvore com sete nós não vazios; em que o nó-pai é o (a); os nós (b,c,e) são internos; e os nós (d,g,f) estão na fronteira.

```
1          a
2         / \              na árvore
3        b   c             no(INFO,E,D)  ex: no(a,E,D)
4       /   / \            é representado como
5      d   e   f           INFO(E,D)     ex: a(E,D)
6           \
7            g
```

Indutivamente, cada nó é um termo: `no(a,ESQ,DIR)`, `ESQ = no(b, ESQ1,[])`, `DIR = no(c, ESQ2, DIR3)`, etc.; um nó-folha possui dois filhos vazios: `ESQ1 = no(d,[],[])`; etc. Numa primeira representação para esta árvore, todos estes termos de nós individuais são unidos num único termo que é armazenado num predicado `arvore/1` na base de dados do Prolog, como segue.

```
1 arvore(no(a, no(b, no(d,[],[]),[]),
2              no(c, no(e,[],no(g,[],[])),
3                    no(f,[],[])) )).
4 wArv(_,[]).
5 wArv(N,no(I,E,D)):-tab(N),write(I),nl,
6                    N1 is N+1, wArv(N1,E),wArv(N1,D).
```

O predicado `wArv/2` escreve a árvore lateralmente indentada, representada com o nó-pai mais à esquerda e os nós-folha mais à direita.

```
?-␣arvore(A),␣␣wArv(1,A).
␣a
␣␣b
␣␣␣d
␣␣c
␣␣␣e
␣␣␣␣g
␣␣␣f
A␣=␣no(a,␣no(b,␣no(d,[],[]),[]),no(c,no(e,no(g,[],[]),[]),␣no(f,[],[])))
```

Em Prolog, árvores são modeladas com termos. Um termo é um functor com argumentos. Quando necessário, termos são armazenados como fatos. Uma **segunda representação** para a árvore descrita acima são sete termos, em que cada um é um nó armazenado num fato, como segue.

```
1 no_arv(a,b,c).
2 no_arv(b,d,[]).
```

```
3  no_arv(d,[],[]).
4  no_arv(c,e,f).
5  no_arv(e,[],g).
6  no_arv(f,[],[]).
7  no_arv(g,[],[]).
8  %%
9  wArv1(_,[]).
10 wArv1(N,X):-no_arv(X,E,D),tab(N),write(X),nl,
11             N1 is N+1, wArv1(N1,E),wArv1(N1,D).
```

Nesta segunda representação, para escrever a árvore chama-se `?-wArv1(1,a)`; aqui é passado o valor do nó-raiz a. Assim, podemos chamar diretamente a escrita de qualquer subárvore, por exemplo:

```
?-␣wArv1(1,c).
␣␣c
␣␣␣e
␣␣␣␣g
␣␣␣f
```

O conteúdo das duas representações da árvore é o mesmo. Diferentes representações atendem diferentes necessidades. A segunda representação oferece alguma flexibilidade para se processar isoladamente os nós; podemos acessar diretamente um nó sem caminhar na árvore. Para se processar a árvore toda, a primeira representação num só fato é mais eficiente: a base de dados é acessada uma só vez na leitura da árvore toda.

6.2.1 Mais sobre representação de informação

Um functor zero-ário representa uma constante — item de informação. Um functor unário representa um rótulo num item de informação — um atributo de um indivíduo. Por exemplo, em `bin(10)`, o `bin` designa uma qualidade, um atributo, do valor 10: ser binário. Aqui, ainda falamos de um só indivíduo. Com um functor binário, temos um construtor de estruturas que permite fazer relações entre dois indivíduos: `pai(joao, maria)`. Uma árvore binária, como estrutura de dados recursiva, é uma generalização de uma lista. Por exemplo, a árvore `no(a,no(b,no(c,[],[]),[]),[])` equivale a uma lista `'.'(a,'.'(b,'.'(c,[])))`. Na lista temos um functor binário: `'.'(INFO, CAUDA_LISTA)`; na árvore temos um functor ternário: `no(INFO, ARV, ARV)`. Estruturas mais complexas que árvores binárias são criadas com functores com um arbitrário número de filhos.

Exercício 6.4 *Desenhe a estrutura de árvore para o termo canônico da lista* `[a,b,c]`*, que é* `'.'(a,'.'(b,'.'(c,[])))`*; e para a árvore* `no(a,no(b,no(c,[],[]),[]),[])`*. Compare-os.*

6.2.2 Percurso em árvores binárias

Um problema clássico de estruturas de dados é o percurso (ou caminhamento) em árvores binárias. Três tipos de percurso são populares (devido a D. Knuth): pré-ordem, pós-ordem e *in*-ordem. Estes modos de caminhamento são baseados num algoritmo recursivo que executa uma busca em profundidade, a partir do nó-raiz: de cima para baixo e da esquerda para a direita — sempre visita primeiro a subárvore da esquerda e depois a subárvore da direita.

- **Pré-ordem:** escreve o dado do nó, e então visita o nó esquerdo e depois o direito.
- **In-ordem:** visita o nó esquerdo, escreve o dado do nó e visita o nó direito.
- **Pós-ordem:** visita o nó esquerdo, visita o nó direito e depois escreve o dado.

Em cada caminhamento, o que muda é o momento em que é usada a informação do nó visitado. Por exemplo, no pré-ordem o `write(I)` está no início do corpo da cláusula, no in-ordem está no meio e no pós-ordem está no fim.

```
1  preOrdem(       []).
2  preOrdem(no(I,E,D)):- write(I),preOrdem(E),preOrdem(D).
3   inOrdem(       []).
4   inOrdem(no(I,E,D)):- inOrdem(E),write(I),inOrdem(D).
5  posOrdem(       []).
6  posOrdem(no(I,E,D)):- posOrdem(E),posOrdem(D),write(I).
7  %%
8  p:- arvore(A),
9      write(preOrdem), write(':'),preOrdem(A),nl,
10     write(inOrdem), write(' :'), inOrdem(A),nl,
11     write(posOrdem), write(':'),posOrdem(A),nl.
```

Estes predicados podem ser testados isoladamente ou em grupo pelo predicado p, como segue:

```
?-arvore(A),preOrdem(A),nl.
  abdcegf
  A=no(a,no(b,no(d,[],[]),[]),no(c,no(e,no(g,[],[]),[]),no(f,[],[])))
  Yes
?-p.
  preOrdem:abdcegf
  inOrdem :dbagecf
  posOrdem:dbgefca
  yes
```

Exercício 6.5 *Defina uma representação alternativa para árvores: baseada em listas, em que o nó* `no(I,E,D)` *é representado como* `[I,E,D]`*. Refaça o predicado* `wArv` *e o predicado* `preOrdem`*.*

EXEMPLO 6.1 (Desenhando árvores)

Podemos pensar num algoritmo que desenha uma árvore. Para facilitar, assumimos que a árvore é desenhada da esquerda para a direita, como mostrado abaixo.

```
?-arvore(A),␣␣wArv2(0,A),nl.
␣␣␣␣f
␣␣c
␣␣␣␣e
␣␣␣␣␣␣g
a
␣␣b
␣␣␣␣d
```

Neste desenho da árvore, temos os nós espacialmente distribuídos, faltando apenas as linhas que ligam os pais e filhos. A solução é um programa wArv2 similar ao percurso in-ordem, porém antes visita-se o nó da direita e depois o nó da esquerda, como segue.

```
wArv2(_,[]).
wArv2(N,no(I,E,D)):-N1 is N+2,wArv2(N1,D),
                                tab(N),write(I),nl,
                                wArv2(N1,E).
```

Neste exercício, estamos fazendo um outro tipo de caminhamento em profundidade, da direita para a esquerda. Assim como temos três possibilidades (pré, in, pós) no sentido da esquerda para à direita temos também três possibilidades complementares no sentido da direita para à esquerda.

Exercício 6.6 ***Faça um programa* wArv3*, similar ao* wArv2*, que escreve junto com cada nó o número da linha e coluna. Como segue.*

```
?- arvore(A), wArv3(1/Lo,1,A).
     f,5,1
   c,3,2
     e,5,3
       g,7,4
 a,1,5
   b,3,6
     d,5,7
       A = no(a, no(b, no(d, [], []), []),...
         Lo = 8
```

Solução: Este programa pode ser usado para se desenhar a árvore numa matriz (ou grade) de linhas e colunas, em que cada nó da árvore ocupa uma posição, nas coordenadas (coluna, linha). Por exemplo, o nó d está na coluna 5 da linha 7; o nó-raiz a está na coluna 1 da linha 5. Segue o solução.

```
wArv3(L/L,_,[]).
wArv3(L/Lo,N,no(I,E,D)):-
        N1 is N+2,
        wArv3(L/L1, N1,D),
        tab(N),write((I,N,L1)),nl,
        L2 is L1+1,
        wArv3(L2/Lo, N1,E).
```

Com esta solução, estamos caminhando em direção a um programa gráfico de desenho para árvores. Podemos assumir que temos duas primitivas gráficas: uma para escrever um nó numa posição especificada por (coluna, linha) e outra primitiva para desenhar um arco de uma posição da matriz para outra.

A solução apresentada é estendida pelo exercício que segue, escrevendo também quais os arcos que devem ser desenhados para termos uma árvore.

Exercício 6.7 ***Faça* wArv4*, a partir de* wArv3*, em que são escritos também os arcos necessários para desenhar a árvore, como exemplificado abaixo:*

```
?- arvore(A), wArv4(1/Lo,1/_,A).
     no(f, 5/1)
   no(c, 3/2)
     no(e, 5/3)
       no(g, 7/4)
arco(5/3, 7/4)
arco(3/2, 5/3)
arco(3/2, 5/1)
 no(a, 1/5)
   no(b, 3/6)
     no(d, 5/7)
arco(3/6, 5/7)
arco(1/5, 3/6)
arco(1/5, 3/2)
   A = no(a, no(b, no(d, [], []), []), ...
   Lo = 8
```

Solução: Aqui vemos que para construirmos a subárvore c temos que desenhar três arcos. Um arco entre c-f, da posição 3/2 para 5/1; um arco entre c-e, da posição 3/2 para 5/3; e outro arco entre e-g, da posição 5/3 para 7/4. Segue a solução.

```
1 wArv4(L/L,_,[]).
2 wArv4(L/Lo,N/L1,no(I,E,D)):-
3         N1 is N+2,
4         wArv4(L/L1, N1/Ld,D),
5         tab(N),write(no(I,N/L1)),nl,
6         L2 is L1+1,
7         wArv4(L2/Lo, N1/Le,E),
8         (E=[]->true; write(arco(N/L1,N1/Le)),nl),
9         (D=[]->true; write(arco(N/L1,N1/Ld)),nl).
```

Acrescentamos o comando de escrita dos arcos, considerando que, quando um nó-filho é vazio, não devemos desenhar o arco. Para desenhar o arco foi necessário acrescentar um parâmetro L1, que informa a linha onde o nó é desenhado. No nó-pai este parâmetro é usado para saber em que linhas os dois nós-filho são desenhados (Le, Ld). Um arco é desenhado entre um nó-pai e dois nós-filho.

Para se fazer um desenho gráfico da árvore é necessário pensar em primitivas gráficas que exibem cada nó numa posição de uma matriz ou grade. Cada valor de linha e coluna pode ser multiplicado pelo número de pixels que define graficamente o quadrado de uma posição da matriz. Por exemplo, assumindo caracteres com o mesmo tamanho, toma-se o valor em pixels da altura e largura de um caractere, como o tamanho de uma posição da matriz.

EXEMPLO 6.2 (Retornando uma lista de nodos)

Um algoritmo alternativo de percurso (por exemplo, pré-ordem) pode retornar uma lista de nós, em vez de escrevê-los na saída. Abaixo, temos uma versão pre_ordem2 em que a lista dos nós visitados é coletada com o auxílio do append.

```
preOrdem2(       [], []).
preOrdem2(no(I,E,D), [I|EDs]):- preOrdem2(E,Es),preOrdem2(D,Ds),
                                append(Es,Ds,EDs).
```

```
?- arvore(A),preOrdem2(A,L).
   A = no(a,no(b,no(d,[],[]),[]),no(c,no(e,no(g,[],[]),[]),no(f,[],[]))) ,
   L = [a,b,d,c,e,g,f]
```

Exercício 6.8 *Faça os caminhamentos* `inOrdem2/2` *e* `posOrdem2/2` *para retornarem uma lista com os nós da árvore.*

Exercício 6.9 *Modifique o caminhamento* `inOrdem/1` *para escrever somente os nós-folha da árvore, os que não têm filhos.*

Exercício 6.10 *Modifique o caminhamento* `preOrdem2/2` *para retornar somente os nós-folha da árvore, os que não têm filhos.*

Solução:

```
preOrdem2(         [],  []):-!.
preOrdem2(no(I,[],[]), [I]):-!.
preOrdem2(no(_,E,D),   EDs):- !,
        preOrdem2(E,Es),preOrdem2(D,Ds),append(Es,Ds,EDs).
```

6.3 Árvore de busca

Uma estrutura de árvore pode ser usada como índice para uma coleção de dados, por exemplo, um arquivo. Um índice segue uma ordem, por exemplo, ordem crescente para valores numéricos. Uma árvore binária ordenada é ilustrada abaixo: todos os filhos à esquerda de um nó-pai têm um valor menor que o valor do nó-pai e todos filhos à direita têm um valor maior que o valor do nó-pai. Assim, um caminhamento em in-ordem resulta numa lista ordenada, em ordem crescente.

```
                  7
                /   \
               5     12
              / \    /  \
             3   6  11  15
inOrdem: 3,5,6,7,9,11,12,15
```

Uma árvore com o papel de índice de busca a um conjunto de dados é chamada de árvore de busca. Normalmente, num arquivo de dados são usadas três operações básicas:

- busca: dado um valor, acessar o nó correspondente;
- inserção: dado um valor, inseri-lo na árvore, de forma que a árvore continue ordenada;
- remoção: dado um valor, remover o nó correspondente.

6.3.1 Busca

Como organizações de arquivos, as árvores de busca reduzem significativamente o número de comparações para acessar os dados.

A busca de um valor X, numa árvore de busca é codificada em três regras, que consideram o valor de cada nó I, `no(I,E,D)`:

- se `X=I`, então retorna sucesso;
- se `X<I`, então busca na subárvore da esquerda;
- se `X>I`, então busca na subárvore direita.

Abaixo, codificamos estas regras no predicado `esta/2`. Não temos uma regra para uma busca de um valor numa árvore vazia, então, neste caso o predicado falha.

```
1 esta(X,no(I,E,D)):-X=I,!.
2 esta(X,no(I,E,D)):-X<I,!,esta(X,E).
3 esta(X,no(I,E,D)):-X>I,!,esta(X,D).
4 %
5 inOrdem2(       [], []).
6 inOrdem2(no(I,E,D), EiDs):- inOrdem2(E,Es),inOrdem2(D,Ds),
7                             append(Es,[I|Ds],EiDs).
```

Este esquema de busca é também usado para a inserção e a remoção de nós. Na remoção é necessário navegar até a posição em que está o nó a ser removido e na inserção navega-se até a fronteira da árvore em que o nó deve ser inserido.

Seguem dois exemplos de uso do predicado `esta`. O predicado `arvore2` retorna uma árvore (ele é apresentado logo mais). O predicado `inOrdem2` retorna uma lista ordenada com os nós da árvore.

```
?- arvore2(A),inOrdem2(A,I).
   A = no(7,no(5,no(3,[],[]),no(6,[],[])),no(12,no(11,no(9,...)), ...))) ,
   I = [3,5,6,7,9,10,11,12,13,14,15,17]
?- arvore2(A),esta(2,A).
   No
?- arvore2(A),esta(11,A),!.
   Yes
```

6.3.2 Inserção

A inserção, `ins/3`, funciona com o mesmo esquema básico de uma busca, porém com dois parâmetros adicionais: a árvore que entra e a árvore que sai, já com o novo nó. Ela também tem uma regra para tratar da inserção de um valor numa árvore vazia, que é onde é realizada a inserção. Assim, consideramos dois principais casos, como segue:

- para inserir um valor numa árvore vazia, retorna-se um nó com o valor e dois filhos vazios;
- para inserir um valor X numa (sub)árvore com um nó-raiz com valor Y, considera-se três casos:
  - se `X=I`, então despreza-se o valor X;

- se X<I, insere-se na subárvore da esquerda
- se X>I, insere-se na subárvore da direita

Em resumo, se o valor já está na árvore, o valor é desprezado. Senão, busca-se a posição. A inserção em si ocorre numa folha. Abaixo as regras são codificadas no Prolog.

```
1  ins(X,[],no(X,[],[])):-!.
2  ins(X,no(I,E,D),no(I,E,D)):- X=I,!.
3  ins(X,no(I,E,D),no(I,E1,D)):-X<I,!,ins(X,E,E1).
4  ins(X,no(I,E,D),no(I,E,D1)):-X>I,!,ins(X,D,D1).
5  %%
6  insL([X|Xs],A/Ao):-!,ins(X,A,A1),insL(Xs,A1/Ao).
7  insL([],A/A):-!.
8  %%
9  arvore1(A):-insL([7,5,12,3,6],[]/A).
10 arvore2(A):-insL([7,5,12,3,6,11,9,10,15,13,17,14],[]/A).
11 arvore3(A):-insL([3,4,4,5,6],[]/A).
```

Para facilitar a construção de uma árvore a partir de uma lista de valores temos o predicado insL. Usando um acumulador, ele constrói uma árvore, passo a passo. Temos também três predicados criando três árvores para testes: arvore1, arvore2 e arvore3. Seguem alguns testes para estes predicados:

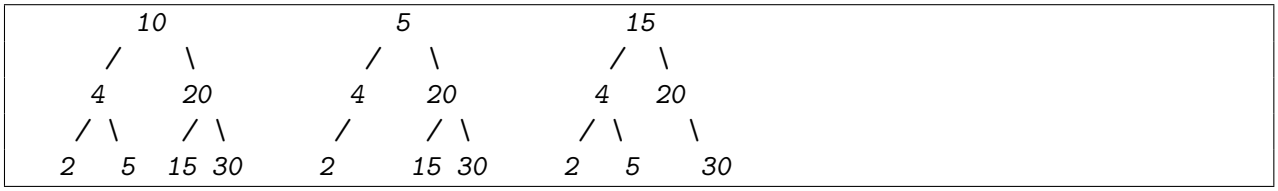
```
?- arvore1(A),wArv(0,A),inOrdem2(A,I).
7
 5
  3
  6
 12
   A = no(7,no(5,no(3,[],[]),no(6,[],[])),no(12,[],[])) ,
   I = [3,5,6,7,12] ;

?- arvore2(A),inOrdem2(A,I).
   A = no(7,no(5,no(3,[],[]),no(6,[],[])),no(12,no(11,no(9,...)), ...))) ,
   I = [3,5,6,7,9,10,11,12,13,14,15,17]

?- arvore3(A), wArv(0,A), inOrdem2(A,I).
3
 4
  5
   6
   A = no(3,[],no(4,[],no(5,[],no(6,[],[])))) ,
   I = [3,4,5,6] ;
```

Uma árvore binária de busca, quando percorrida em in-ordem retorna os dados ordenados, como mostram os exemplos acima. Se criarmos uma árvore a partir de uma lista ordenada de nós, ela fica deformada (arvore3 acima), parecendo uma lista. Neste caso, a busca não faz uso da estrutura de árvore: é como se fosse feita numa lista.

A máxima eficiência na busca acontece numa árvore equilibrada. Uma árvore binária é equilibrada se cada nó tem o mesmo número de filhos à sua direita e à sua esquerda. Existem algoritmos para se construir árvores balanceadas. A versão do algoritmo de inserção apresentada aqui, para listas com dados "bem-desordenados", criará uma árvore que tenderá ao equilíbrio.

6.3.3 Remoção

A remoção de um nó de uma árvore binária ordenada é um pouco mais complexa. Abaixo temos uma árvore com o valor 10 no nó-raiz. Se removemos o valor 10, temos duas possibilidades de ajuste da árvore, para preencher o buraco deixado pelo valor 10: (i) escolher o maior valor da subárvore da esquerda (valor 5) ou (ii) escolher o menor valor da subárvore da direita (valor 15).



Abaixo, segue o algoritmo de remoção. São considerados cinco casos. Os dois primeiros tratam da remoção de um nó-folha com um único filho: simplesmente remove-se o valor e a subárvore filha ocupará o lugar do nó removido. Os outros três casos implementam uma busca (os mesmos testes do algoritmo de busca) e quando X=Y chama-se o predicado `remMin`, que remove o valor mínimo da subárvore da direita para ocupar o lugar deixado pelo nó removido.

```
1  %--(in,in,     out)
2  rem(X,no(X,[],D),D):-!.
3  rem(X,no(X,E,[]),E):-!.
4  rem(X,no(X,E,D),no(I,E,D1)):-remMin(I,D,D1). %% poe I (min) no lugar X
5  rem(X,no(I,E,D),no(I,E1,D)):-X<I,!,rem(X,E,E1).
6  rem(X,no(I,E,D),no(I,E,D1)):-X>I,!,rem(X,D,D1).
7  %
8  %-----(out,in,    out)
9  remMin(X,no(X,[],D),D):-!.
10 remMin(X,no(I,E,D),no(I,E1,D)):-!,remMin(X,E,E1).
```

Remover o menor valor de uma árvore binária, `remMin`, consiste em uma busca em profundidade: o menor é o nó que está mais à esquerda da árvore. Os predicados `rem` e `remMin` são testados como segue.

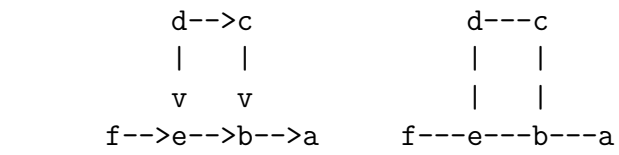
```
?- arvore1(A),inOrdem(A),nl,!,remMin(M,A,A1),inOrdem(A1),nl.
356712
56712
A = no(7, no(5, no(3, [], []), no(6, [], [])), no(12, [], []))
M = 3
A1 = no(7, no(5, [], no(6, [], [])), no(12, [], []))

?- arvore1(A),inOrdem(A),nl,!,rem(7,A,A1),rem(12,A1,A2),inOrdem(A2),nl.
356712
356
A = no(7, no(5, no(3, [], []), no(6, [], [])), no(12, [], []))
A1 = no(12, no(5, no(3, [], []), no(6, [], [])), [])
A2 = no(5, no(3, [], []), no(6, [], []))
```

Exercício 6.11 *Modifique os predicados da árvore de busca: (1) assuma que a informação de cada nó é* `reg(NOME, IDADE)`*, refaça as operações para manter os nomes em ordem alfabética, isto é, a chave de busca é o nome.*

6.4 Manipulação de grafos

Uma árvore é uma generalização de listas. Grafos são generalizações de árvores. Um grafo é definido por um conjunto de nós e um conjunto de arcos. Um arco é um relacionamento entre dois nós.

Existem dois tipos principais de grafos. No primeiro temos os arcos dirigidos num só sentido; no segundo todo arco é bidirecional, nos dois sentidos, como ilustrado abaixo.



Inicialmente vamos trabalhar com um grafo com arcos bidirecionais, representando um grafo como um conjunto de fatos. A regra `arco/2` define que cada `arco_(X,Y)` pode ser também `arco(Y,X)`, como segue.

```
1  arco_(d,c).
2  arco_(d,e).
3  arco_(c,b).
4  arco_(f,e).
5  arco_(e,b).
6  arco_(b,a).
7  arco(X,Y):-arco_(X,Y);arco_(Y,X).
8  %%
9  nodo(X):-arco_(X,_);arco(_,X).
10 nodos(L):-setof(X,nodo(X),L).
11 %%
12 cam0(X,Y,_):-arco(X,Y).
13 cam0(X,Z,C):-arco(X,Y), \+ member(Y,C), cam0(Y,Z,[Y|C]).
```

O predicado `nodos` retorna o conjunto de todos os nós do grafo numa lista. Ele é definido pelo metapredicado `setof/3`. O `setof` coleta todos os valores `X` para o predicado `nodo(X)`; o primeiro parâmetro do `setof` é a variável `X`, que define *quem são* os objetos a serem coletados; o segundo é o *predicado* que será executado para coletar os valores; e o terceiro é a lista que armazena o conjunto. Segue um teste para os predicados `nodos`.

```
?- setof(X,nodo(X),L).
   X = _ L = [a, b, c, d, e, f]

?- bagof(X, nodo(X), L)?
   X = _ L = [d, d, c, f, e, b, c, e, b|...]

?- nodos(L).
   L = [a,b,c,d,e,f]
```

O predicado `cam0/3` verifica se tem um caminho entre dois nós do grafo. Ele é baseado na regra indutiva:

- se temos um `arco(X,Y)`, então temos um caminho de X para Y;

- se temos um `arco(X,Y)` e também um caminho de Y para Z, então temos um caminho de X para Z.

Este predicado testa também se este caminho não tem ciclos. Para isso, armazenamos os arcos já visitados, na lista `C`. Quando passamos pelo `arco(X,Y)` incluímos o `X` na lista `C`. O predicado `cam0` é verdadeiro se existe um caminho; porém não retorna uma lista com o caminho.

```
?- cam0(f,a,[]).
   yes
```

Abaixo temos uma nova versão deste predicado `cam1/3`, que é baseado em um acumulador que retorna o caminho percorrido no final do processo.

```
1 cam1(X,Y,Co):-cam1a(X,Y,[X]/Co).
2 cam1a(X,Y,C/[Y|C]):-arco(X,Y).
3 cam1a(X,Z,C/Co):-arco(X,Y), \+ member(Y,C), cam1a(Y,Z,[Y|C]/Co).
4 %%
5 bag1(B):-bagof(L,cam1(f,a,L),B).
```

Podemos testar o `cam1` chamando-o diretamente ou com o predicado `bagof/3`. Por exemplo, a chamada `bagof(L,cam1(f,a,L),B)` retorna uma lista de listas de caminhos válidos de `a` até `f` em `B`. O predicado `bag1(B)` tem como corpo esta chamada. Seguem exemplos de testes:

```
?- cam1(f,a,L).
   L = [a,b,e,f] ;
   L = [a,b,c,d,e,f] ;
   no
?- bagof(L,cam1(f,a,L),B).
   L = _ ,
   B = [[a,b,e,f],[a,b,c,d,e,f]]
?- bag1(B).
   B = [[a,b,e,f],[a,b,c,d,e,f]]
```

Exercício 6.12 *Mude* `cam1` *para trabalhar com um grafo direcionado.*

6.4.1 Usando um custo em cada nodo

Em aplicações de grafos é comum associarmos um custo a cada arco. Quando há mais de um caminho, podemos escolher o caminho com menor custo.

Segue uma nova versão do grafo com custo. Neste grafo assumimos que os arcos (`arco1`) são direcionais. O predicado `cam2/4` estende a versão `cam1/3` com um novo parâmetro que devolve uma lista de custos associada ao caminho. Cada vez que um nó é acrescentado no caminho, acrescenta-se também um custo. Ambos o caminho e o custo são coletados com a técnica de acumuladores.

```
1 arco1(d,c,1).
2 arco1(d,e,2).
3 arco1(c,b,1).
4 arco1(f,e,3).
```

```
arco1(e,b,1).
arco1(b,a,1).
%
cam2(X,Y,L,V):-cam2a(X,Y,[X]/L,[]/V).
cam2a(X,Y,C/[Y|C],V/[Va|V]):-arco1(X,Y,Va).
cam2a(X,Z,C/Co,V/Vo):-arco1(X,Y,Va), \+ member(Y,C),
                      cam2a(Y,Z,[Y|C]/Co, [Va|V]/Vo).
bag2(B):-bagof(par(L,V),cam2(d,a,L,V),B).
```

Podemos testar o `cam2/4` chamando-o diretamente ou com o predicado `bag2` que acumula os possíveis caminhos numa lista.

```
?- cam2(d,a,L,V).
   L = [a,b,c,d], V = [1,1,1] ;
   L = [a,b,e,d], V = [1,1,2] ;
   no
?- bag2(B).
   B = [par([a,b,c,d],[1,1,1]),par([a,b,e,d],[1,1,2])]
```

Aqui vemos que o caminho `[a,b,c,d]` é mais econômico que o caminho `[a,b,e,d]`. A lista dos valores possui um elemento a menos que a lista do caminho, pois o valor acontece entre dois nós.

Exercício 6.13 *Faça um predicado que escolhe a melhor solução para a lista de pares devolvida por* `bag2`. *Inicialmente, faça um predicado que some os custos para cada par. Depois faça um predicado que escolha o par com menor custo.*

Exercício 6.14 *Reescreva* `cam2` *substituindo a lista* `[1,1,1]` *por uma expressão* `0+1+1+1`.

6.4.2 Trabalhando com uma lista de arcos

Uma abordagem alternativa de se trabalhar com grafos usa uma *lista de adjacências*, onde cada par da lista representa um arco, como em `[d-c,d-e,c-b,f-e,e-b,b-a]`. Aqui temos os nodos `[a,b,c,d,e,f]`.

Nesta representação para grafos, o processamento depende basicamente do predicado member. Por exemplo, a programação de um predicado que retorne um conjunto dos arcos usa uma regra com 4 casos. Dado um arco `X-Y`:

- se ambos `X` e `Y` não estão (são membros) no conjunto, então incluí-los;
- se `X` não está no conjunto, então incluí-lo;
- se `Y` não está no conjunto, então incluí-lo;
- se ambos estão no conjunto, então desprezá-los.

Esta regra é embutida no predicado `nos/2` através de uma disjunção de condicionais (separados por ";"). Cada condicional é um *if-then* (->). Na codificação desta disjunção usamos um duplo parêntese, começando em `(( \+member... ;` e terminando em `... N /No) ))`. O objetivo deste duplo parêntese é deixar mais claro, destacar, que estas disjunções estão agrupadas, como um objeto único. É um recurso sintático, poderíamos ter usado apenas um parêntese. O segundo parâmetro do predicado `nosA/2` é o acumulador que coleta os elementos do conjunto.

```
nos(X,Y):-nosA(X,[]/Y).
nosA([],X/X).
nosA([X-Y|XYs],N/No):-
     (( \+member(X,N), \+member(Y,N) -> nosA(XYs,[X,Y|N]/No);
                       \+member(Y,N) -> nosA(XYs,[  Y|N]/No);
                       \+member(X,N) -> nosA(XYs,[X  |N]/No);
                                        nosA(XYs,     N /No)  )) .
arcoXY(X-Y,L):-member(X-Y,L).
cam3(X,Y,G,C):-cam3a(X,Y,G,[X]/C).
cam3a(X,Y,G,C/[Y|C]):-arcoXY(X-Y,G).
cam3a(X,Z,G,C/Co):-arcoXY(X-Y,G), \+ member(Y,C), cam3a(Y,Z,G,[Y|C]/Co).
%%
ciclo(G):-nos(G,NG),member(X,NG),member(Y,NG),X\=Y,
          cam3(X,Y,G,_),cam3(Y,X,G,_).
cobre(C,G):-nos(G,NG),\+ (( member(N,NG),\+member(N,C) )).
```

Nesta representação de grafos em listas, o predicado arco, `arcoXY`, também é definido a partir do member: existe um arco entre dois nodos `X` e `Y` se o nodo `X-Y` é membro da lista que representa o grafo; assumimos, aqui, arcos direcionados. O predicado `cam3` é similar ao `cam2`. O que muda é a chamada ao predicado `arcoXY`.

O predicado `ciclo/1` é verdadeiro se um dado grafo tem um ciclo: dados dois nós, `X` e `Y`, um diferente do outro; se existe um caminho de X para Y e vice-versa, então existe um ciclo.

O predicado `cobre/2` é verdadeiro se um caminho passa por todos os nós do grafo: qualquer nó do grafo deve estar no caminho - não existe um nó no grafo que não está no caminho. Um caminho que cobre um grafo é chamado de Hamiltoniano.

Seguem alguns testes, para os predicados `nos/2`; `cam3/3`; `ciclo/1`; e `cobre/2`.

```
?- nos([a-b,b-c,b-d],N).
   N = [d,c,a,b]
?- cam3(a,d,[a-b,b-c,c-e,e-d,d-b,c-d,b-d],N).
   N = [d,b,a] ;
   N = [d,c,b,a] ;
   N = [d,e,c,b,a] ;
?- ciclo([a-b,b-a]).
   yes
?- ciclo([a-b,b-c,c-z,c-d]).
   no
?- ciclo([a-b,b-c,c-a,a-z]).
   yes
?- cobre([d,b,a],[a-b,b-c,c-e,e-d,d-b,c-d,b-d]).
   no
?- cobre([d,e,c,b,a],[a-b,b-c,c-e,e-d,d-b,c-d,b-d]).
   yes
```

Exercício 6.15 *Reescreva* `nosA` *sem usar o operador de condicional* `->`*. Use corte.*

Exercício 6.16 *Usando o* `bagof`*, faça um predicado que colete os nodos* `arco_(X,Y)` *numa representação em uma lista de arcos* `[X-Y, ...]`*.*

Exercício 6.17 *Faça uma versão de* cam3, *que trabalha com nós bidirecionais, como a* cam1.

Exercício 6.18 *Usando o predicado* cobre *e* cam3, *faça um predicado que devolve todos os caminhos Hamiltonianos de um grafo.*

Exercício 6.19 *Defina um predicado* ligados/3. *Num grafo dirigido, dois nós estão ligados, se existe um caminho entre eles ou se existe um terceiro nó e existe um caminho entre eles e o terceiro nó.*

Solução:

```
1 camx(X,Y,G):-cam3(X,Y,G,_),!; cam3(Y,X,G,_),!.
2 ligados(X,Y,G):-camx(X,Y,G),!.
3 ligados(X,Y,G):-nos(G,NG),!,member(Z,NG),camx(Z,X,G),camx(Z,Y,G).
```

Exercício 6.20 *Defina um predicado* conexo/1. *Um grafo dirigido é conexo se todos os seus nós estão interligados. Dois nós podem estar ligados, mas sem um caminho entre eles, por exemplo, em* [a-b, c-b, b-d, d-e] *não existe um caminho entre a e c. Sugestão: use duas negações e os predicados* nos, member *e* ligados.

```
?- ligados(a,z,[a-b, b-c, b-d, z-b, k-z]).
Yes
?- ligados(a,z,[a-b,   b-d, z-k,k-z,c-z]).
No
?- conexo([a-b, b-c, b-d, z-b]).
Yes
?- conexo([a-b,   b-d, z-k,k-z,c-z]).
No
```

Exercício 6.21 *Defina um predicado para gerar as árvores de espalhamento de um grafo. Uma árvore de espalhamento é um subgrafo conexo (um subconjunto de arcos) que não contém ciclos. Use os predicados* subconjunto, conexo *e* ciclo, *fazendo uma versão declarativa, sem usar recursividade; use os três predicados citados em uma única cláusula.*

6.5 **Diferença de listas (tópico avançado)

Diferença de listas é uma técnica usada para otimizar algoritmos de manipulação de listas, em que duas listas são concatenadas num tempo nulo, isto é, sem termos que percorrer todos os elementos de uma delas.

No início do capítulo apresentamos o predicado pre_ordem2, que é mostrado abaixo, que retorna uma lista com os nós visitados no percurso da árvore. Os valores são coletados na lista com o auxílio do predicado append.

```
1 preOrdem2(       [], []).
2 preOrdem2(no(I,E,D), [I|EDs]):- preOrdem2(E,Es),preOrdem2(D,Ds),
3                                 append(Es,Ds,EDs).
```

```
?- arvore(A),preOrdem2(A,L).
   A = no(a,no(b,no(d,[],[]),[]),no(c,no(e,no(g,[],[]),[]),no(f,[],[]))) ,
   L = [a,b,d,c,e,g,f]
```

Em `preOrdem2`, o predicado append junta o resultado de duas subárvores, a da esquerda com a da direita. O problema é que o append não é um algoritmo eficiente para concatenar duas listas. Ele sempre percorre toda a primeira lista. Assim, o tempo de concatenação é proporcional ao comprimento da primeira lista.

A técnica de *diferença de listas* possibilita remover o predicado append usando um processo que concatena duas listas num tempo zero: ligando a segunda lista como cauda da primeira. Para isso é mantida uma variável na cauda da primeira lista.

Uma diferença de listas é representada com um operador binário. Usaremos o "/", mas poderia ser qualquer outro, como o -". A lista `[a,b,c]` é representada como a diferença de listas `[a,b,c|X]/X` ou `[a,b,c|X]-X` — leia-se: *a lista* `[a,b,c|X]` *menos a lista* `X`. Não importa o valor de `X`, o resultado será sempre `[a,b,c]`. Por exemplo, se `X` é a lista vazia, teremos a diferença de listas `[a,b,c|[]]/[]` que é igual a `[a,b,c]/[]`. Com diferença de listas, uma mesma lista tem inúmeras instâncias equivalentes. Por exemplo, a lista vazia é `X/X` que pode ser instanciada por `[]/[]` ou `[a]/[a]` ou `[a,b]/[a,b]` ou ...

Em resumo, uma diferença de listas é definida pelos construtores:

- `X/X` para lista vazia, sendo `X` uma variável;
- `[C1,C2,...CN|Yc]/Yc` para uma lista não vazia; sendo `C1,C2,...CN` o conteúdo da lista e `Yc` uma variável.

Pela definição, para transformar uma lista simples, `[a,b,c]`, em uma diferença de listas, devemos acrescentar uma variável na cauda da lista simples, `[a,b,c|X]`, e depois criar o termo composto `[a,b,c|X]/X`.

Por outro lado, para obter uma lista simples a partir de uma diferença de lista, basta unificar a cauda da diferença de listas com a lista vazia. Como segue,

```
?-[a,b,c|X]/X = L/[]
   L=[a,b,c]
```

Se unificarmos a cauda de uma diferença de lista com um valor arbitrário, por exemplo, uma lista, o resultado pode não ser mais uma diferença de lista. Segue um exemplo:

```
?- X=[d,e,f], [a,b,c|X]/X=Z.
   X = [d,e,f] ,
   Z = [a,b,c,d,e,f] / [d,e,f]
```

Neste exemplo, o resultado `Z` não é mais uma diferença de listas em que cauda deve necessariamente ser uma variável. Aqui `Z` é um termo fechado, sem variáveis. Não podemos mais incluir alguma coisa na cauda num tempo zero.

6.5.1 Concatenar diferenças de listas

Numa solução com esta técnica, todas as listas devem ser representadas com diferença de listas e o resultado das operações também deve retornar uma diferença de listas. Assim, a concatenação da lista `[a,b,c]` com a lista `[d,e,f]` é representada como segue.

```
?- A/B=[a,b,c|X]/X, B/C=[d,e,f|Y]/Y, A/C=XY.
   A = [a,b,c,d,e,f|C] ,
   B = X = [d,e,f|C] ,
   C = Y = _ ,
   XY = [a,b,c,d,e,f|C] / C
```

Neste exemplo, o resultado XY é uma diferença de listas. A cauda da diferença de listas é a variável C. A pergunta do exemplo pode ser transformada no predicado append_dl para concatenar diferenças de listas, como segue.

```
1 append_dl(A/B, B/C, A/C).
```

Este predicado pode ser lido assim: A/C é o resultado da concatenação das diferenças de listas A/B e B/C. A concatenação de diferenças de listas é transitiva. Assim, podemos generalizar a regra. Se temos n diferenças de listas, de X_1/Xc_1 até X_n/Xc_n, podemos concatenar todas elas, fazendo-se $Xc_{i-1} = X_i$, para todo i. O resultado será a diferença de listas X_1/Xc_n. Toda diferença de listas tem uma variável na cauda. Estamos ligando cada cauda com a próxima lista, sem percorrer os elementos de cada lista. O predicado append_dl pode ser usado como segue:

```
?- append_dl([a,b|X]/X, [c,d|Y]/Y, R).
   X = [c,d|Y] ,
   Y = _ ,
   R = [a,b,c,d|Y] / Y
```

O append_dl pode ser usado para incluir um elemento no final de uma lista, sem percorre-la. Assim, podemos implementar eficientemente um tipo de dado fila: inserindo no final e excluindo do início. Por exemplo,

```
1 incFinal_dl(X, A/[X|Xs], A/Xs).
2 excInic_dl(X,[X|Xs]/Xc, Xs/Xc).
```

Seguem alguns testes para incluir e excluir elementos de uma fila:

```
?-incFinal_dl(a,[1,2|B]/B,X/Xc).
B = [a|Xc] ,
X = [1,2,a|Xc] ,
Xc = _

D=[1,2|X]/X,incFinal_dl(a,D,D1),incFinal_dl(b,D1,D2),D2=Z/Zc.
D = [1,2,a,b|Zc] / [a,b|Zc] ,
X = [a,b|Zc] ,
D1 = [1,2,a,b|Zc] / [b|Zc] ,
D2 = [1,2,a,b|Zc] / Zc ,
Z = [1,2,a,b|Zc] ,
Zc = _

?- excInic_dl(X,[1,2,3|Y]/Ys,Z/Zs).
X = 1 ,
Y = _ ,
Ys = Zs = _ ,
Z = [2,3|Y]
```

O predicado incFinal_dl foi derivado em uma versão inicial baseada no append para diferenças de listas. O append_dl, que é um fato, é removido por um processo de reescrita e simplificação, como segue.

```
   incFinal_dl(X, DLi, DLo):-append_dl(DLi,[X|Xs]/Xs,DLo).
                          %% append_dl(A/B, B/C, A/C).
=> incFinal_dl(X, A/B, A/C):-B=[X|Xs],C=Xs.
=> incFinal_dl(X, A/[X|Xs], A/Xs).
```

Para concluir, devemos dizer que transformar uma lista simples numa diferença de lista é uma operação cara. Devemos percorrer toda a lista e incluir um elemento no final, como segue.

```
1    transforma(L,X/Xc):-append(L,[Xc],X).
```

Por outro lado, para transformar uma diferença de listas numa lista simples é só unificar a cauda com a lista vazia, num tempo zero, como mostrado anteriormente.

Na prática, normalmente, não é necessário transformar listas simples em diferenças de listas. Um programa com diferenças de listas constrói passo a passo as diferenças de listas incluindo um elemento por vez na cabeça, como exemplificaremos no percurso em árvore binária, reescrito com diferenças de listas.

6.5.2 Usando diferenças de listas no percurso de árvores

Agora, podemos reescrever os três algoritmos de caminhamento em árvore usando a técnica de diferença de listas. Abaixo, apresentamos a solução. O trabalho de reescrita é deixado como exercício.

```
1 pre_dl([], X/X).
2 pre_dl(no(I,E,D),[I|Is]/EDs):-pre_dl(E,Is/Es),pre_dl(D,Es/EDs).
3 pos_dl([], X/X).
4 pos_dl(no(I,E,D),  Is/EDs):-pos_dl(E,Is/Es),pos_dl(D,Es/[I|EDs]).
5 in_dl([],X/X).
6 in_dl(no(I,E,D), Is/EDs):-in_dl(E,Is/[I|Es]),in_dl(D,Es/EDs).
```

Para se usar diferenças de listas não é necessário criar listas simples e transformá-las em diferenças de listas, nem é necessário usar explicitamente o append_dl.

```
?- arvore(A),preOrdem2(A,L).
A = no(a,no(b,no(d,[],[]),[]),no(c,no(e,no(g,[],[]),[]),no(f,[],[]))) ,
L = [a,b,d,c,e,g,f]

?- arvore(A),pre_dl(A,P/[]),in_dl(A,I/[]),pos_dl(A,S/[]).
A = no(a,no(b,no(d,[],[]),[]),no(c,no(e,no(g,[],[]),[]),no(f,[],[]))) ,
P = [a,b,d,c,e,g,f] ,
I = [d,b,a,g,e,c,f] ,
S = [d,b,g,e,f,c,a]
```

Cada resultado de um caminhamento é uma lista simples, obtida com uma lista vazia na cauda da diferença de listas. De certo modo, a diferença de listas fica embutida no código. Externamente, parece um predicado normal, retornando uma lista simples. Como resultado, o predicado pre_dl tem um código mais simples e mais eficiente que o predicado equivalente preOrdem2. Compare-os:

```
1 pre_dl([], X/X).
2 pre_dl(no(I,E,D), [I|Is]/EDs):-pre_dl(E,Is/Es),pre_dl(D,Es/EDs).
3 preOrdem2( [], []).
4 preOrdem2(no(I,E,D), [I|EDs]):-
5             preOrdem2(E,Es),preOrdem2(D,Ds),append(Es,Ds,EDs).
```

Em resumo, diferença de listas é uma técnica que visa à implementação eficiente da concatenação de listas. Apesar de ser necessário trabalhar com o operador de diferença de listas e duas variáveis no lugar de uma, com diferenças de listas as operações explícitas de append são removidas.

Exercício 6.22 *Mostre como obter o algoritmo* `pre_dl` *com a técnica diferença de listas. Inicie com* `pre_ordem2` *e termine com* `pre_dl`*.*

Exercício 6.23 *Mostre como transformar o* `in_ordem2/2` *em* `in_dl`*.*

CAPÍTULO 7

Animação de programas

Por animação de programas, entendemos o estudo de técnicas que implementam uma interface amigável e intuitiva com o usuário. Estas técnicas permitem representar na tela do computador os modelos abstratos embutidos nos algoritmos. Por exemplo, a solução para o problema das torres de Hanoi tipicamente é uma lista de movimentos. Com animação busca-se simular os movimentos na tela do computador.

Neste capítulo trabalhamos com dois problemas: torres de Hanoi e jogo-da-velha. A solução do primeiro problema é um algoritmo recursivo, enquanto a solução do segundo problema faz uso da programação por regras declarativas, as quais codificam as estratégias de jogo. A solução destes problemas envolve o uso integrado das diversas técnicas de programação estudadas até aqui. As torres de Hanoi resulta num programa pequeno, quase 50 linhas de código. O jogo-da-velha resulta em programas de aproximadamente 100 linhas de código, um programa de complexidade média.

7.1 Torres de Hanoi

Torres de Hanoi é um problema clássico usado para se apresentar programas recursivos. Este problema surgiu de uma lenda que aconteceu em um mosteiro aos arredores de Hanoi. Deus, logo após ter criado o mundo, deu uma missão aos monges do mosteiro, dizendo que quando a missão estivesse concluída o mundo iria acabar.

A missão foi enunciada assim: Existem três hastes, na primeira delas existe uma torre com 64 discos de ouro. O objetivo é mover os 64 discos para a haste do centro usando a terceira haste como haste auxiliar. Na movimentação existem apenas duas regras:

- os discos são movidos de uma haste para outra, um a um; e
- nunca um disco maior pode estar sobre um disco menor.

A melhor solução para um problema de N discos leva $2^N - 1$ movimentos. Portanto, se todos os movimentos forem corretos são necessários $2^{64} - 1 = 1.84467e + 19$ movimentos. Este número

corresponde a mais de 18 quintiliões de movimentos[1].

Para efeito de programação, o problema é enunciado para 2, 3 ou 5 discos. Abaixo, temos a solução para o problema com 2 discos, em que são usados $2^2 - 1 = 3$ movimentos.

```
?-hanoi(2).
  esq    centro   dir
   |       |       |        (Estado Inicial)
   =       |       |
  ===      |       |
"""""""" """""""" """""""

   |       |       |
   |       |       |        (Move um disco da esq para a dir)
  ===      |       =
"""""""" """""""" """""""

   |       |       |
   |       |       |        (Move um disco da esq para o centro)
   |      ===      =
"""""""" """""""" """""""

   |       |       |
   |       =       |         (Move um disco da dir para o centro)
   |      ===      |
"""""""" """""""" """""""
```

A estratégia para resolver o jogo consiste num procedimento `hanoi(N,A,B,C)` que move N discos de A(esq) para B(centro) usando C(dir) como haste auxiliar:

- se `N=0` então o problema está resolvido;
- se não, o problema de mover N discos da haste A para B, usando a C como auxiliar, é decomposto em:
  - mover M=N-1 discos da haste A para a haste C usando a haste B como auxiliar;
  - depois mover um disco (o do fundo da pilha) da haste A para B; e
  - depois mover os M discos da haste C para a haste B usando a haste A como auxiliar.

Por ser um procedimento recursivo, ele só realmente executa o movimento base a cada vez que é chamado. O movimento base move um disco de uma pilha para outra. O procedimento inicial codificando esta estratégia da solução é apresentado abaixo.

```
1 hanoi0(N) :- write('Mover '), write(N),
2              write(' discos da Esq para o Centro'),nl,
3              write(' usando Dir como pilha auxiliar:'),nl,
4              moveXY0(N,esq,centro,dir).
5 moveXY0(0,_,_,_) :- !.
6 moveXY0(N,A,B,C) :- !, M is N-1,
```

[1] Isto é uma eternidade toda. Não precisamos perder o sono preocupados com o fim do mundo. Os monges devem ainda estar trabalhando para cumprir a missão.

```
7         moveXY0(M,A,C,B),
8         exibeMovXY0(A,B),get0(_),
9         moveXY0(M,C,B,A).
10 exibeMovXY0(X,Y) :- !,nl, write(' Move um disco de '), write(X),
11                           write(' para '), write(Y), nl.
```

Segue a execução deste procedimento, para uma torre de três discos. São executados sete movimentos.

```
?- hanoi0(3).
 Mover 3 discos da Esq para o Centro
    usando Dir como pilha auxiliar:
 Move um disco de esq para centro
 Move um disco de esq para dir
 Move um disco de centro para dir
 Move um disco de esq para centro
 Move um disco de dir para esq
 Move um disco de dir para centro
 Move um disco de esq para centro
```

Nesta solução incluímos um comando `get0(_)` para rodar o programa passo a passo; após exibir o movimento, digitamos um <enter> para executar o próximo movimento. Nosso objetivo, agora, é estender esta solução para exibir passo a passo o movimento das torres como apresentado no início da seção.

7.1.1 Salvando e atualizando o estado das hastes

Para animar o procedimento temos que definir uma estrutura de dados para representar as três hastes. E a cada movimento, por exemplo, *Move um disco da esquerda para o centro*, devemos atualizar a estrutura de dados. Além disso, devemos ter um predicado que exibe a estrutura de dados logo após ser atualizada.

Para salvar o estado do jogo, usamos dois fatos:

- `nElem(N)` salva o número de discos passados como parâmetro no jogo; e
- `estado(E,C,D)` salva o valor das três hastes.

Por exemplo, se chamamos `hanoi(3)`, são criados os fatos: `nElem(3)` e `estado( [1,2,3], [], [])`. A lista `[1,2,3]` representa um torre (pilha) de três discos sendo que o menor, o 1, está no topo e o maior, o 3, está no fundo da pilha. O primeiro movimento da solução *Move um disco da esquerda para o centro* atualiza o estado para `estado([2,3],[1],[])`.

Os fatos `nElem/1` e `estado/3` são declarados como dinâmicos, pois eles são atualizados pelos predicados `mknElem/1` e `mkEstados/3`, durante a execução do programa. Cada vez que estes predicados são executados, eles criam um novo fato. Neles, o `retract` remove a versão velha e o `assert` cria a versão nova. Segue abaixo, o código destes predicados.

```
1 :-dynamic(nElem/1).
2 :-dynamic(estado/3).
3 nElem(0).
4 estado([],[],[]).
```

```
5  mkEstado(E,C,D):- estado(X,Y,Z), retract(estado(X,Y,Z)),
6                                   assert(estado(E,C,D)).
7  mknElem(N)     :- nElem(E),retract(nElem(E)), assert(nElem(N)).
```

É relativamente simples desenhar torres e hastes com um tamanho fixo, por exemplo, três. Porém, queremos desenhar torres de qualquer tamanho acima de um (dependendo dos limites da tela, até um máximo de 10 ou 15 discos).

Para isso definimos alguns predicados de entrada e saída. O `writeN/2` escreve uma repetição do caractere passado como parâmetro — serve para escrever um disco de tamanho N. O predicado `wrtDisco` escreve um disco de tamanho D para uma haste de tamanho N ou apenas um elemento da haste. O predicado `wrtECD` escreve as três hastes com discos, linha por linha.

Para facilitar a escrita das hastes usamos um predicado `ajustaN`, que permite transformar um `estado([1,2,3],[],[])` em um novo `estado([0,1,2,3], [0,0,0,0], [0,0,0,0])`. Esta representação (normalizada) do estado é usada só para o desenho: cada zero corresponde a um elemento de uma haste. Mesmo quando a pilha está cheia, desenhamos um elemento de haste sobre o topo, para parecer mais realista. O predicado `fazLista` é usado para inicializar a torre do jogo, por exemplo, para um jogo `hanoi(5)` é criada uma lista que representa uma torre com 5 discos `[1,2,3,4,5]`. Segue o código dos predicados que são usados no desenho do estado.

```
1  writeN(N,_) :- N<1,!.
2  writeN(N,C) :- write(C), N1 is N-1, writeN(N1,C).
3  %%
4  wrtDisco([0|Ds],N,Ds) :- T is (N*2+1) // 2 ,tab(T), write('|'), tab(T).
5  wrtDisco([D|Ds],N,Ds) :- T is (((N*2+1) - (D*2-1)) // 2), tab(T),
6                           writeN(D*2-1,'='), tab(T).
7  wrtDiscoECD(_,_,_,0,_) :- !.
8  wrtDiscoECD(E,C,D,N,M):- wrtDisco(E,M,E2), tab(1),
9                           wrtDisco(C,M,C2), tab(1),
10                          wrtDisco(D,M,D2), nl,
11             N1 is N-1, wrtDiscoECD(E2,C2,D2,N1,M).
12 wrtHastes(E,C,D,T) :-
13     nl, ajustaN(E,T,E2), ajustaN(C,T,C2), ajustaN(D,T,D2),
14     wrtDiscoECD(E2,C2,D2,T,T), Z is T*2+1,
15     writeN(Z,'"'), tab(1), writeN(Z,'"'), tab(1),writeN(Z,'"'),nl.
16 %%
17 ajustaN(L,N,R) :- length(L,Tam), fazLista2(N-Tam,0,L/R).
18 fazLista2(N,_,Li/Li) :- N=<0, !.
19 fazLista2(N,V,Li/Lo) :- N1 is N-1, fazLista2(N1,V,[V|Li]/Lo).
20 %%
21 fazLista(M,L):-!, fazLista(M,1,L).
22 fazLista(M,N,[]) :- N>M,!.
23 fazLista(M,N,[N|R]) :- N1 is N+1, fazLista(M,N1,R).
```

Segue a execução de alguns dos predicados codificados acima: `writeN/2`, `ajustaN/3` e `fazLista/2`. A execução dos outros pode ser vista na animação apresentada no início da seção.

```
?- writeN(10,'=').
==========
?- ajustaN([1,2],4,L).
   L = [0, 0, 1, 2]
```

```
?- fazLista(5,L).
   L = [1, 2, 3, 4, 5]
```

O predicado `hanoi/1` define a versão animada do programa. Aqui, é necessário inicializar o estado do jogo e desenhá-lo na tela. Este predicado chama o predicado principal do programa `moveXY`, que é praticamente o mesmo da versão inicial `moveXY0`. O que muda é a presença de um predicado `novoXYEst/2` que atualiza o estado antes de exibir o movimento com `exibeMovXY/2`.

O predicado `novoXYEst/2`, abaixo, está definido para todas as combinações de hastes (esquerda, centro, direita), duas a duas. Mover uma haste da esquerda para o centro centro é mover o valor da cabeça da lista da esquerda para a lista do centro, por exemplo, de `estado([3,4], [5], [1,2])` para `estado([4],[3,5],[1,2])`.

```
1  hanoi(N) :- fazLista(N,L), mkEstado(L,[],[]), mknElem(N),
2            estado(Xs,Ys,Zs), write(' Estado Inicial'), nl,
3            wrtHastes(Xs,Ys,Zs,N+1), moveXY(N,e,c,d).
4  moveXY(0,_,_,_) :- !.
5  moveXY(N,A,B,C) :- !, M is N-1,
6         moveXY(M,A,C,B),
7         novoXYEst(A,B), exibeMovXY(A,B),get0(_),
8         moveXY(M,C,B,A).
9  exibeMovXY(X,Y) :- !, nl, write(' Move um disco de '),
10                  write(X),  write(' para '), write(Y), nl,
11              estado(Xs,Ys,Zs), nElem(T), wrtHastes(Xs,Ys,Zs,T+1).
12 novoXYEst(e,c) :- estado([X|Xs],Ys,Zs), mkEstado(Xs,[X|Ys],Zs).
13 novoXYEst(e,d) :- estado([X|Xs],Ys,Zs), mkEstado(Xs,Ys,[X|Zs]).
14 novoXYEst(d,e) :- estado(Xs,Ys,[Z|Zs]), mkEstado([Z|Xs],Ys,Zs).
15 novoXYEst(d,c) :- estado(Xs,Ys,[Z|Zs]), mkEstado(Xs,[Z|Ys],Zs).
16 novoXYEst(c,d) :- estado(Xs,[Y|Ys],Zs), mkEstado(Xs,Ys,[Y|Zs]).
17 novoXYEst(c,e) :- estado(Xs,[Y|Ys],Zs), mkEstado([Y|Xs],Ys,Zs).
```

Este exemplo ilustra uma solução baseada num problema do tipo transição de estados — um estado é mantido e atualizado a cada passo ou transição. Este exemplo mostra, também, como controlar os movimentos, passo a passo, com um comando de leitura, o `get0/1` que lê qualquer caractere, mesmo os de controle que não são visíveis na tela.

Na versão inicial, a solução do problema é simples e compacta (são 11 linhas de código). Na versão com animação na tela, temos um código várias vezes maior que o código inicial:

- seis linhas no fragmento 1 – salva o estado;
- mais 23 linhas fragmento 2 – escreve os discos;
- mais 17 linhas do fragmento principal;
- juntando os três fragmentos dá um total 46 linhas.

Agora podemos falar sobre animação de programas, que é o título deste capítulo, como uma técnica que permite visualizar na tela do computador, de uma forma amigável, o comportamento de um programa. Esta técnica tem várias aplicações, uma delas pode ser para o ensino de algoritmos, outra pode ser para programas como jogos interativos nos quais o computador faz o papel de um dos jogadores, por exemplo, num jogo de xadrez contra o computador.

Exercício 7.1 *Faça uma nova versão do programa animado sem salvar os estados no banco de dados do Prolog. Acrescente um ou mais parâmetros nos predicados, passando o estado a ser atualizado a cada passo.*

Exercício 7.2 *No capítulo sobre estrutura de dados existe uma versão de um programa que cria uma árvore binária. Anime-a, para criar a cada passo um nó da árvore.*

7.2 Jogo-da-Velha

O jogo-da-velha consiste em um tabuleiro de 3x3, como mostrado abaixo. É um jogo entre dois adversários: normalmente um deles marca uma "cruz"e o outro marca uma "bola"em uma das casas do tabuleiro. Um dos jogadores começa o jogo marcando o tabuleiro e em seguida passa a vez ao adversário. O jogo termina com um vencedor ou com empate. Um jogador vence quando faz três "cruzes"(ou bolas) alinhadas, no sentido horizontal, vertical ou diagonal. Quando já não existem possibilidades de alguém fazer três marcações alinhadas, o jogo empata. Seguem alguns exemplos de marcações de estados de jogos.

```
%            exemplo:   vence:   empates:
%  1|2|3      x| |      o|o|x    x|o|x   x|x|o
%  ------     -----     -----    -----   -----
%  4|5|6       |o|      x|o|     x|o|    o|o|x
%  ------     -----     -----    -----   -----
%  7|8|9       | |      x|x|o    o|x|o   x| |
```

Aqui, estudamos uma versão do jogo no qual um dos jogadores é o computador. A representação do tabuleiro é um termo com nove argumentos, cada argumento é uma casa do tabuleiro. Uma posição não marcada (vazia) é representada por uma variável livre. Quando uma posição é marcada, o valor cruz (`x`) ou bola (`o`) é unificado com a variável livre que está na posição correspondente.

Para testar a situação de uma posição no tabuleiro são usados quatro predicados: `cruz(N,Tab)`, `bola(N,Tab)`, `vazio(N,Tab)` e `cheio(N,Tab)`. Estes predicados são definidos a partir da primitiva `arg(N,Termo,V)`, que retorna o argumento `N` de um Termo, como o valor `V`. Este valor pode ser qualquer objeto, inclusive uma variável. Este predicado é também usado para se realizar a marcação de uma posição no tabuleiro (foi renomeado como `moveC/3`).

Os predicados `var/1` e `nonvar/1` são usados, respectivamente, para testar se um termo é ou não uma variável livre. Por exemplo, o predicado `cruz/2`, abaixo, é lido como: se o argumento `N` de `Tab`(uleiro) não é uma variável e é igual a `x`, então temos uma cruz na posição `N`. O predicado `vazia/2` é verdadeiro se a posição `N` é uma variável livre.

```
1 %% tab(_,_,o, _,_,_, _,_,_)  % tabuleiro como um termo
2 %% tab(1 2 3  4 5 6  7 8 9)
3 %%
4 moveC(N,Tab,C):- arg(N,Tab,C).
5  cruz(N,Tab)  :- arg(N,Tab,V), nonvar(V), V=x.
6  bola(N,Tab)  :- arg(N,Tab,V), nonvar(V), V=o.
7 vazia(N,Tab)  :- arg(N,Tab,V), var(V).
8 cheia(N,Tab)  :- \+ vazia(N,Tab).
```

Abaixo exemplificamos o uso destes predicados.

```
?- T=tab(_,_,o, _,_,_, _,_,_), vaziaN(3,T).
   No
?- T=tab(_,_,o, _,_,_, _,_,_), cheia(3,T).
   Yes
?- T=tab(_,_,o, _,_,_, _,_,_), bola(3,T).
   Yes
?- T=tab(_,_,o, _,_,_, _,_,_), cruz(3,T).
   No
```

Um jogador vence quando, após um movimento, três marcações iguais (cruzes ou bolas) estão alinhadas. Para saber se três marcações estão alinhadas temos o predicado `emlinha3`, que é definido para as posições horizontais (3 possibilidades), verticais (3 possibilidades) e diagonais (2 possibilidades).

```
1  emlinha3([1,2,3]). %% horiz  %% posições em linha
2  emlinha3([4,5,6]).
3  emlinha3([7,8,9]).
4  emlinha3([1,4,7]). %% vert
5  emlinha3([2,5,8]).
6  emlinha3([3,6,9]).
7  emlinha3([1,5,9]). %% diag
8  emlinha3([3,5,7]).
9  %%
10 vence(T,venceu(cruz)):- emlinha3([A,B,C]), cruz(A,T),cruz(B,T),cruz(C,T),!.
11 vence(T,venceu(bola)):- emlinha3([A,B,C]), bola(A,T),bola(B,T),bola(C,T),!.
```

O predicado `vence/2` testa se três valores em linha são marcados com cruz ou bola. Ele devolve um termo indicando quem venceu: `venceu(cruz)` ou `venceu(bola)`. Poderíamos devolver só o valor `cruz` ou `bola`, mas o termo carrega uma informação mais completa: na pergunta "X vence?", ele responde "Cruz venceu".

É necessário um predicado para saber se o jogo terminou: "game over". O jogo termina se um dos jogadores vence ou numa situação de empate. É empate quando nenhum dos dois jogadores pode ganhar. Isto significa que, dado um estado do jogo, se preenchermos as posições livres com bolas (ou com cruzes) não teremos três bolas (ou cruzes) em linha.

Para definirmos o `empate/1` usamos o predicado `preenche` que sistematicamente seleciona uma posição vazia (1..9) e preenche a posição com bola (ou cruz). O predicado termina quando não existem mais posições vazias.

```
1  preenche(XO,T):- member(X,[1,2,3,4,5,6,7,8,9]),
2                   vazia(X,T),moveC(X,T,XO),!,preenche(XO,T).
3  preenche(XO,T).
4  empate(T):- preenche(o,T),\+ vence(T,_),!,preenche(x,T),\+ vence(T,_).
```

Abaixo, chamamos o predicado `empate`. A primeira pergunta verifica que não é empate um jogo num tabuleiro com apenas duas posições preenchidas. A segunda pergunta verifica o empate num tabuleiro com apenas uma posição livre. Após um teste de empate o tabuleiro fica totalmente preenchido. Quando o empate falha, o tabuleiro permanece como estava. Sempre que um predicado falha, tudo o que foi feito sobre os dados dos seus parâmetros é desfeito.

```
?- (T=tab(o,_,x,  _,_,_,  _,_,_), empate(T)).
   No
?- (T=tab(x,o,x,  x,o,_,  o,x,o), empate(T)).
   T = tab(x,o,x,x,o,o,o,x,o)
   Yes
```

7.2.1 Estratégia de jogo para vencer (ou não perder)

Mesmo num jogo simples como o jogo-da-velha é necessário usar uma estratégia para não se perder o jogo pelo adversário. Por exemplo, suponha que eu jogo com a cruz e é a minha vez de jogar:

- primeiro, se já tenho duas cruzes alinhadas e a terceira posição da linha está livre, então devo jogar nesta posição para ganhar (prioridade um é ganhar);
- segundo, se o adversário tem duas bolas alinhadas e a terceira está livre, tenho que me defender, jogando na posição livre para ele não ganhar (prioridade dois, não perder na próxima jogada);
- terceiro, se não posso ganhar e nem preciso me defender, então devo escolher a melhor posição para a jogada: por exemplo, primeiro vejo se a posição 5 (do centro) está livre; se sim, jogo nela; se não, vejo se um canto está livre e jogo nele; se não, jogo em qualquer posição livre.

O primeiro e segundo casos são similares. Eles podem ser programados com um predicado que verifica se existe uma ameaça (dois em linha, com a terceira posição vazia). Existem dois casos para o predicado `ameaca`: um para bola e outro para cruz. Seguem dois testes para o predicado que verifica uma ameaça:

```
1 ameaca(Tab,CB,W) :- emlinha3(Pos),ameaca(CB,Pos,Tab,W),!.
2 %%
3 ameaca(cruz,[A,B,C],T,A) :- vazio(A,T),cruz(B,T),cruz(C,T).
4 ameaca(cruz,[A,B,C],T,B) :- vazio(B,T),cruz(A,T),cruz(C,T).
5 ameaca(cruz,[A,B,C],T,C) :- vazio(C,T),cruz(A,T),cruz(B,T).
6 %%
7 ameaca(bola,[A,B,C],T,A) :- vazio(A,T),bola(B,T),bola(C,T).
8 ameaca(bola,[A,B,C],T,B) :- vazio(B,T),bola(A,T),bola(C,T).
9 ameaca(bola,[A,B,C],T,C) :- vazio(C,T),bola(A,T),bola(B,T).
```

```
?- T=tab(_,o,o, _,x,_, _,_,_),ameaca(T,cruz,P).
   No
?- T=tab(_,o,o, _,x,_, _,_,_),ameaca(T,bola,P).
   T = tab(_G399, o, o, _G402, x, _G404, _G405, _G406, _G407)
   P = 1 ;
   No
```

No predicado `escolheMov/2` passamos o tabuleiro e o jogador; ele retorna um novo tabuleiro com a jogada feita pelo jogador (computador ou oponente). Não é necessário passar um tabuleiro de entrada e outro de saída, porque as posições livres do tabuleiro são variáveis. Assim, fazer uma jogada numa determinada posição consiste em unificar uma variável com um valor.

Agora, estamos em condições de codificar a estratégia de jogo exposta acima — no predicado `escolheMov`. Assumimos que o computador joga com bola e que o adversário joga com cruz.

```
escolheMov(T, computador):-
      ameaca(T,bola,W),!,moveC(W,T,o),! %% vence
   ; ameaca(T,cruz,W),!,moveC(W,T,o),! %% defesa
   ; vazia(5,T),moveC(5,T,'o'),!
   ; chute(9,W),member(W,[1,3,7,9]),vazia(W,T),moveC(W,T,'o'),!
   ; chute(9,W),member(W,[2,4,6,8]),vazia(W,T),moveC(W,T,'o'),!.
```

Para saber se o computador pode vencer na próxima jogada perguntamos `ameaca(T, bola, W)`, `T` é o tabuleiro e `W` é a posição em que acontece a ameça. Se o predicado `ameaca` for verdadeiro, então usamos o predicado `moveC(W,T,o)` para jogar uma bola na posição `W`. Se não podemos vencer, temos que verificar se o adversário está nos ameaçando, com `ameaca(T,cruz,W)`. Neste caso, também, temos que jogar na posição `W`.

Caso não estejamos ameaçados, seguimos com a estratégia de jogo: primeiro no centro; segundo num dos cantos (posições 1,3,7,9); e, terceiro numa outra posição qualquer (2,4,6,8).

Para jogar bola num canto, faz-se a pergunta:

```
?- member(W,[1,3,7,9]),vazio(W,T),moveC(W,T,'o').
```

Aqui o predicado member seleciona em W um dos valores dos cantos da lista [1,3,7,9]; se o canto W está vazio, joga-se em W; caso contrário, por retrocesso, é escolhido o próximo valor e o processo se repete. Se nenhuma das posições dos valores da lista está livre, a pergunta falha. Este mesmo processo é válido para jogar nas posições pares [2,4,6,8]; o que muda é a lista com as posições.

Esta solução para jogar nos cantos funciona. Mas ela sempre escolhe o primeiro valor da lista dos cantos. Uma solução mais inteligente pode escolher um valor da lista aleatoriamente a cada jogada. Para isso podemos usar um predicado `chute` que funciona como segue.

```
?- chute(5,X).
   X = 5;  X = 1;  X = 4 ;  X = 2 ; X = 2 ; X = 1 ; ...
```

Este predicado `chute` é implementado a partir de um predicado que gera números aleatórios. Para fazer retrocesso no chute, gerando sucessivos chutes (por exemplo, até acertar um dos cantos), usa-se o predicado `repeat`. Sem o `repeat`, só um valor de chute é gerado e, se a posição deste valor não estiver vazia ou não estiver na lista dos cantos, o corpo da cláusula que escolhe o movimento falha.

```
chute(N,S):- repeat, S is random(N)+1.
```

7.2.2 O jogo do adversário do computador

Um jogo entre um computador e um adversário (ou oponente) consiste em um ciclo de rodadas, em que um jogador inicia o jogo e passa a vez ao seu adversário. Na tela do computador, a cada jogada, um novo tabuleiro deve ser exibido, considerando o derradeiro movimento.

Segue o predicado principal `jogo` que inicializa um tabuleiro vazio, exibe o tabuleiro e chama o oponente do computador para jogar, com o predicado `jogar/2`. Este predicado, recursivo na

cauda, executa o ciclo de jogadas: a cada jogada o parâmetro jogador é trocado com o predicado proximo: o próximo do computador é o oponente e vice-versa. Em cada ciclo existe a chamada ao predicado escolheMov que é chamado para o jogador da vez. Após a escolha do movimento, é exibido o tabuleiro com o estado atualizado. Ao mesmo tempo, após cada jogada é verificado se o jogo terminou com o predicado gameOver. O jogo termina com um vencedor ou com empate. Os predicados vence e empate já foram comentados. O predicado gameOver também retorna um resultado que pode ser venceu(bola), venceu(cruz) ou empate.

```
jogo :- T = tab(A,B,C,  D,E,F,   G,H,I),
        exibeJogo(T, inicio),
        jogar(T, oponente).
jogar(T, Jogador):- gameOver(T,Result),!,msgFim(Result).
jogar(T, Jogador):- escolheMov(T, Jogador),!,
                    exibeJogo(T, Jogador),!,
                    proximo(Jogador, Oponente), !,
                    jogar(T, Oponente).
proximo(computador,oponente).
proximo(oponente,computador).
%%
exibeJogo(T,J):- write('jogou:'),write(J),desenha(T).
msgFim(X):-write('GAME OVER:'), write(X),nl,nl.
%%
gameOver(T,V) :- vence(T,V).
gameOver(T,empate) :- empate(T).
```

Para completar a apresentação do predicado escolheMov é necessário descrever como é feita uma jogada para o adversário do computador. O predicado escolheMov(T,oponente) testa com o predicado vaziaN se a posição lida corresponde a uma posição vazia no tabuleiro; se este teste falha, uma mensagem informa que a jogada não é válida; e é repetida a chamada ao predicado de escolha do movimento.

```
testaOk(P,Tab) :- vazia(P,Tab),arg(P,Tab,'x'),!;
                  write('Jogada invalida,tente outra!'),nl,
                  escolheMov(Tab,oponente).
escolheMov(T, oponente):- write('jogue (1..9):'),nl,
                          read(P), testaOk(P,T).
```

7.2.3 Desenhar o tabuleiro

Finalmente, falta descrever o predicado que desenha o tabuleiro. Desenhar o tabuleiro a partir do termo tab/9 não apresenta maiores dificuldades. Para escrever cada linha do tabuleiro definimos um predicado wrtLinha para o qual são passados como parâmetros os valores para as linhas horizontais (1,2,3), (4,5,6) e (7,8,9).

```
%% desenha o tabuleiro
wrtLinha(X,Y,Z,T):-arg(X,T,V1), wVal(V1),write('|'),
                   arg(Y,T,V2), wVal(V2),write('|'),
                   arg(Z,T,V3), wVal(V3),nl.
```

```
wVal(X):- var(X)->write(' ');write(X).
desenha(T) :- nl, tab(7),wrtLinha(1,2,3,T), tab(7),write('------'),nl,
                  tab(7),wrtLinha(4,5,6,T), tab(7),write('------'),nl,
                  tab(7),wrtLinha(7,8,9,T).
```

Segue a execução do predicado `desenha`.

```
?-T=tab(_,_,o, _,x,_, _,_,_), desenha(T).
      | |o
     ------
      |x|
     ------
      | |
```

Diferentemente do problema das torres de Hanoi apresentado na seção anterior, o jogo-da-velha não guarda o estado de cada jogada como um termo no banco de dados do Prolog. Sempre que trabalhamos com um ciclo de transições é possível fazer a escolha entre: (1) usar o estado como um parâmetro a ser passado entre os predicados do programa; ou (2) usar um termo global armazenado no banco de dados do Prolog.

Nesta versão do jogo usamos um parâmetro nos predicados

para representar o estado do jogo. A cada jogada uma variável livre do termo é substituída por um valor. No final do jogo o termo está quase cheio — quase porque o jogo normalmente termina antes de se preencher todas as casas. Por ser um termo com variáveis livres foi necessário apenas passar como parâmetro de entrada o tabuleiro (sem retorná-lo a cada modificação).

Ao contrário, no problema das torres, a cada movimento um novo estado é criado. Portanto, para fazer uma versão do problema das torres sem usar o banco de dados, é necessário passar o estado como entrada e também como saída, nos predicados que executam os movimentos. Entra o estado atual e sai o novo estado; por exemplo, entra `estado([1,2],[],[])` e sai `estado([2],[1],[])`.

7.2.4 A listagem do programa todo

Vale a pena ver o código do programa como um todo. Juntando todos os fragmentos de código em um programa, obtemos um texto com aproximadamente 90 linhas de código-fonte. Destas mais que um quarto e menos que um terço são linhas de comentários.

No meio do programa temos várias perguntas que servem para testar os predicados codificados. Normalmente estas perguntas estão próximas do código do predicado. São exemplos de usos do predicado.

```
%% Jogo-da-velha
%% ------------------------------------------------------------
%% tab(_,_,o, _,_,_, _,_,_)  % tabuleiro como um termo
%% tab(1 2 3  4 5 6  7 8 9)
%%
emlinha3([1,2,3]). %% horiz  %% posições em linha
emlinha3([4,5,6]).
emlinha3([7,8,9]).
emlinha3([1,4,7]). %% vert
emlinha3([2,5,8]).
```

```
emlinha3([3,6,9]).
emlinha3([1,5,9]). %% diag
emlinha3([3,5,7]).
%%
moveC(N,Tab,C):- arg(N,Tab,C).
 cruz(N,Tab)  :- arg(N,Tab,V), nonvar(V), V=x.
 bola(N,Tab)  :- arg(N,Tab,V), nonvar(V), V=o.
vazia(N,Tab)  :- arg(N,Tab,V), var(V).
cheia(N,Tab)  :- \+ vazia(N,Tab).
%%-------------------------------------------------------
gameOver(T,V) :- vence(T,V).
gameOver(T,empate) :- empate(T).
vence(T,venceu(cruz)):- emlinha3([A,B,C]), cruz(A,T),cruz(B,T),cruz(C,T),!.
vence(T,venceu(bola)):- emlinha3([A,B,C]), bola(A,T),bola(B,T),bola(C,T),!.
preenche(XO,T):- member(X,[1,2,3,4,5,6,7,8,9]),
                 vazia(X,T),moveC(X,T,XO),!,preenche(XO,T).
preenche(XO,T).

empate(T):- preenche(o,T),\+ vence(T,_),!,preenche(x,T),\+ vence(T,_).
%% ?- (T=tab(o,_,x, _,_,_ ,_,_,_), empate(T)).
%% ?- (T=tab(o,o,x, x,x,o,o,o,x), empate(T)).
%%-------------------------------------------------------------------
testaOk(P,Tab) :- vazia(P,Tab),arg(P,Tab,'x'),!;
                  write('Jogada invalida,tente outra!'),nl,
                  escolheMov(Tab,oponente).
escolheMov(T, oponente):- write('jogue (1..9):'),nl,
                          read(P), testaOk(P,T).
escolheMov(T, computador):-
     ameaca(T,bola,W),!,moveC(W,T,o),! %% vence
   ; ameaca(T,cruz,W),!,moveC(W,T,o),! %% defesa
   ; vazia(5,T),moveC(5,T,'o'),!
   ; chute(9,W),member(W,[1,3,7,9]),vazia(W,T),moveC(W,T,'o'),!
   ; chute(9,W),member(W,[2,4,6,8]),vazia(W,T),moveC(W,T,'o'),!.
%%
ameaca(Tab,CB,W) :- emlinha3(Pos),ameaca(CB,Pos,Tab,W),!.
ameaca(cruz,[A,B,C],T,A) :- vazia(A,T),cruz(B,T),cruz(C,T).
ameaca(cruz,[A,B,C],T,B) :- vazia(B,T),cruz(A,T),cruz(C,T).
ameaca(cruz,[A,B,C],T,C) :- vazia(C,T),cruz(A,T),cruz(B,T).
ameaca(bola,[A,B,C],T,A) :- vazia(A,T),bola(B,T),bola(C,T).
ameaca(bola,[A,B,C],T,B) :- vazia(B,T),bola(A,T),bola(C,T).
ameaca(bola,[A,B,C],T,C) :- vazia(C,T),bola(A,T),bola(B,T).
%%-----------------------------------------------------------
%% desenha o tabuleiro
wrtLinha(X,Y,Z,T):-arg(X,T,V1), wVal(V1),write('|'),

                    arg(Y,T,V2), wVal(V2),write('|'),
                    arg(Z,T,V3), wVal(V3),nl.
wVal(X):- var(X)->write(' ');write(X).
desenha(T) :- nl, tab(7),wrtLinha(1,2,3,T), tab(7),write('------'),nl,
                  tab(7),wrtLinha(4,5,6,T), tab(7),write('------'),nl,
                  tab(7),wrtLinha(7,8,9,T).
```

```
%% ?- T=tab(_,_,o, _,x,_, _,_,_), desenha(T).
%%-----------------------------------------------------------------
%% esquema principal do jogo
jogo :- T = tab(A,B,C,  D,E,F,   G,H,I),
        exibeJogo(T, inicio),
        jogar(T, oponente).
jogar(T, Jogador):- gameOver(T,Result),!,msgFim(Result).
jogar(T, Jogador):- escolheMov(T, Jogador),!,
                    exibeJogo(T, Jogador),!,
                    proximo(Jogador, Oponente), !,
                    jogar(T, Oponente).
proximo(computador,oponente).
proximo(oponente,computador).
%%
exibeJogo(T,J):- write('jogou:'),write(J),desenha(T).
msgFim(X):-write('GAME OVER:'), write(X),nl,nl.
chute(N,S):-repeat, S is random(N).
%---------------------------------------
%% base para teste
t0(tab(o,o,o, *,*,*, *,*,*)).
t1(tab(o,x,o, x,o,x, o,x,o)).
t2(tab(o,x,o, x,*,x, *,*,*)).
%% ?- t0(T), vence(T,V).
%% ?- t1(T), gameOver(T,O).
%% ?- t2(T), gameOver(T,O).
```

7.2.5 Falando sobre testes

Às vezes é bom criar uma base de fatos para testar um programa. Por exemplo, abaixo temos três fatos, `t0`, `t1` e `t2`, com três valores para o tabuleiro. Junto com estes fatos podemos deixar em forma de comentários perguntas prontas para serem disparadas no interpretador. Esta é uma recomendação que fazemos para facilitar a manutenção dos programas. Quando alteramos um predicado basta selecionar com o mouse o texto da pergunta e jogá-lo no interpretador.

Seguem alguns exemplos que usam os fatos da base de testes dos predicados descritos acima.

```
%% base para teste
t0(tab(o,o,o, *,*,*, *,*,*)).
t1(tab(o,x,o, x,o,x, o,x,o)).
t2(tab(o,x,o, x,*,x, *,*,*)).
%% ?- t0(T), vence(T,V).
%% ?- t1(T), gameOver(T,O).
%% ?- t2(T), gameOver(T,O).
```

```
?- t0(T), vence(T,V).
   T = tab(o, o, o, *, *, *, *, *, *)
   V = o
   Yes
?- t1(T), gameOver(T,O).
```

```
   T = tab(o,x,o,x,o,x,o,x,o) ,
   O = venceu(bola) ;
?- t2(T), gameOver(T,O).
   No
```

Existem diversos motivos para se criar uma base de testes:

- primeiro, para documentar o que estamos pensando quando criamos um predicado; quando alguém está lendo o programa, ele vê os exemplos de uso;
- segundo, para ser usada quando estamos dando manutenção ao predicado (por exemplo, corrigindo e estendendo o predicado); devemos refazer os testes iniciais e atualizar as perguntas da base de testes;
- terceiro, para evitar a digitação repetida de perguntas longas; principalmente para programas não triviais, com listas, árvores e estruturas mais complexas. No desenvolvimento de um programa é normal repetir uma pergunta N vezes até o predicado desenvolvido agir como esperado;
- por fim, uma base de testes permite sistematizar o teste dos predicados; podemos pensar sobre os casos a serem testados e codificar os que realmente são significativos.

Em Prolog, programas ou componentes de programas extensos (mais de 50 linhas) e complexos necessitam ser comentados com perguntas que exemplificam e testam os predicados. Um programa mal documentado dificulta ou impossibilita um processo posterior de leitura e manutenção. Isto se deve em parte porque o Prolog é uma linguagem que permite uma codificação compacta.

Por outro lado, devemos ter o cuidado para não acrescentar muitos comentários não significativos, o que também pode prejudicar a leitura do programa.

Na programação, devemos preparar perguntas para todos os predicados não triviais ou predefinidos (como append, member, select). Devemos também preparar bases de fatos para sistematizar os testes.

Em termos de engenharia de software existem vários critérios para medir a complexidade de um programa. Um deles é o número de linhas do código-fonte sem comentários. O programa do jogo da velha tem 90 linhas de código: 25 linhas de comentários e 65 sem comentários.

Um programador Prolog experiente pode ler o programa do jogo-da-velha mesmo sem os comentários. Mas, para isso, terá um trabalho de reengenharia, no qual são examinados, um a um, os predicados não triviais dos programas. Por outro lado, um programador Prolog iniciante sentirá muita dificuldade na leitura de um programa do porte do jogo-da-velha sem comentários. Um terço de código, como comentários, é um número aceitável e significativo para um programa do porte do jogo-da-velha.

Outras medidas de complexidade são: (1) o número de predicados únicos e (2) o número de construções condicionais.

O número de construções condicionais é uma das medidas mais apuradas. Um predicado definido por cinco cláusulas codifica cinco condições diferentes. Mas os condicionais também podem estar codificados com o operador de disjunção (";"). Por exemplo, o predicado `wVal/1` embute dois condicionais e o predicado `escolheMov` embute cinco condicionais. Sobre o número de condicionais, no programa jogo-da-velha tem 36. Sobre o número de predicados únicos são 23; destes, 8 são definidos por mais de uma cláusula (com recursividade e/ou condicionais).

Um predicado no Prolog corresponde grosseiramente a um procedimento numa linguagem como C, ou Pascal. Porém, quando comparamos duas soluções similares, uma codificada em linguagem

imperativa (e.g., C, Pascal) e outra em Prolog, observamos que em Prolog o número de predicados é usualmente maior que o número de procedimentos da versão imperativa. Por exemplo, em C ou Pascal, um único procedimento desenha-tabuleiro provavelmente empacotaria os três predicados do Prolog (`desenha`, `wVal` e `wrtLinha`).

Em Prolog, um dos motivos que facilita a divisão de um procedimento em vários predicados é a reduzida necessidade de *açúcar sintático* na definição de um predicado. Compare um predicado em Prolog com a sua versão em Pascal, abaixo. Por açúcar sintático, entendemos as palavras-chave da linguagem que são usadas com o propósito de expressar uma construção da linguagem. Em Pascal, são as palavras *procedure, char, begin, end, then, else*. São 35 caracteres no Prolog contra 64 no Pascal. Claramente, o procedimento em Pascal é bem mais verboso que o predicado em Prolog.

```
wVal(X) :- X='*' -> write(' '); write(X).
%%
procedure wVal(X:char);
  begin
    if X='*' then write(' ') else write(X);
  end;
```

7.3 Projetos: Utilizando estratégias de jogo

A estratégia de jogo codificada nesta versão do jogo-da-velha ainda é pobre. Cabe ao leitor enriquecê-la. Por exemplo, sabemos que se um jogador parte jogando na posição central e o oponente não joga num canto, o oponente perde o jogo. A versão acima não codifica esta estratégia. Veja abaixo o que acontece:

```
computador bola
movimentos:     (1) (2) (3) (4) (5) (6) (7) (8)
                5o, 2x, 1o, 9x, 3o, 7x, 8o, 4x,

                +++ +x+ ox+ ox+ oxo oxo oxo oxo
                +o+ +o+ +o+ +o+ +o+ +o+ +o+ xo+  empate
                +++ +++ +++ ++x ++x x+x xox xox
```

Neste caso, pode-se usar uma "armadilha": o computador no quinto movimento jogando na posição sete cria duas linhas com ameaça da bola, e a cruz perde o jogo. Veja abaixo:

```
computador bola
movimentos: (1) (2) (3) (4) (5) (6) (7)
            5o, 2x, 1o, 9x, 7o, 3x, 4o,

            +++ +x+ ox+ ox+ ox+ oxx oxo
            +o+ +o+ +o+ +o+ +o+ +o+ oo+  venceu bola
            +++ +++ +++ ++x o+x o+x oox
```

Esta estratégia pode ser generalizada: se no segundo movimento o oponente não jogar num canto ele perde, não importa em qual das quatro posições ele joga.

Outra estratégia está relacionada com os cantos. Como ilustrado abaixo, a idéia é encontrar uma posição comum livre em duas linhas, cada linha tem uma marcação bola e duas livres. Se jogamos a marcação comum criamos uma armadilha.

```
computador bola
movimentos:    (1) (2) (3) (4) (5) (6) (7)
               1o, 5x, 9o, 3x, 7o, 4x, 8o,

               o++ o++ o++ o+x o+x o+x o+x
               +++ +x+ +x+ +x+ +x+ xx+ xx+  venceu bola
               +++ +++ ++o ++o o+o o+o ooo
```

Exercício 7.3 *Programe as armadilhas descritas acima para o jogo do computador ficar mais inteligente.*

Exercício 7.4 *Faça uma versão do jogo na qual se pode escolher quem começa. Quando o oponente inicia o jogo, certifique-se que o computador nunca perde.*

Exercício 7.5 *Faça uma versão do jogo em que o computador inicie jogando em uma posição aleatória.*

Exercício 7.6 *Faça uma versão do jogo que simule a inteligência de uma criança: jogando em posições aleatórias; às vezes, pensando e se defendendo ou, às vezes, caindo numa armadilha. Neste caso, o oponente pode criar armadilhas e vencer o jogo.*

CAPÍTULO 8

Classificação e ordenação

Este capítulo complementa o capítulo sobre processamento de listas. São apresentados diversos exemplos de algoritmos de ordenação fazendo uso das operações definidas para listas. Em geral, estes algoritmos não têm um valor prático, em termos de eficiência, para serem usados com o propósito de ordenar um grande conjunto de dados. Um deles, o método por intercalação, é normalmente programado como o predicado de uso geral que vem embutido em sistemas Prolog (chamado de `sort/2`).

O estudo destes algoritmos visa ao exercício de técnicas de programação em problemas práticos. São apresentados sete métodos de ordenação: ordenação por permutação, por inserção direta, por seleção, por troca, por intercalação (2 métodos) e por partição e troca.

Nestes métodos são revistos vários algoritmos sobre listas e apresentados alguns novos, tais como: permutação, testar se uma lista está ordenada, inserir um elemento numa lista ordenada mantendo a ordem, selecionar o maior elemento de uma lista, trocar dois elementos contíguos; particionar uma lista em duas listas de mesmo tamanho, intercalar duas listas já ordenadas em uma lista ordenada, etc.

No final do capítulo, dedicamos uma seção ao teste de performance de predicados, no qual são comparadas as performances de alguns algoritmos de ordenação, de acordo com o número de inferências e o tempo em segundos. Uma inferência corresponde a uma operação da máquina abstrata do Prolog (equivale grosseiramente a uma operação de unificação).

8.1 Ordenação por permutação

A ordenação por permutação é um método declarativo de ordenação. Ele usa a estratégia de solução de problemas chamada *geração e teste*. O objetivo deste método é verificar, por meio de testes, se cada permutação gerada está ordenada. Do conjunto de todas as permutações sabemos que pelo menos uma delas está ordenada, i.e., é a solução.

```
1  permOrd(L,S):-permutation(L,S),isOrdered(S).
2
3  isOrdered([]).
4  isOrdered([_]).
5  isOrdered([X,Y|XYs]):-X=<Y,isOrdered([Y|XYs]).
6
7  select(X,[X|Xs],Xs).
8  select(X,[Y|Xs],[Y|Zs]):-select(X,Xs,Zs).
9
10 permutation(Xs,[Z|Zs]):-select(Z,Xs,Ys),permutation(Ys,Zs).
11 permutation([],[]).
```

O predicado `permOrd/2` ordena a lista L em S. Os predicados de manipulação de listas foram definidos no capítulo sobre listas: `permutation/2` gera todas as permutações de uma lista, enquanto `isOrdered/1` testa se a lista está ordenada.

```
?- isOrdered([2,3,7,7]).
   Yes
?- isOrdered([3,2,7]).
   No
?- permutation([5,2,4],L).
   L= 5,2,4;
   L= 5,4,2;
   L= 2,5,4;
   ...
?- permOrd([5,2,4],L).
  L=[2,4,5]
```

8.2 Inserção direta

O método de classificação por inserção direta sistematicamente insere os elementos da lista de entrada, um a um, numa nova lista ordenada.

Há dois procedimentos para programá-lo: com acumulador e sem acumulador. Ambos fazem uso de um predicado `insOrd/3`, que insere um elemento em uma lista mantendo-a ordenada. O predicado `insOrd/3` é definido por três regras:

- `insOrd` de um elemento `X` numa lista vazia é retornar `[X]`;
- `insOrd` de um elemento `X` numa lista com cabeça `Y`, se `X=<Y` resulta em inserir `X` antes de `Y`;
- caso contrário, se `X>Y`, chama-se recursivamente `insOrd/2` para inserir `X` na posição correta na cauda da lista.

```
insOrd(X,[Y|L],[X,Y|L]) :- X =< Y,!.
insOrd(X,[Y|L],[Y |Io]) :- X >  Y,!, insOrd(X,L,Io).
insOrd(X,[],[X]).
```

```
?- insOrd(4, [2,3,5,7],L).
   L=[2,3,4,5,7]
```

Neste programa, a cláusula base `insOrd(X,[],[X])` é a última do predicado `insOrd` e só é executada uma vez, quando o processo termina. Se ela for a primeira regra do predicado, será testada a cada iteração, apesar de ser executada apenas quando a lista estiver vazia. Nesta posição, ela é testada e executada uma única vez, trazendo eficiência ao predicado.

Com acumulador

Dada uma lista de entrada L, toma-se o elemento da cabeça de L para ser inserido em uma lista ordenada, representada no acumulador AC. Quando este processo termina, o acumulador ACC é devolvido como a solução. O processo codificado no predicado `insDirAcc/2` é exemplificado abaixo.

L-entrada	ACC	S-saída
7 4 8 3 8	[]	
4 8 3 8	7	
8 3 8	4 7	
3 8	4 7 8	
8	2 4 7 8	
[]	3 4 7 8 8	3 4 7 8 8

Sem acumulador

Pega-se a cabeça da lista de entrada e chama-se o algoritmo para ordenar a cauda, então insere-se a cabeça na posição correta, na cauda que já está ordenada. Este método é codificado em `insDir/2`.

```
insDirAcc(L,S):- insDirAcc(L,[],S).
insDirAcc([C|Ls],ACC,   S):-insOrd(C,ACC,ACC1),insDirAcc(Ls,ACC1,S).
insDirAcc([],    ACC, ACC).
%
insDir([C|Ls],So) :- insDir(Ls,Si),insOrd(C,Si,So).
insDir([],[]).
```

```
?-insDir([7,4,8,3,8],S)
   S=[3,4,7,8,8]
?-insDirAcc([7,4,8,3,8],S)
   S=[3,4,7,8,8]
```

8.3 Ordenação por seleção

Na ordenação por seleção, o maior elemento de uma lista de entrada é selecionado e inserido como cabeça de uma lista onde a solução é acumulada. Este método é similar ao da inserção direta, porém, neste o processamento ocorre na seleção do elemento a ser removido, enquanto que naquele o processamento ocorre na inserção do elemento.

L-entrada	S-saída
7 4 8 3 8	[]
7 4 3 8	8
7 4 3	8 8
4 3	7 8 8
3	4 7 8 8
[]	3 4 7 8 8

```
selecaoOrd(L,[M|S]):-remMax(M,L,Lo), selectOrd(Lo,S).
selecaoOrd([],[]).
%
maxL([X],X).
maxL([X|Xs],M):-max(Xs,M1),(X>M1,M=X; X=<M1,M=M1).
%
remMax(M,L,Lo):-maxL(M,L),select(M,L,Lo).
```

Os predicados maxL/2 e select/3 foram apresentados no capítulo sobre listas.

8.4 Ordenação por troca (bublle sort)

O método de ordenação por troca examina sistematicamente dois elementos contíguos na lista de entrada e se o elemento da direita for menor que o elemento da esquerda, eles são trocados de posição. O processo termina quando não há mais trocas a serem feitas.

```
trocaOrd(L,S):-
              append(ORD,[A,B|Ls],L), B<A,!,   %% tira A,B
              append(ORD,[B,A|Ls],Li),         %% troca B,A
              trocaOrd(Li,S).
trocaOrd(L,L).
```

```
?- L=[1,2,6,4,7], append(ORD,[A,B|Ls],L),B<A.
   ORD=[1,2], A=6, B=4, Ls=[7]
?- trocaOrd([1,2,6,4,7],L).
   L=[1,2,4,6,7]
```

Aqui usamos o append/3 para extrair dois elementos contíguos de uma lista. Sistematicamente são destacados dois elementos da lista, A e B, e, quando B<A, eles são trocados de posição usando append/3., como:

```
?- append(_,[A,B|_],[2,7,4,5]).
A = 2, B = 7 ;
A = 7, B = 4 ;
A = 4, B = 5 ; No
```

8.5 Ordenação por intercalação

A estratégia é particionar uma lista em duas, ordenando isoladamente cada uma delas e intercalando os resultados. No particionamento, se a lista de entrada tem um número par de elementos, as duas listas de saída terão o mesmo número de elementos. Se for impar, uma das listas terá um elemento a mais. Particionar consiste em tomar dois elementos da cabeça da lista e colocar cada um em uma das partições.

```
particiona([],[],[]).
particiona([X],[X],[]).
particiona([X,Y|XYs], [X|Xs],[Y|Ys]):-particiona(XYs,Xs,Ys).
%%
merge([A,B|Ls],S):-
    particiona([A,B|Ls],La,Lb),
    merge(La,Las),merge(Lb,Lbs),
    intercalar(Las,Lbs,S).
merge([X],[X]).
merge([],[]).
%
intercalar([A|As],[B|Bs], [A|ABs]):-A=<B, intercalar(As,[B|Bs],ABs).
intercalar([A|As],[B|Bs], [B|ABs]):-A >B, intercalar([A|As],Bs,ABs).
intercalar([],Bs,Bs).
intercalar(As,[],As).
```

Intercalar duas listas, já ordenadas, consiste em escolher a lista que tem o menor elemento na cabeça, assumir este elemento como a cabeça do resultado e chamar recursivamente o processo para obter a cauda do resultado.

```
?- particiona( [6,2,1,4],L, M).
   L = [6,1], M = [2,4]
?- intercalar( [1,2], [4], L).
   L = [1,2,4]
?- merge([1,2,6,4,7],L).
   L = [1,2,4,6,7]
```

Exercício 8.1 *Examine o particiona/3 e responda. Quando temos um número ímpar de elementos na lista de entrada, qual das duas partições terá um elemento a mais? Justifique.*

Exercício 8.2 *Reescreva* `intercalar` *usando um comando* `if-then`*.*

Solução:

```
intercalar([A|As],[B|Bs],R):- (( A=<B -> R=[A|ABs],intercalar(As,[B|Bs],ABs);
                                 A >B -> R=[B|ABs],intercalar([A|As],Bs,ABs)   )).
```

8.6 Ordenação por intercalação (otimizada)

O Prolog possui um algoritmo de ordenação embutido, implementado como um algoritmo de ordenação por intercalação. Porém, no Prolog, o trabalho de particionar explicitamente uma lista

exige muito tempo e recursos de memória. Por isso, é possível simular o particionamento da lista a partir da informação do número de elementos.

O Prolog oferece um algoritmo para ordenação em que cada elemento de uma lista é associado a uma chave de ordenação. Assim, os valores a serem ordenados são organizados como nos testes apresentados abaixo. A chave é o primeiro elemento do par (uma chave pode ser qualquer constante do Prolog, incluindo termos compostos). O segundo elemento do par também pode ser qualquer termo. É possível, por exemplo, ordenar uma lista de registros de pessoas usando como chaves o nome delas.

```
?- sortKey([1-f,5-b,1-c,6-d,15-e,3-d],S).
  S = [1 - f,1 - c,3 - d,5 - b,6 - d,15 - e]
%
?- sortKey([f-1,b-5,c-1,d-6,e-15,d-3],S).
  S = [b - 5,c - 1,d - 6,d - 3,e - 15,f - 1]
```

Segue o código para o algoritmo de ordenação:

```
1  sortKey(List, Sorted) :-
2      length(List, Length),
3      sortKey(Length, List/ _, Result),
4      Sorted = Result.
5  sortKey(2, [X1, X2|L]/L, R) :- !,
6      X1 = K1-_,
7      X2 = K2-_,
8      (( K1 @=< K2 ->  R = [X1, X2]
9                   ;   R = [X2, X1]   )).
10 sortKey(1, [X|L]/L, [X]) :- !.
11 sortKey(0, L/L, []) :- !.
12
13
14 sortKey(N, L1/L3, R) :-
15     N1 is N // 2, N2 is N - N1,
16     sortKey(N1, L1/L2, R1),
17     sortKey(N2, L2/L3, R2),
18     mergeKey(R1, R2, R).
19 mergeKey([], R, R) :- !.
20 mergeKey(R, [], R) :- !.
21 mergeKey(R1, R2, [X|R]) :-
22     R1 = [X1|R1a], X1 = K1-_,
23     R2 = [X2|R2a], X2 = K2-_,
24  ((  K1 @> K2 ->  X = X2, mergeKey(R1, R2a, R)
25                  ;   X = X1, mergeKey(R1a, R2, R)    )).
```

Neste algoritmo, a maior dificuldade está em ter que percorrer toda a lista para calcular o seu comprimento. A partir do comprimento, faz-se a partição implícita pela divisão do valor N por dois, em `N1 is N//2` e `N2 is N-N1`. Assim, as duas partições terão comprimento N1 e N2. Enquanto as partições são maiores que dois, o algoritmo é recursivamente chamado. Quando uma partição é menor ou igual a dois, os valores são efetivamente removidos da lista de entrada. Cada subchamada `sortKey(N, Li/Lo, S)` pega N elementos da lista Li e deixa os demais elementos em Lo. No final da subchamada, os N elementos estão ordenados em S.

Exercício 8.3 *Refaça o algoritmo acima, chame-o de* `xsortKey/2`*, para trabalhar com uma lista simples de elementos sem os pares (chave - termo).*

8.7 Ordenação por partição e troca (Quick sort)

Outro algoritmo eficiente para ordenação de vetores de elementos é conhecido como quick sort. Dada uma lista de entrada, seleciona-se um elemento como pivô (neste exemplo é assumido o primeiro elemento da lista). Tomando-se o pivô, particiona-se a lista em duas, uma com todos os elementos menores que o pivô e outra com os maiores ou iguais. Este processo serve para ordenar ambas as listas, a dos menores e a dos maiores. Sabe-se que o elemento pivô deve ficar no meio dos dois resultados: menores, pivô, maiores.

`partic(Lista, Pivo, Lm(enores), LM(aiores))` seleciona todos os menores do elemento pivô para `Lm` e os demais para `LM`. Por exemplo, se a entrada for [7,4,8,3,8], o elemento pivô é o 7 e as listas são Lm=[3,4] e LM =[8,8].

```
1 partic([X|L],Pivo,[X|Lm],LM):- X< Pivo,!,partic(L,Pivo,Lm,LM).
2 partic([X|L],Pivo,Lm,[X|LM]):- X>=Pivo,!,partic(L,Pivo,Lm,LM).
3 partic([],_,[],[]).
```

```
?- partic([4,8,3,8],7,Lm,LM).
   Lm = [4, 3], LM = [8, 8]
```

Para obter o resultado, ordena-se Lm e LM em Sm=[3,4] e SM=[8,8], junta-se Sm, pivo e SM, nesta ordem, resultando em [3,4,7,8,8].

Para o quick sort são apresentadas três versões:

- a primeira, `qsort/2`, usa o append/3 para concatenar as listas;
- a segunda, `qs_acc`, usa um *acumulador*, de forma que o append não é necessário;
- a terceira, `qs_dl`, usa *diferença de listas* para implicitamente executar a concatenação de listas.

A técnica de uso de acumulares foi vista em vários programas e a da diferença de listas foi apresentada no capítulo sobre estruturas de dados.

Na versão `qsort` podemos ler o corpo principal como: particiona-se a lista de entrada `[X|Xs]` em duas, Lm (menores) e LM (maiores), do elemento pivô `X`. Chama-se o `qsort` de cada uma delas e, com o append, unem-se os resultados, com o cuidado de colocar o elemento pivô no meio do resultado `Lm+X+LM`.

Na versão com acumulador, `qs_acc`, o predicado auxiliar `qs_acc1` faz todo o processamento. Os valores são acumulados dos menores em direção aos maiores, de forma que a lista resultante fique em ordem crescente.

Inicialmente, a lista é particionada em duas. A seguir, chama-se o predicado `qs_acc1` para cada uma das partições. Depois, chamando o `qs_acc1`, ordenamos cada uma das listas, iniciando pela lista dos maiores. Um acumulador é usado para juntar os dois resultados parciais. O elemento-pivô é colocado no acumulador antes da lista ordenada dos maiores.

A versão com diferença de listas é bem próxima à versão com acumulador. Porém, a estratégia da solução é um pouco diferente, talvez mais fácil. O `qs_dl1` de uma lista é a diferença de listas `Si/So` se o `qs_dl1` da lista dos menores é `Si/[X|SM]` e o `qs_dl1` da lista dos maiores é `SM/So`.

Aqui a cauda do resultado dos menores é ligada ao resultado dos maiores. O elemento-pivô fica no meio.

As soluções com acumulador e diferença de listas são similares, mas cada uma usa a sua estratégia particular: com o acumulador guardam-se resultados intermediários, dos maiores para os menores, e sempre o resultado é incluído na cabeça do acumulador; com a diferença de listas, vemos o resultado como listas parciais, uma ligada na cauda da outra.

```
1  qsort([X|Xs],S):-
2      partic(Xs,X,Lm,LM),qsort(Lm,Sm),qsort(LM,SM),append(Sm,[X|SM],S).
3  qsort([],[]).
4  %%
5  qs_acc(L,S) :- qs_acc1(L, []/S).
6  qs_acc1([X|L],Acc/S) :-
7       partic(L,X,Lm,LM),qs_acc1(LM,Acc/SM),qs_acc1(Lm,[X|SM]/S).
8  qs_acc1([],S/S).
9  %%
10 qs_dl(L,S) :- qs_dl1(L, S/[]).
11 qs_dl1([X|L],Si/So) :-
12      partic(L,X,Lm,LM),qs_dl1(Lm,Si/[X|SM]),qs_dl1(LM,SM/So).
13 qs_dl1([],S/S).
```

Seguem exemplos de execução:

```
?- qsort([1,6,2,7],S).
   S = [1, 2, 6, 7]
?- qs_acc([1,2,6,1,7,1,0,11,-1],K).
   K = [-1,0,1,1,1,2,6,7,11]
?- qs_dl([1,2,6,1,7,1,0,11,-1],K).
   K = [-1,0,1,1,1,2,6,7,11]
```

8.8 Medindo a performance de programas

Neste capítulo vimos diversos algoritmos de ordenação. A seguir, vamos avaliar a performance destes algoritmos. No SWI-Prolog temos um predicado `time/1` que retorna o tempo gasto para execução de um predicado, bem como o número de inferências lógicas que foram executadas pela máquina abstrata do Prolog (todo sistema Prolog tem um predicado similar a este). Esta segunda medida é bem significativa para avaliar a performance de um programa Prolog.

Inicialmente definimos três listas como três fatos com, respectivamente, 25, 50 e 100 elementos. Podemos também usar o predicado `lista_rand/3` que gera uma lista de valores randômicos (foi estudado no capítulo sobre programação aritmética).

Precisamos também de um comando de repetição `repeatN/2` para executarmos o algoritmo várias vezes, pois para apenas 100 elementos o tempo será sempre 0.0 segundos.

```
1  lista_rand(   [],V,N):-N<1,!.
2  lista_rand([R|L],V,N):-N1 is N-1, R is random(V), lista_rand(L,V,N1).
3  %%
4  repeatN(0,_):-!.
5  repeatN(N,Call):-call(Call),N1 is N-1,repeatN(N1,Call).
```

```
%%
lista100r(L):-lista_rand(L,50,100).
%%
lista25([1,2,6,1,7,1,0,11,-11,2,6,1,-11,2,6,1,7,1,0,11,-11,2,6,1,7]).
lista50([1,2,6,1,7,1,0,11,-11,2,6,1,-11,2,6,1,7,1,0,11,-11,2,6,1,7,
         1,2,6,1,7,1,0,11,-11,2,6,1,-11,2,6,1,7,1,0,11,-11,2,6,1,7]).
lista100([11,-11,2,6,1,7,1,0,11,-11,2,6,1,7,1,0,11,-11,2,6,1,7,1,0,11,-1,
1,2,6,1,7,1,0,11,... ]).
```

Seguem alguns predicados de testes para a lista de 25 e 50 elementos. Os predicados `t01`, `t02` e `t03` apenas verificam se os algoritmos estão funcionando.

```
t01 :- lista25(L),write('qsort:'), qsort(L,S),write(S).
t02 :- lista25(L),write('qs_dl:'), qs_dl(L,S),write(S).
t03 :- lista25(L),write('qs_acc:'), qs_acc(L,S),write(S).
%
tm1 :- lista25(L),write('qsort:'), time(qsort(L,S)).
tm2 :- lista25(L),write('qs_dl:'), time(qs_dl(L,S)).
tm3 :- lista25(L),write('trocaOrd'), time(trocaOrd(L,S)).
%%
tm15 :- lista50(L),write('qsort:'), time(qsort(L,S)).
tm25 :- lista50(L),write('qs_dl:'), time(qs_dl(L,S)).
tm35 :- lista50(L),write('trocaOrd'), time(trocaOrd(L,S)).
```

Seguem os números para os testes com listas de 25 e 50 elementos (`tm?` e `tm?5`). O número de inferências para os algoritmos de quicksort cresce ligeiramente quando duplicamos o tamanho da lista. Porém, para o algoritmo de ordenação por troca `trocaOrd`, o valor é 8 vezes maior, saltou de 5,595 para 46,646 inferências.

```
?- t01.
    qsort:[-11, -11, -11, 0, 0, 1, 1, 1, 1, 1,...]
?- t02.
    qs_dl:[-11, -11, -11, 0, 0, 1, 1, 1, 1, 1,...]
?- t03.
    qs_acc:[-11, -11, -11, 0, 0, 1, 1, 1, 1, 1,...]
%%
?-tm1.
  % 447 inferences in 0.00 seconds (Infinite Lips)
   qsort:
?- tm2.
  % 401 inferences in 0.00 seconds (Infinite Lips)
  qs_dl:
?- tm3.
  % 5,595 inferences in 0.00 seconds (Infinite Lips)
   trocaOrd
%%
?-tm15.
    % 1,333 inferences in 0.00 seconds (Infinite Lips)
    qsort:
?- tm25.
```

```
   % 1,236 inferences in 0.00 seconds (Infinite Lips)
    qs_dl:
?- tm35.
    % 46,646 inferences in 0.11 seconds (424055 Lips)
    trocaOrd
```

Seguem os testes para alguns dos algoritmos de classificação, para listas de 100 elementos, primeiro com a lista codificada como fato e depois para a lista gerada com valores randômicos. O objetivo é saber qual o mais rápido e se a geração dos valores randômicos afeta o resultado.

```
1  tm10 :- lista100(L),write('qsort:'), time(repeatN(100,qsort(L,S))).
2  tm20 :- lista100(L),write('qs_dl:'), time(repeatN(100,qs_dl(L,S))).
3  tm30 :- lista100(L),write('qs_acc:'), time(repeatN(100,qs_acc(L,S))).
4  tm40 :- lista100(L),write('ins_dir:'), time(repeatN(100,insDir(L,S))).
5  tm50 :- lista100(L),write('sort:'), time(repeatN(100,sort(L,S))).
6  tm60 :- lista100(L),write('xsortKey'), time(repeatN(100,xsortKey(L,S))).
7  %%
8  tm10r :- lista100r(L),write('qsort:'), time(repeatN(100,qsort(L,S))).
9  tm20r :- lista100r(L),write('qs_dl:'), time(repeatN(100,qs_dl(L,S))).
10 tm30r :- lista100r(L),write('qs_acc:'), time(repeatN(100,qs_acc(L,S))).
11 tm40r :- lista100r(L),write('ins_dir:'), time(repeatN(100,insDir(L,S))).
12 tm50r :- lista100r(L),write('sort:'), time(repeatN(100,sort(L,S))).
13 tm60r :- lista100r(L),write('xsortKey'), time(repeatN(100,xsortKey(L,S))).
```

Abaixo, temos alguns resultados dos testes. O teste `tm10` mostra que o quick sort com uma lista randômica é mais eficiente. O algoritmo de inserção direta `tm40` se estende três vezes mais que o quicksort. O algoritmo embutido no Prolog `sort/2` é o mais rápido, pois, provavelmente está compilado em assembler. Os demais são executados na máquina abstrata do Prolog. O valor em Lips (Logic Inferences Per Second) deveria estar mais ou menos estável para todos os testes, pois define a performance de um sistema Prolog numa determinada máquina, o que aconteceu com exceção para o algoritmo `sort` embutido no Prolog.

```
?-tm10.
% 434,601 inferences in 0.66 seconds (658486 Lips)
?- tm10r.
% 284,801 inferences in 0.44 seconds (647275 Lips)
qsort:

?- tm40.
% 874,175 inferences in 1.49 seconds (586695 Lips)
?- tm40r.
% 960,231 inferences in 1.59 seconds (603919 Lips)
ins_dir:

?- tm50.
% 401 inferences in 0.06 seconds (6683 Lips)

?- tm50r.
% 401 inferences in 0.06 seconds (6683 Lips)
sort:
```

```
?- tm60.
% 272,701 inferences in 0.50 seconds (545402 Lips)
xsortKey
?- tm60r.
% 272,201 inferences in 0.49 seconds (555512 Lips)
xsortKey
```

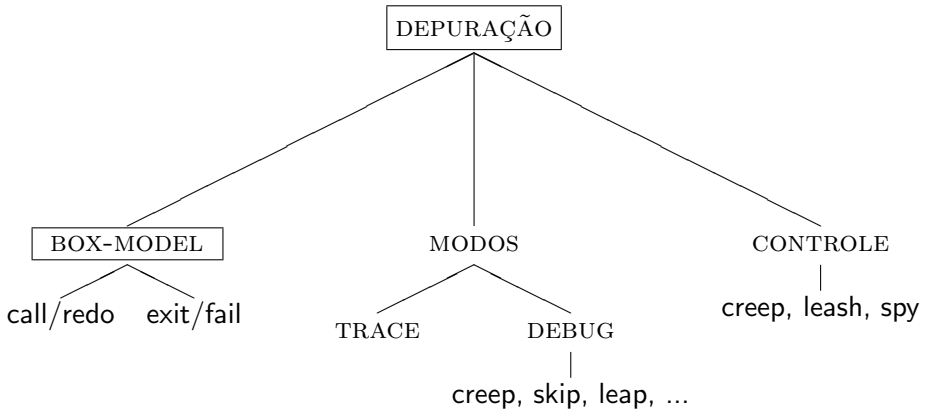
Este último predicado `tm60` testa o algoritmo `xsortKey`, uma versão simples do `sortKey`, solicitado como exercício.

Parte II

Ambientes de Programação e Sintaxe do Prolog ISO

CAPÍTULO 9

Depuração de Programas



O objetivo inicial deste capítulo é saber entrar e sair do modo trace, além de conhecer os diferentes cenários usados na depuração de programas em Prolog.

Somente um programador com experiência pode ter o domínio dos comandos de trace. Um programador iniciante deve começar apenas dando uma olhada no que está disponível. Conforme a necessidade, de forma incremental, estes recursos são aprendidos. Assim, este capítulo tem o propósito de servir como um guia de referência para consulta.

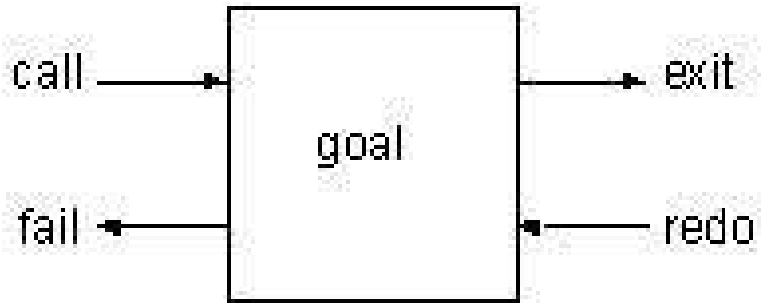


Figura 9.1: O modelo de caixa e portas de Byrd

O Prolog (Edimburgo) inclui um depurador de programas baseado num modelo de execução procedural, que permite ao usuário interagir examinando a execução de um programa em diferentes níveis de detalhamento.

Para depuração, todos os sistemas Prolog usam o *modelo de Byrd*, Figura 9.1. Este modelo é uma ferramenta conceitual que permite inspecionar o fluxo de controle do código de predicados de um programa durante a execução. Sob este modelo, com o uso combinado de alguns comandos, um programador experiente examina o código sendo executado num modo rápido e eficaz, localizando e diagnosticando problemas.

Alguns sistemas Prolog (por exemplo, LPA e Amzi) oferecem uma interface gráfica para o depurador. Basicamente é o modelo de caixas de Byrd floreado, por exemplo, o LPA permite também caminhar passo a passo sobre o código fonte.

9.1 O modelo de execução de Byrd

O modelo de Byrd (1980) [15] é baseado numa visão procedural do controle do fluxo de execução de um predicado. Este modelo é também chamado de *box model* ou modelo caixa e portas. Cada caixa é associada a um predicado. Uma caixa tem quatro portas, duas de entrada (call/redo) e duas de saída (exit/fail):

- **Call:** porta de entrada, chamando procedimento (predicado);
- **Exit:** porta de saída com sucesso;
- **Redo:** porta de entrada por retrocesso;
- **Fail:** porta de saída por falha.

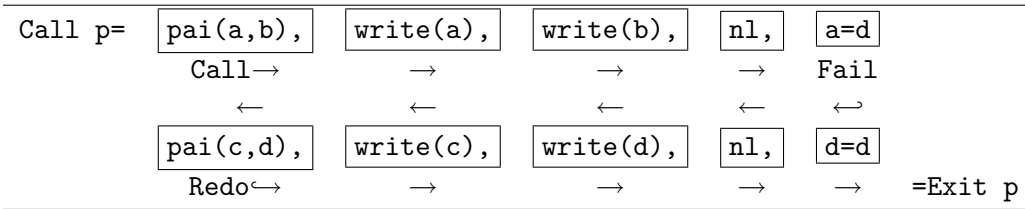
```
1 pai(a,b).
2 pai(c,d).
3 pai(e,f).
4 p:-pai(X,Y),write(X),write(Y),nl,Y=d.
```

Dado o programa acima, ilustramos uma seqüência de chamadas para a pergunta `?- p`. Para cada chamada de predicado (ou pergunta) é criada uma caixa. Caixas são encadeadas em uma estrutura aninhada, ligando-se porta com porta (por exemplo, `call` → `exit` → `call`).

Quando perguntamos por `p`, estamos chamando a execução do corpo de `p`. No retrocesso, os predicados não lógicos de entrada e saída (`write` e `nl`) não são reexecutados - estes predicados

não têm a porta `redo`. Somente os predicados com cláusulas alternativas têm a porta `redo`; isto é, podem ser reexecutados (como `pai/2`).

Note que quando o corpo do predicado `p` já é verdadeiro, não é mais chamada a terceira regra `pai/2`.



Aqui, uma seta para a direita indica que houve um `call` e um `exit`. Uma seta para a esquerda indica que está ocorrendo o retrocesso. As duas ocorrências de portas `fail` e `redo` estão explicitamente marcadas.

Usando o trace default do Prolog podemos visualizar de outra forma a seqüência de chamadas (call/redo) e saídas (exit/fail) ilustrada acima. O modo de trace mais básico é o *passo-a-passo* que é ativado pela pergunta `trace` e pelo comando interativo `creep` ou `enter`.

```
?-trace.
 Yes
[trace] 2 ?- p.
   Call: (6) p ? creep
   Call: (7) pai(_G374, _G375) ? creep
   Exit: (7) pai(a, b) ? creep
   Call: (7) write(a) ? creep
a
   Exit: (7) write(a) ? creep
   Call: (7) write(b) ? creep
b
   Exit: (7) write(b) ? creep
   Call: (7) nl ? creep

   Exit: (7) nl ? creep
   Call: (7) b=d ? creep
   Fail: (7) b=d ? creep
   Redo: (7) pai(_G374, _G375) ? creep
   Exit: (7) pai(c, d) ? creep
   Call: (7) write(c) ? creep
c
   Exit: (7) write(c) ? creep
   Call: (7) write(d) ? creep
d
   Exit: (7) write(d) ? creep
   Call: (7) nl ? creep

   Exit: (7) nl ? creep
   Call: (7) d=d ? creep
   Exit: (7) d=d ? creep
   Exit: (6) p ? creep
Yes
```

O depurador exibe em cada linha uma série de informações: o nome da porta; o número indicando o nível de profundidade de chamadas (geralmente não inicia por 0); e o predicado associado à caixa, com os valores para as variáveis. Se for uma porta `call/redo` serão as variáveis de entrada e se for uma porta `exit` serão as variáveis de saída. Na porta `fail` as variáveis não são unificadas com valores (permanecem livres).

9.2 Controle do nível de espionagem

Existem três comandos para controlar o detalhamento do trace:

- **creep** - entra no modo passo a passo, o mais detalhado;
- **leash** - seleciona as portas das caixas a serem espionadas;
- **spy** - seleciona os predicados a serem espionados.

Durante a depuração, podemos executar passo a passo (creep - rastejar no nível mais baixo, mais detalhado) ou podemos ignorar certas portas de caixas tomando-as como caixas pretas. O comando leash (ou trace/2) permite escolher os tipos de portas a serem visitadas. Portas ativadas para espionagem são chamadas portas de depuração (debug ports).

Quando rastejamos (creep) num predicado, todas as portas podem ser mostradas, mas o depurador pára somente nas portas selecionadas por leash (alguns sistemas só exibem as portas selecionadas).

Além do tipo de porta, podemos selecionar quais os predicados que queremos espiar (ou examinar). Assim, o programa roda até alcançar a execução de um predicado marcado como espionado (ou breakpoint).

	Escolha do nível de detalhamento:
leash	ativa/desativa portas para interação;
trace/2	ativa/desativa o trace em predicados;
spy/1	marca o predicado para ser espionado;
nospy/1	desmarca o predicado para ser espionado.

O comando `leash(Portas)` é chamado para uma lista de portas do conjunto: `call, redo, exit, fail, all`. O valor `all` refere-se a todas as portas; uma porta precedida por `(+)` é ativada e uma porta precedida por `(-)` é desativada.

Seguem alguns exemplos de escolha de portas,

```
?- leash(call).
?- leash([call, exit]).
?- leash([]).            % desativa todas
?- leash(all).           % ativa as 4 portas
?- leash([-call, +exit]) % desativa call ativa exit
```

Os predicados `spy/1` e `nospy/1` podem receber uma lista de parâmetros como argumentos. Por exemplo: `?- spy([p/1,q/3])`

O predicado `trace/2` equivale a um predicado spy e a um leash, portanto ele tem dois parâmetros, por exemplo, `?- trace(prog/1, [+fail, +exit])`. No geral, `?-trace(Pred, Portas)` é equivalente a `?-spy(Pred)` e `?- leash(Portas)`.

Existe também o `trace/0` que eqüivale a um spy em todos os predicados do programa e ao mesmo tempo a um leash em todas as portas. Isto é, por default, todas as portas e todos os predicados são selecionados.

9.3 Modos de depuração: trace ou debugging

Existem duas principais formas de depuração: (1) no modo **trace** o Prolog só mostra as informações das portas; (2) no modo **debug** o depurador pára a execução e permite a interação com o usuário, por exemplo, para executar o programa passo a passo.

	Predicados para entrar no modo depuração:
trace/notrace	ativa/desativa o trace em predicados **não** pára nas portas, mas mostra os valores nas portas;
debug/nodebug	ativa/desativa o trace, parando em cada porta dos predicados que estão sendo espionados;
debugging	informa os predicados já marcados para espionar;

Como exemplo, vamos examinar o append1/3 só em algumas portas. Primeiro só a porta call. Depois ambas as portas, call/exit.

```
1 append1([],Y,Y):-true.
2 append1([X|Xs],Ys, [X|Zs]):-append1(Xs,Ys,Zs).
```

```
?- trace(append1/3,+call).
%         append1/3: [call]
Yes
[debug] 13 ?- append1([a,b],[c,d],X).
 T Call: (7) append1([a, b], [c, d], _G513)
 T Call: (8) append1([b], [c, d], _G589)
 T Call: (9) append1([], [c, d], _G592)
X = [a, b, c, d]
%
?-trace(append1/3,[+call,+exit]).
%         append1/3: [call, exit]
Yes
[debug] 16 ?- append1([a,b],[c,d],X).
 T Call: (7) append1([a, b], [c, d], _G513)
 T Call: (8) append1([b], [c, d], _G589)
 T Call: (9) append1([], [c, d], _G592)
 T Exit: (9) append1([], [c, d], [c, d])
 T Exit: (8) append1([b], [c, d], [b, c, d])
 T Exit: (7) append1([a, b], [c, d], [a, b, c, d])
X = [a, b, c, d]
```

9.4 Interagindo com o trace

No modo debug o depurador pára em cada porta e interage com o usuário por meio de cinco comandos: creep, skip, leap, retry e unify. O creep força o rastejo passo a passo; o skip salta

sobre parte de predicados; o leap salta para a próxima porta; o retry e unify combinados permitem desfazer uma falha para se ver até aonde o programa vai.

	Comandos para interagir com o trace:
(c)creep	ou <enter> continue, mostre a próxima porta;
(s)skip	de uma porta call/redo vai direto para a próxima porta exit/fail – ignora sub-chamadas;
(l)leap	vai para próxima porta de um predicado sendo espionado;
(r)retry	de uma porta fail/exit executa novamente a última chamada;
(u)unify	numa porta call, força a execução com sucesso.

Abaixo, criamos um cenário de depuração no SWI-Prolog para o programa que segue. O leitor é convidado a criar este cenário em seu sistema Prolog, carregando este programa e executando os comandos como indicado. Diferente versões de Prolog podem apresentar pequenas variações do conteúdo deste cenário.

```
1 member1(X,[X|_]):-true.
2 member1(X,[_|Xs]):-member1(X,Xs).
3 pessoa(X):-member1(X,[jose,ana,joao,maria,pedro,antonia]).
4 namora(jose,ana).
5 casal(joao,maria).
6 casal(pedro,antonia).
7 casado(X):-casal(X,_);casal(_,X).
8 solteiro(X):- pessoa(X),\+ casado(X).
```

```
?- spy(solteiro/1).
% Spy point on solteiro/1
Yes
[debug] ?-solteiro(X).
   Call: (7) solteiro(_G437) ? skip
   Exit: (7) solteiro(jose) ? skip
X = jose
%%
[debug] ?-solteiro(X).
   Call: (7) solteiro(_G437) ? creep
   Call: (8) pessoa(_G437) ? skip
   Exit: (8) pessoa(jose) ? skip
   Call: (8) casado(jose) ? skip
   Fail: (8) casado(jose) ? skip
   Exit: (7) solteiro(jose) ? leap
X = jose
%%
Yes
[debug] ?- solteiro(X).
   Call: (7) solteiro(_G437) ? creep
   Call: (8) pessoa(_G437) ? leap
   Exit: (7) solteiro(jose) ? leap
X = jose
```

```
[debug] ?-solteiro(X).
   Call: (7) solteiro(_G437) ? h(elp)
Options:
+:          spy        -:          no spy
a:          abort      A:          alternatives
b:          break      c (ret, space): creep
[depth] d:  depth      e:          exit
f:          fail       [ndepth] g: goals (backtrace)
h (?):      help       i:          ignore
l:          leap       L:          listing

n:          no debug   p:          print
r:          retry      s:          skip
...
```

Os predicados predefinidos no sistema (tais como `member, write`) não são examinados pelo depurador.

Mudando-se o modo de depuração (debug/trace) não se alteram as portas selecionadas – para isso, é necessário usar o comando leash.

Pela combinação de três principais comandos (creep - rastejar, leash - ativar porta, e spy/trace - escolher predicados), o Prolog apresenta um ambiente de depuração dos mais flexíveis encontrados em linguagens de programação. Isto é particularmente válido para programadores experientes, que sabem combinar com arte estes três comandos. Por exemplo, pulando sobre o código que estamos seguros de que está correto, vamos rapidamente aos pontos que queremos examinar e, neste caso, podemos entrar no modo passo a passo (creep), ... Entrando e saindo em diferentes níveis, vamos identificando e diagnosticando os problemas existentes.

Exercício 9.1 *No seu sistema Prolog, repita cada um dos cenários de depuração apresentados neste capítulo.*

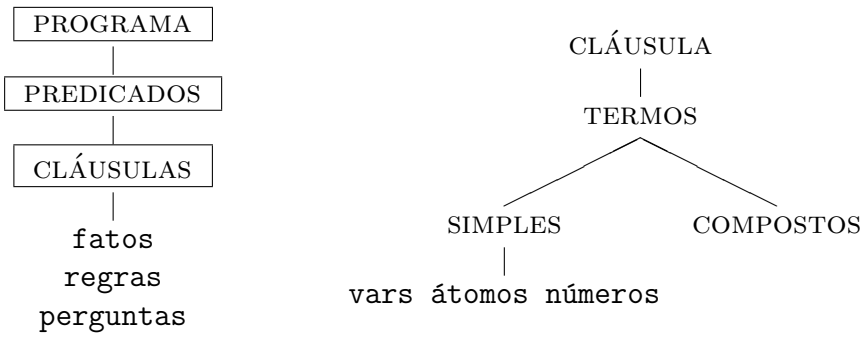
Exercício 9.2 *No seu Prolog, como é ativado o modo de trace (trace/debug) - com e sem interação?*

Exercício 9.3 *No seu Prolog, como são selecionadas as portas e os predicados?*

Exercício 9.4 *No seu Prolog, como interagir: creep, leap, skip, stop, help, etc. no modo debug?*

CAPÍTULO 10

O padrão ISO Prolog



Este capítulo apresenta uma sintaxe para a linguagem Prolog, de acordo com o padrão ISO (Deransart et ali. 1996) [10], que já adotado nos principais sistemas Prolog. São apresentadas diversas seções: a notação para cláusulas e operadores; expressões aritméticas; construindo e decompondo termos; predicados para manipular a base de fatos e predicados de entrada e saída.

Com exceção dos predicados de entrada e saída, o ISO Prolog é compatível com a sintaxe da linguagem do padrão Edimburgo. Descrevemos a sintaxe do Prolog ISO e, também, os predicados de entrada e saída do padrão Edimburgo que estão presentes em quase todos os sistemas Prolog e são largamente citados em livros de Prolog.

Não é essencial um estudo detalhado do conteúdo deste capítulo.

Porém, é necessário saber o que existe na linguagem Prolog. Aqui os predicados são listados mais para referência; o seu uso é mais bem compreendido em programas apresentados nos outros capítulos deste livro.

Um dos assuntos menos compreendidos no Prolog é o correto uso dos operadores. A sintaxe do Prolog é essencialmente baseada em operadores, por exemplo, uma cláusula `c:-p,q` é definida em termos de dois operadores `(:-)` e `(,)`, que internamente é representada como `':-'(c,','(p,q))`. Neste capítulo, dedicamos algumas páginas para clarificar o uso de operadores.

10.1 Cláusulas e Termos

Um **programa** Prolog é uma seqüência de cláusulas. **Cláusulas** podem ser de quatro tipos:

- fatos: `choveu.  pai(joao, ana).`
- regras: `p:-q.  tio(T,X):-pai(P,X),irmao(P,T). ab(X):-a(X);b(X).`
- assertivas: `:-op(300, xfy, ::).`
- perguntas: `?-p.  ?-pai(X,Y).`

Uma cláusula sempre termina por ponto e é formada por itens atômicos chamados de **termos**, que são os itens básicos para descrever dados e também programas. Assertivas são equivalentes a perguntas, porém são executadas durante a carga do arquivo enquanto que as perguntas são executadas na console.

10.1.1 Tipos de dados

Termos são usados para descrever predicados (procedures) mas também como tipos de dados. Existem quatro tipos de termos:

- variáveis: `A  B  Pessoa  Total_Pesssoa  X2`
- símbolos atômicos (átomos): `pessoa 'Joao' joao j11111`
- números: `1 -1000 10 12.3E4`
- termos compostos (estruturas): `1+1  +(1,1) p(X+1, 'PES'(joao,END))`

Uma **variável** inicia com uma letra maiúscula ou com o caractere de `underscore` (_) e consiste de letras, dígitos e underscores. Um único underscore denota uma variável anônima.

Um **símbolo atômico** é também chamado de átomo. Os átomos são escritos de três formas:

- Uma seqüência de letras, dígitos e/ou underscores, iniciando com letra minúscula:
  `aaaa  bbb_OU_aaa aBcD`
- Uma seqüência de um ou mais caracteres do conjunto de caracteres gráficos
  `# $ & * + - . / : < = > ? @ ^ ~ \`
  exemplos comuns são: `<=  \+  ?-  ::==`
- Uma seqüência de caracteres entre apóstrofos:
  `'Joao X' joao 'joao'`

Os símbolos atômicos formados por caracteres gráficos são também chamados de tokens gráficos (ou símbolos gráficos). Um símbolo gráfico é delimitado (ou termina) com um espaço, letra ou dígito. Na sentença `\++ ==>(aa)+ ::bb` temos seis ocorrências de símbolos atômicos, dos quais quatro são de símbolos gráficos: `\++` ␣ `==>` ␣ `(aa)` ␣ `+` ␣ `::`␣`bb`. O parêntese não faz parte de um símbolo gráfico, o que possibilita escrever operadores como functores, por exemplo, `+(1,*(2,3))` é o mesmo que "1+2*3"; os operadores são apresentados na próxima seção.

Para escrever caracteres especiais entre apóstrofos, é usada a barra invertida (backslash) \. Este símbolo no fim de uma linha dentro de apóstrofos denota que o string continua na próxima linha. Por exemplo,

```
... continua na prox\
    ima linha'.
```

equivale ao string `'... continua na próxima linha'`.

Outros usos da barra invertida são:

\a	caractere Ascii 7, beep code
\b	caractere de backspace
\f	caractere de formfeed
\n	caractere de nova linha
\r	caractere de enter sem nova linha
\t	tabulação horizontal
\v	tabulação vertical
\x23\	caractere com código hexadecimal 23
\23\	caractere com código octal 23
\\	barra invertida
\'	apóstrofo
\"	aspas
\‘	backquote

Apóstrofos podem acontecer dentro de átomos, escrevendo-se um apóstrofo duplo, por exemplo, `'joao d''agua' = 'joao d'agua'`.

Números

Números são os inteiros e os reais. Reais são descritos numa notação de ponto flutuante, por exemplo: `-2.34 2.34E5 -2.34E+5 2.34E-10`. Note que, `.234` e `2.` não são números válidos.

Inteiros são escritos de várias formas:

- como uma seqüência de dígitos decimais, por exemplo, `10 -345 1990`.
- como uma série de dígitos binários, octais ou hexadecimais precedidos respectivamente por `0b`, `0o` e `0x` tais como `0b101010`, `0o270`, `0x03BC`.
- como caracteres precedidos por `0'` tais como `0'a`, `0'z`; correspondem aos valores Ascii dos caracteres, respectivamente (97 e 122).

Testando os tipos de dados

O Prolog oferece um conjunto de predicados para testar o tipo de dados de um termo: `var(T)` e `nonvar(T)` testam se T é (não é) uma variável; `atomic(T)` é verdadeiro, se T é um termo atômico ou numérico; `atom(T)` é verdadeiro, se T é um termo atômico; `compound(T)` é verdadeiro, se T é uma estrutura não atômica; `number(T)` é verdadeiro, se T é um valor numérico; `integer(T)` é verdadeiro, se T é um valor inteiro; `float(T)` é verdadeiro, se T é um valor em ponto flutuante.

Segue um conjunto de testes para ilustrar o uso destes predicados, com variáveis e termos compostos:

```
?- var(1).  No
?- var(X).  X = _G174    Yes
?- nonvar(X).  No
```

```
?- nonvar(a).  Yes
?- nonvar(pai(X,Y)).  Yes  X = _G234, Y = _G235
?- nonvar(10). Yes
?- compound(pai). No
?- compound(pai(X,Y)). Yes    X = _G246, Y = _G247
```

Segue outro conjunto de teste para números e átomos:

```
?- number(10). Yes
?- number(a). No
?- float(1). No
?- float(1.1). Yes
?- atom(pai(joao,maria)). No
?- atom(pai). Yes
?- atom(2).  No
?- atomic(2). Yes
```

10.1.2 Termo composto

Um **termo composto** é definido por um functor, seguido por uma seqüência de argumentos entre parênteses, por exemplo `f(a,b,c)`. O **functor** é um símbolo atômico, incluindo os símbolos gráficos, por exemplo `+(1,2)`. Usualmente, podemos escrever um número arbitrário de espaços entre dois termos. Assim sendo, `f(␣␣␣a␣,␣b)` vale; porém f␣(a,b) resulta em erro de sintaxe. **Cuidado!** Não é permitido um espaço em branco entre o nome do functor e o abre parênteses.

Num termo composto, functores podem vir aninhados e fazendo referências aos vários tipos de termos atômicos, por exemplo

```
f(g(X,sss,_), h(i(12,3.4),Z), 'ABC abc', '123').
```

Um símbolo atômico também pode ser visto como um functor com zero argumentos.

O Tipo Lista

Uma lista é um termo composto. Por exemplo, `[a,b]` e `[a|B]` são representações sintáticas para os termos compostos `'.'(a,'.'(b,[]))` e `'.'(a,'.'(B,[]))` respectivamente. O símbolo gráfico de lista vazia é um símbolo atômico. Listas são estudadas em detalhes nos próximos capítulos.

Em Prolog tudo o que inicia com letra maiúscula é uma variável, por exemplo, `ABC`; tudo o que inicia com letra minúscula é constante (ou átomo), por exemplo, `abc`; cadeias de caracteres entre apóstrofos também são átomos, por exemplo, `'abc'` ou `'ABC'`. No entanto, `"ABC"`em Prolog não é um termo atômico como `'ABC'`. Uma seqüência de caracteres delimitada por aspas é uma representação sintática para uma lista de códigos ASCII, por exemplo, `"ABC"=[65,66,67]` e `"abc"=[97,98,99]`. Podemos perguntar ao interpretador:

```
?- X="ab ". X=[97,98,32] Yes"
?- X="abc", write_canonical(X).  %% ou  displayq(X)
'.'(97, '.'(98, '.'(99, [])))
X = [97, 98, 99] Yes
```

10.1.3 Dados vs programa = símbolos funcionais vs predicativos

No Prolog, termos são inicialmente classificados em duas categorias:

- símbolos predicativos: como procedimentos;
- símbolos funcionais: dados e estruturas de dados ou funções.

Por exemplo, no programa abaixo, `s/1` é um termo funcional e `p/1`, `is/2` são predicados. `+` é um símbolo funcional usado como uma função aritmética, que é avaliada pelo predicado `is/2`.

```
p(s(x)) :- p(x).  X is X+1.  is(X, X+1).
```

Os símbolos predicativos fazem o papel de procedimentos. Num programa sintaticamente válido, todo símbolo predicativo deve ser definido por pelo menos uma cláusula, da qual ela deve ser a cabeça.

Os símbolos funcionais assumem dois papéis:

- de funções executáveis, por exemplo `"+ * mod"` em `"X is Y mod 2*3"`
- de estruturas de dados, como em `"pessoa/2"` em `"retorne(pessoa(Nome,End)) :-procure(Nome,End),.. "`.

No Prolog, não existem regras que restringem o uso de um átomo para ser apenas símbolo funcional ou apenas um símbolo predicativo. Por exemplo, `pessoa` na cláusula abaixo
`retorne(pessoa(Nome,End)):-pessoa(Nome,_,_,End).`
assume os dois papeis: em `"pessoa/2"` é um símbolo funcional e em `"pessoa/4"` é um símbolo predicativo.

Em Prolog functores com diferentes números de argumentos são objetos diferentes, isto é, não unificam.

```
?- pessoa(N,E) = pessoa(N,Idade,E).
   No
```

Um programador experiente, no Prolog, tem facilidade em olhar e identificar quais símbolos são usados para definir estruturas de dados, funções ou predicados. Em alguns casos, o mesmo símbolo assume vários papéis em contextos diferentes.

10.1.4 Comentários

Um programa é descrito em um arquivo de texto que é formado por uma lista de linhas. As linhas são compostas de unidades atômicas chamadas de tokens (ou palavras). **Comentários** são usados como anotações no programa, as quais são ignoradas pelo analisador sintático.

Existem dois tipos de comentários:

- comentário de várias linhas: começa com `/*` e termina com `*/`;
- comentário de uma linha: começa com `%` e termina no fim da linha.

Um comentário de várias linhas é útil para incluir um trecho de um programa, como um conjunto de linhas, podendo conter vários comentários de linha.

10.2 Operadores

Os operadores tornam o código-fonte mais legível. Por exemplo, as expressões `1+2*3` e `+(1,*(2,3))` são equivalentes, sendo que a primeira está escrita com uso de operadores e a segunda com functores. Assim, os operadores permitem descrever expressões simbólicas de um modo mais econômico e natural. Seguem alguns exemplos:

```
1 ?- 1+2*3 = +(1,*(2,3)).
2     Yes
3 ?- 1+2*3=X, Y= +(1,*(2,3)), X=:=Y.
4     X= 7, Yes
5 ?- X=1+2*3, displayq(X).  %% write_canonical(X)
6     +(1, *(2, 3))
```

Toda expressão sem parênteses é parentizada antes da sua execução, para se saber como avaliar a expressão.

Prioridade	*Especificador*	*Operadores*
1200	xfx	`:-  -->`
1200	fx	`:-  ?-`
1100	xfy	`;`
1050	xfy	`->`
1000	xfy	`,`
900	fy	`\+`
700	xfx	`= \= == \== @< @=< @> @>= is =:= =\= < =< > >= =..`
500	yfx	`+  -  /\  \/`
400	yfx	`*  /  //  rem  mod  <<  >>`
200	xfx	`**`
200	xfy	`^`
200	fy	`\  -`
100	xfx	`@` (para módulos)
50	xfx	`:` (para módulos)

Figura 10.1: Operadores predefinidos conforme o Prolog ISO

Todo sistema Prolog possui um conjunto predefinido de operadores. Visando à padronização destes operadores, a norma ISO definiu, para o Prolog, o conjunto mostrado na Figura 10.1. A informação desta tabela, que define a precedência e associatividade dos operadores, permite o correto uso dos parênteses em expressões aritméticas ou simbólicas, como veremos nesta seção.

Um símbolo funcional ou predicativo com um ou dois argumentos pode ser declarado como operador. Por exemplo, `-1 = -(1)` é um functor com um argumento; e, `1 mod 2 = mod(1,2)` é um functor com dois argumentos. Existem três classes de operadores:

- pré-fixo: `-(1) = -1,  (-(- 1)) = 1, not not true = true`
- in-fixo: `1+3*5-(4*4) = 0`
- pós-fixo: `3!=6, (2!)! = 2! =2,  (3! !)=(3!)! = 720`

Os operadores são definidos a partir de regras de precedência e associatividade que possibilitam ao analisador sintático incluir numa expressão os pares de parênteses que forem necessários. A **precedência** é usada quando numa expressão temos operadores diferentes. Por exemplo, em `1+3*5-(4*4)` a multiplicação deve ser executada antes da soma e da subtração. Assim, precedência significa "deve ser executado antes". Por outro lado, a **associatividade** acontece numa expressão em que ocorrem várias vezes o mesmo operador[1], por exemplo, em `4/2/2`. Se for `(4/2)/2` resulta em 1; porém, se for `4/(2/2)`, resulta em 4. No primeiro caso é dada a prioridade de execução para o operador mais à direita (/ é associativo à direita) e no segundo caso para o operador mais à esquerda (/ é associativo à direita). No Prolog ISO, o operador / é associativo à esquerda.

Os operadores são um tipo de açúcar sintático, para efeito de leitura e escrita de termos. Toda expressão definida com operadores tem a sua representação interna em forma de termos definidos por functores. Por exemplo,

```
(4/2)/2   =   /( /(4,2), 2)  =  1
 4/(2/2)  =   /(4, /(2, 2))  =  4
```

Numa expressão, o uso explícito de parênteses inibe o uso das regras de precedência e associatividade. Por exemplo, a expressão `(1+3)*(5-4)*4=16` é executada respeitando-se o uso explícito dos parênteses.

Em Prolog, um operador é definido por dois parâmetros: um **especificador** que informa a classe e a associatividade; e um valor de **prioridade** para a precedência: de 0 até 1200. Valores maiores indicam precedência menor. Segue uma relação entre especificadores e associatividade.

Especificador	Classe	Associatividade
`yfx`	in-fixo	à esquerda
`xfy`	in-fixo	à direita
`xfx`	in-fixo	não associativo
`xf`	pós-fixo	não associativo
`yf`	pós-fixo	à direita
`fx`	pré-fixo	não associativo
`fy`	pré-fixo	à esquerda

Um mesmo átomo pode ser um operador em mais de uma classe. Por exemplo, o sinal de menos(-) é pré-fixo, porém, como operador de subtração ele também é in-fixo (`-1-3 = -4`).

CUIDADO: A norma ISO diz que, por motivos de eficiência na análise sintática (reduzir o *look-ahead*), um operador não pode ao mesmo tempo ser in-fixo e pós-fixo.

No Prolog ISO, os operadores são agrupados em classes com mesma precedência e associatividade. Por exemplo, todos os símbolos de igualdade e desigualdade são da classe `xfx` (não associativo, in-fixo) com prioridade 700. Os operadores `{+,-, /\, \/}` estão também numa mesma classe, `yfx` (associativa à esquerda) com prioridade 500. No Prolog ISO, existem também alguns operadores pré-fixos (`fy, fx`), por exemplo, `{?-, -}`; porém, não existem operadores pós-fixos predefinidos.

Numa expressão em Prolog, todos os caracteres visíveis possuem algum significado, com exceção dos parênteses que, em muitos casos, são desprezados pelo reconhecedor sintático. Por exemplo, estas duas expressões `1+2 = (( (1)+(2) ))` são iguais em termos de significado, mas a segunda está escrita com vários parênteses opcionais:

[1] Ou operadores diferentes mas com a mesma precedência.

```
?- X=(((1))+(2)).
   X = 1+2 yes
```

10.2.1 Operadores vs Árvore sintática

Um sistema Prolog constrói uma árvore sintática para definir o uso de parênteses numa expressão. Nas folhas estão os valores de menor precedência. A raiz é o valor de maior precedência. Tudo o que não é operador é considerado com prioridade 0: variáveis, números, functores, etc.

```
2 + 3 * 4 =:= X       (expressão)
2 + *(3,4) =:= X      (menor precedência *, 400)
+(2,*(3,4)) =:= X     (depois +, 500)
=:=(+(2,*(3,4)),X)    (depois =:=, 700)
```

As regras para construção da árvore sintática são baseadas na definição dos argumentos do operadores (o especificador e a prioridade). Cada nó da árvore terá dois filhos, se for da classe in-fixo, ou um filho, se for pré/pós-fixo. Uma árvore sintática, decorada com o especificador e a prioridade de cada operador, é válida se satisfaz as duas restrições que seguem:

- nós abaixo do operando `x` **devem ter prioridade menor;**
- nós abaixo do operando `y` **devem ter prioridade menor ou igual.**

Assim, uma árvore válida para a expressão `2+3*4 =:= Z` é dada abaixo (os valores dos parâmetros dos operadores são definidos na Figura 10.1).

```
        xfx =:=700
        /  \
+500 yfx    Z
     /  \
    2   yfx *400
       /   \
      3     4
```

Nesta árvore, se tentarmos colocar a multiplicação como raiz da árvore, as regras não serão satisfeitas, pois o nó com valor 400 está acima do nó com valor 500:

```
    *400 yfx
        /   \
 +500 yfx   4
     /   \
    2     3
```

Podemos também analisar o que acontece com a árvore da expressão 2+3+4. Seguem as duas possíveis árvores, da esquerda para `(2+3)+4` e da direita para `2+(3+4)`. Aqui, somente a árvore da esquerda é válida (500=500 no lado `y`), a da direita não vale porque 500 não é menor que 500, no lado `x`.

```
    +500 yfx                    yfx +500
       /   \                    /   \
  +500 yfx    4              2    yfx +500
     /   \                        /   \
    2     3                      3     4
```

Exercício 10.1 *Dadas as definições dos operadores do Prolog, conforme a Figura 10.1, faça a árvore das expressões que seguem:*

a. `1-2-3*4/4**6`

b. `x+y mod x / w * 3`

c. `p:-q,r,s; x,y,z; a,b`

Exercício 10.2 *Dadas as definições dos operadores abaixo, faça a árvore das expressões que seguem:*

```
:-op(300,xfy, (+)). :-op(400, yfx, (-)). :-op(500,xfy, (*)).
```

a. `1-2-3*4/4**6`

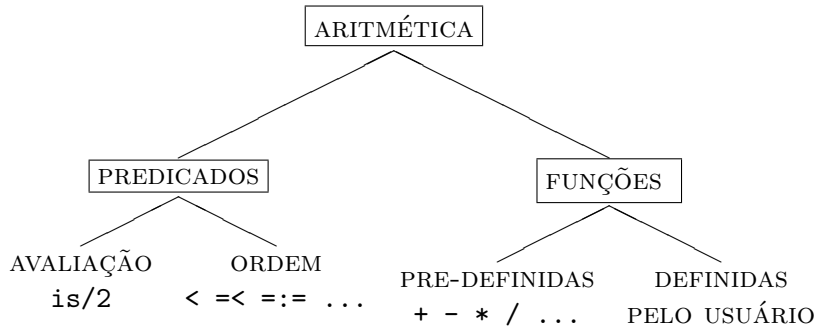
b. `a*b+c*d-e+f`

c. `a+b+c`

d. `a-b-c`

10.3 Expressões aritméticas

Um operador em Prolog pode denotar um predicado ou um símbolo funcional (função). Os predicados, quando aparecem no corpo de uma cláusula, disparam a avaliação das expressões envolvidas. Porém, um símbolo funcional só é avaliado se estiver no contexto de um predicado. Nesta seção clarificamos este tema.



O Prolog é uma linguagem desenvolvida especialmente para computação simbólica. A unificação, que é seu motor de computação, não avalia expressões aritméticas.

```
?- X = 1+3.
   X = 1+3 Yes
```

Por exemplo, na pergunta acima, o interpretador responde sem avaliar a expressão. Para o Prolog a expressão 1+3 corresponde a um functor + com dois argumentos

```
   ?- X = 1+3, write_canonical(X).
      +(1, 3)  Yes
   ?- X = 1+3, display(X).
      +(1, 3)  Yes
```

Para a manipulação simbólica, por exemplo, simplificar uma expressão aritmética, é importante que o Prolog permita representar expressões sem serem avaliadas. Por outro lado, quando precisamos um resultado de uma expressão, usamos o operador `is/2`, que força a avaliação de uma expressão. Este predicado é similar ao comando de atribuição, em linguagens procedurais (:= no Pascal, = no C++).

```
  ?- Y is 1+5.
      Y=6  Yes
  ?- X = 1+3*65/2, Z is X.
      X = 1+3*65/2, Z = 98.5 Yes
```

O predicado `is/2` exige que todas as variáveis do lado direito sejam fechadas (casadas com um valor). Por exemplo, à pergunta `?-X is A+3`, em que A é uma variável livre (não fechada), o interpretador responde com uma mensagem de erro.

```
 ?- X is A+3.
   ERROR: Arguments are not sufficiently instantiated
```

No lado esquerdo da atribuição, podemos ter também constantes. Neste caso, o lado direito é avaliado e, se for igual ao lado esquerdo, o predicado é verdadeiro.

```
?- 3 is 3.
        Yes
?- X=3, X is X.
    X=3 Yes
?- 3 is 2+1.   Yes
?- A=2, 3 is A+1.
   A=2    Yes
?- A=2, A is A/2+A/2.
   A=2    Yes
```

Por fim, um erro comum para principiantes é tentar incrementar uma variável fechada (já casada com um valor): `"X is X+1"`. Em lógica nunca é verdade. Deve-se atribuir o valor a uma variável nova.

```
?- X=4, X is X+1.
   No
?- X=4, X1 is X+1.
   X=4, X1=5  Yes
```

`(V is E)`	(V recebe o valor da expressão E)
	Sejam X e Y expressões aritméticas
`(X>Y) (X<Y)`	(X maior que Y) e (X menor que Y)
`(X>=Y) (X=<Y)`	(X maior ou igual a Y) e (X menor ou igual que Y)
`(X=:=Y)`	(X igual a Y)
`(X=\=Y)`	(X diferente de Y)
	Sejam I e J inteiros; Seja X real; e Seja N um inteiro ou real
`(N+N)  (N-N)`	Adição e subtração
`(N*N)  (N/N)`	Multiplicação e divisão
`(I//I)  (I rem I) (I mod I)`	Inteiros: Divisão, resto e módulo
`N ** N`	Exponenciação (resultado é real)
`-N`	Reverso do sinal
`abs(N)`	Valor absoluto
`ceiling(N)`	Menor inteiro não menor que `N`
`cos(N) sin(N) atan(N)`	Coseno, Seno, Arco tangente (em radianos)
`log(N) exp(N)`	Logaritmo $\log_e N$, Antilogaritmo natural, e^N
`sqrt(N)`	Raiz quadrada
`sign(N)`	Sinal (-1, 0, ou 1 para neg, zero, ou pos)
`float(N)`	Converte para real
`float_fractional_part(X)`	Parte fracional de `X` (negativo se `X` é negativo)
`float_integer_part(X)`	Parte inteira de `X` com sinal
`floor(X)`	Maior inteiro não maior que `X`
`truncate(X)`	Parte inteira de `X`
`round(X)`	Inteiro mais próximo a `X`
`(I>>J) (I<<J)`	Bit–shift `I` para a direita/esquerda `J` bits
`(I/\J) (I\/J)`	Bitwise: funções **and** e **or**
`\I`	Bitwise: complemento (inverte todos os bits de `I`)

Figura 10.2: Funções aritméticas do Prolog ISO

10.3.1 Functores aritméticos embutidos no Prolog

O Prolog possui um conjunto de predicados e funções aritméticas predefinidas na linguagem. Como predicados aritméticos temos o operador de atribuição `is/2` e todos os operadores de comparação (Figura 10.2). Os demais operadores aritmético são funções.

Expressões formadas por operadores aritméticos são avaliadas somente quando aparecem no lado direito do predicado `is/2` ou em ambos os lados de um operador de comparação. Em outros casos, as expressões aritméticas são manipuladas pela unificação como se fossem uma estrutura de dados qualquer, não sendo avaliadas aritmeticamente.

Seguem alguns exemplos, comparando os operadores aritméticos com o operador de unificação – somente os operadores avaliam as expressões aritméticas.

```
?- X=4, 1+3+2 < X+3.  Yes  X = 4
?- 3+1<4.  No
?- 3+5 =:= 5+3.  Yes
?- 3+5 = 5+3.   No
```

Além dos predicados aritméticos, o Prolog possui um conjunto de funções aritméticas, para se fazer cálculos matemáticos. Ambos, predicados e funções, podem ser declarados como operadores: por exemplo, `is/2` e `+/2` , em que o primeiro é um predicado e o segundo é uma função.

Exercício 10.3 *Qual a diferença entre um termo e um predicado? Dê exemplos?*

Exercício 10.4 *Qual a diferença entre um predicado aritmético, uma função e um operador? Dê exemplos?*

Exercício 10.5 *Usando o predicado* `write_canonical`*, examine a estrutura dos termos que seguem: a)* `1-2-3*4/4**6`*, b)* `x+y mod x/w*3`*, c)* `p:-q,r,s; x,y,z; a,b`.

Solução:

```
?- write_canonical(1-2-3*4/4**6 ).
   -(-(1, 2), /(*(3, 4), **(4, 6)))
?- write_canonical(x+y mod x / w * 3 ).
   +(x, *(/(mod(y, x), w), 3))
?- write_canonical((p:-q,r,s; x,y,z; a,b) ).
    :-(p, ;(','(q, ','(r, s)), ;(','(x, ','(y, z)), ','(a, b))))
```

Exercício 10.6 *Dada a definição dos operadores abaixo, usando o predicado* `write_canonical` *examine a estrutura dos termos que seguem:*

```
 :-op(300,xfy, (+)). :-op(400, yfx, (-)). :-op(500,xfy, (*)).
```
a) `1-2-3*4/4**6` b) `a*b+c*d-e+f` c) `a+b+c` d) `a-b-c` .

10.4 Construindo e decompondo termos

Para trabalharmos com metaprogramação é necessário saber construir e decompor termos. Termos são as estruturas de dados básicas do Prolog. Segue a lista dos predicados usados para manipular termos no Prolog:

- `functor(Termo, F, A)` — retorna a Aridade (A) e o Functor (F) do Termo. Com este predicado perguntamos qual o functor e quantos argumentos o termo possui; mas também podemos criar um novo functor com N argumentos.

```
functor(pai(joao,maria),F,A).
   F = pai  A = 2 Yes
?- functor(maria, F, A).
   F = maria A = 0  Yes
?- functor(F, pai, 4).
   F = pai(_G468, _G469, _G470, _G471)  Yes
```

- `arg(N, Termo, Arg)` — retorna o N-ésimo argumento (Arg) do Termo. O predicado `arg` é usado para extrair argumentos de um termo, portanto, o Termo e o N (número do argumento) devem estar instanciados. Seguem dois exemplos, o segundo mostra um erro, pois o termo não está instanciado.

```
?-arg(2,pai(joao,maria),A).
   A = maria
?- arg(2,X,maria).
   ERROR: arg/3: Arguments are not sufficiently instantiated
```

- `Termo =.. Lists` — transforma a Lista no termo e vice-versa.

```
?- pai(joao,maria) =.. L .
   L = [pai, joao, maria]
?- T =.. [x,1, 34, paulo].
   T = x(1, 34, paulo)
```

- `copy_term(T1,T2)` — cria uma cópia do termo T1 em T2, em que todas as variáveis livres são novas.

```
?- copy_term(pai(J,K),T1).
   J = _G273   K = _G274
   T1 = pai(_G362, _G363)
```

Aqui estes predicados são listados para referência. No decorrer do livro alguns deles são usados num contexto de aplicações práticas.

10.5 Base de fatos e regras (cláusulas e predicados)

O Prolog trabalha a partir de um programa, que é uma lista de cláusulas (regras e fatos), que define um conjunto de predicados. Quando carregamos um programa com `consult/1` as suas cláusulas são armazenadas na *base de fatos e regras*, que é manipulada pelos predicados apresentados a seguir:

- `:-dynamic(F/A)` — Diz que um predicado pode ser modificado durante a execução do programa; é usada junto com `assert` e `retract`.
- `current_predicate(F/A)` `clause(Cab, Cau)` — O primeiro retorna todos os predicados não predefinidos no sistema; o segundo retorna por retrocesso todas as cláusulas dos predicados declarados como dinâmicos.
- `asserta(C)` `assert(C)` `assertz(C)` — Armazena de modo permanente a cláusula passada como parâmetro. O primeiro e o segundo armazenam no início da lista de cláusulas associadas ao predicado; o terceiro no fim.
- `retract(C)` `abolish(F/A)` — O primeiro remove uma cláusula da base de fatos; o segundo remove todas as cláusulas do predicado definido pelo functor e pela aridade; a declaração de dinâmico é removida juntamente.

Segue um exemplo que faz uso de alguns dos predicados descritos (este fragmento de programa faz parte de um programa maior apresentado no capítulo sobre compiladores).

```
1 :-dynamic(tab/3).
2 initTab :- abolish(tab/3),dynamic(tab/3).
3 mk(C,X,V) :-tab(C,X,V),!;assert(tab(C,X,V)).
4 wTab :-tab(CV,ID,N),nl, wrL(['; ',tab(CV,ID,N)]), fail.
5 wTab .
6 %% ?-  mk(var,a123,123).
7 %% ?-  mk(const, zero, 0).
```

Exercício 10.7 *Teste os predicados* `initTab`, `mk` *e* `wTab`. *Explique o que faz cada um deles.*

10.5.1 Encontrando múltiplas soluções

Existem três predicados para manipular múltiplas soluções de perguntas: `findall`, `bagof` e `setof`. Segue um exemplo.

```
?- listing(pai).
   pai(pedro, maria).
   pai(pedro, ana).
?- findall(X,pai(X,Y),L).
   X = _G267   Y = _G268
   L = [pedro, pedro] ;
   No
?- bagof(X,pai(X,Y),L).
   X = _G255, Y = maria, L = [pedro] ;
   X = _G255, Y = ana,   L = [pedro] ;
   No
?- bagof(X,Y^pai(X,Y),L).
   X = _G255, Y = _G268, L = [pedro, pedro] ;
   No
```

O `findall` encontra todas as soluções para uma pergunta. A notação `Y^pai(X,Y)` informa para o sistema manter a variável `Y` livre. Caso contrário o sistema unifica esta variável com o primeiro valor encontrado.

O `bagof` é similar ao `findall`. O `findall` encontra todas as soluções possíveis mudando os valores para as variáveis livres. Já o `bagof` assume um valor fixo para as variáveis livres, trazendo para ele todas as soluções. Acumulativamente, por retrocesso, o `bagof` traz os mesmos valores que o `findall`. Se no `bagof` todas as variáveis são anotadas como livres, ele retorna os mesmos valores do `findall`, ver a última pergunta do exemplo anterior.

Por fim, temos o `setof` que é similar ao `bagof`, porém não retorna valores duplicados.

```
?- assert(p(x)), assert(p(x)), assert(p(y)).
   Yes
?- listing(p).
   p(x). p(x). p(y).
   Yes
?- setof(X,p(X),L).
   X = _G228  L = [x, y] ;
?- bagof(X,p(X),L).
   X = _G228  L = [x, x, y]
```

10.6 Entrada e saída

Iniciamos com a apresentação dos predicados de entrada e saída para o Prolog Edimburgo, que são largamente aceitos e usados na literatura e disponíveis em todos os sistemas. Em seguida, apresentamos as extensões destes predicados para o padrão ISO Prolog. Recomendamos ao leitor, dentro do possível, o uso dos predicados ISO.

10.6.1 Entrada e saída em arquivos (padrão Edimburgo)

O Prolog padrão Edimburgo define um pequeno conjunto de primitivas de entrada e saída, que servem para a leitura e escrita de termos e caracteres.

Lendo e escrevendo caracteres

Seguem as primitivas para manipular caracteres:

- `get0(X)` `get(X)` — O `get0(X)` lê da fita de entrada o valor Ascii do próximo caractere; O `get(X)` só retorna os caracteres visíveis (que podem ser impressos); pulando os caracteres com valor Ascii menor que 32 (espaço em branco).
- `put(X)` — inverso ao `get/1`, escreve na fita de saída o caractere correspondente ao valor Ascii X.

Segue um exemplo em que se escreve a palavra "Alo"a partir dos valores Ascii das três letras:

```
?- X="Alo".
   X = [65, 108, 111]
?- put(65),put(108),put(111).
   Alo
```

Enquanto o `get/1` lê somente os valor Ascii dos caracteres visíveis, o `get0/1` lê também o valor dos códigos Ascii para os caracteres de controle, como o tab (8) e espaço (32). No exemplo abaixo, o get0/1 lê o caractere de espaço (32) que é pulado pelo `get/1`.

```
?- get(A),get(B).
|    xy<enter>
   A = 120, B = 121  Yes
?- get(A),get(B).
|    x        y<enter>
   A = 120 B = 121 Yes
?- get0(A),get0(B).
|    x   y<enter>
   A = 120 B = 32 Yes
```

Lendo e escrevendo termos

Nos capítulos anteriores já vimos os predicados `nl`, `tab(N)`, `write(X)` e `read(X)`. Além destes existem dois que são especiais:

- `display(X)` — similar ao `write/1`, mas ignora a declaração dos operadores na escrita dos termos. Assim, todo termo é escrito na forma de functor.
- `writeq(X)` — escreve um termo na saída usando parênteses operadores e outros símbolos gráficos, de modo que o termo possa ser novamente lido pelo `read` do Prolog. Por exemplo, quando escrevemos com `write` um valor entre aspas, elas são desprezadas.

Segue uma lista de exemplos ilustrativos para estes comandos básicos de entrada e saída.

```
?- write(aa),tab(5),write(bb),nl, write(a2),tab(5),write(b2),nl.
   aa     bb
   a2     b2
?- write(a+b+c).
   a+b+c
?- display(a+b+c).
```

```
   +(+(a, b), c)
?- write('Bela Vista').
   Bela Vista
?- writeq('Bela Vista').
   'Bela Vista'
?- write(pai('Ana Maria', 'João')).
   pai(Ana Maria, João)
?- writeq(pai('Ana Maria', 'João')).
   pai('Ana Maria', 'João')
```

Os predicados `tab/1` e `nl` podem ser definidos a partir do predicado `put`, assim:

```
1 nl    :- put(13),put(10). %% enter  nova linha
2 tab(N):- N<1,!.
3 tab(N):- put(32), M is N-1, tab(M).
```

O Prolog padrão Edimburgo usa um modelo de entrada e saída que assume duas fitas correntemente abertas e selecionadas para a leitura e escrita, respectivamente. Por default, a fita de entrada está associada ao teclado e a de saída ao vídeo. A partir disso, o Prolog oferece primitivas para alternar os arquivos associados à fita de entrada e à fita de saída. Assim, podemos ter vários arquivos abertos de entrada e também de saída.

Seguem os predicados para manipulação de arquivos. Eles são usados em conjunto três para abrir e três para fechar:

- `see(X)` `seeing(X)` `seen` — O `see` abre o arquivo para leitura e o associa com a fita de entrada. Se ele já estiver aberto só será selecionado como fita de entrada. X deve ser um nome válido de um arquivo. O `seeing` retorna um alias do arquivo aberto. O `seen` fecha o arquivo associado à atual fita de entrada.

- `tell(X)` `telling(X)` `told` — O `tell` abre o arquivo para escrita e o associa com a fita de saída. Se ele já estiver aberto, é selecionado somente como fita de saída. O `telling` retorna o alias do arquivo de saída. O `told` fecha o atual arquivo de saída.

10.6.2 Trabalhando com arquivos-texto no padrão Edimburgo

Um predicado comum é o que faz a leitura de uma linha de texto, isto é, os códigos dos caracteres são lidos até encontrarmos os códigos de fim-de-linha: o Ascii 13 é o enter e o Ascii 10 é o caractere de nova-linha. O predicado `readlnCodes1` devolve uma lista com os códigos Ascii, chamando o predicado `get0/1`.

O predicado `fwrite1` cria um arquivo chamado `"teste.pl"` escrevendo duas linhas de texto. O predicado `freadN1` processa um arquivo de texto, linha por linha.

```
1 readlnCodes1(Line) :- get0(Char),readlnC1(Char,Line).
2 readlnC1(13,[]) :- get0(10),!.  % fim nova linha + enter
3 readlnC1(13,[]) :- !.  % fim enter
4 readlnC1(10,[]) :- !.  % fim nova linha
5 readlnC1(Char, [Char|Tail]):-get0(Char1), readlnC1(Char1,Tail).
6 fwrite1 :- tell('teste.pl'),
7            write('primeira linha'),nl,
```

```
8             write('segunda linha'),nl,
9             told.
10 freadN1 :- see('teste.pl'), procFile(0), seen.
11 procFile(N):- at_end_of_file, !. %% nao eh padrao
12 procFile(N):- N1 is N+1,
13           readlnCodes1(C),  % write(C),nl,
14           name(Line,C),
15           write(N1:Line),nl,
16           procFile(N1).
17
```

10.6.3 Entrada e saída no Prolog ISO

O Prolog ISO define um conjunto mais abrangente de primitivas para manipular arquivos, generalizando aquelas apresentadas pelo padrão Edimburgo.

Para abrir e fechar arquivos são oferecidos dois comandos `open` e `close`. Estes comandos possuem vários parâmetros: permitem definir se o arquivo a ser lido é do tipo texto ou do tipo binário (fita de bytes); permitem associar o arquivo a um manipulador (handle). Assim, todo comando de leitura e escrita pode ser opcionalmente parametrizado pelo nome do arquivo (ou manipulador).

O padrão ISO criou também dois comandos para associar arquivos já abertos com as fitas de entrada e saída: `set_input/1` e `set_output/1`. Assim, pode-se ainda trabalhar com um modelo similar ao do padrão Edimburgo. No Prolog ISO, os comandos de abrir arquivos (`see`, `tell` no Edimburgo) são agora executados com a primitiva `open` que tem um parâmetro *modo* que define se o arquivo foi aberto para leitura ou escrita.

Na seqüência, inicialmente mostramos como manipular caracteres e bytes em fitas e depois como ler dados dos arquivos.

10.6.4 Leitura de fitas

No Prolog ISO, os predicados `get` e `put` são generalizados para trabalhar com caracteres, códigos de caracteres (por exemplo, ASCII) e bytes (para arquivos binários):

```
1 get_char/[1,2]     get_code/[1,2]     get_byte/[1,2]
2 put_char/[1,2]     put_code/[1,2]     put_byte/[1,2]
3 peek_char/[1,2]    peek_code/[1,2]    peek_byte/[1,2]
```

Estes predicados podem trabalhar com fitas ou com dispositivos como teclado e vídeo. Por "default"a fita de entrada é associada ao teclado e a fita de saída, ao terminal de vídeo.

Arquivos (ou dispositivos de entrada e saída) para o Prolog são vistos como fitas de bytes ou de caracteres. A unidade mínima de leitura é o byte ou caractere. A leitura como caracteres é mais legível se estamos trabalhando com texto, porém o problema é que muitos caracteres de controle (e.g., enter, tab) não possuem representação gráfica.

Inicialmente, mostraremos como trabalhar com códigos e com caracteres em si. Mais adiante, mostraremos como manipular arquivos binários.

Os predicados `read/1` e `write/1` lêem e escrevem termos, que são os objetos do Prolog. O `write`, quando combinado com o `nl`, escreve linhas de texto.

```
?- write(termo(abc, efg)),nl.
   termo(abc, efg)
?- read(UmTermo).
   fato.<enter>
   UmTermo = fato
```

Um predicado que se faz necessário é o de leitura de uma linha de texto.

Podemos pensar em três possibilidades de retorno para um predicado de leitura de linhas, como segue:

```
?- readlnChars(X).
|    uma linha<enter>
   X = [u, m, a, ' ', l, i, n, h, a]
?- readlnStr(X).
|    uma linha<enter>
   X = 'uma linha'
?- atom_chars(A, [u, m, a, ' ', l, i, n, h, a]).
   A= 'uma linha'
?- atom_codes(C, [117, 109, 97, 32, 108, 105, 110, 104, 97]).
   C='uma linha'
```

Quando trabalhamos com os códigos dos caracteres, usamos o predicado `atomo_codes` para converter listas de códigos em átomos e vice-versa. Quando trabalhamos com os próprios caracteres, usamos o predicado `atom_chars`.

```
1 readlnStr(Alias,S):-read_line_to_codes(Alias,Line),atom_codes(S,B).
2 readlnChars([]) :- peek_code(13),get_code(_),!,(peek_code(10),get_code(_);true).
3 readlnChars([]) :- peek_code(10),get_code(_),!.
4 readlnChars([Char|Tail]):-get_char(Char),readlnChars(Tail).
```

Os predicados `get_code/get_char/get_byte` são similares aos predicados `get/get0` do padrão Edimburgo. Já os predicados `peek_code/peek_char/peek_byte` são novos, e olham o próximo byte na fita de entrada sem avançar o cursor da fita. Quem avança o cursor da fita é o correspondente `get_`. Acima codificamos o `readlnChars` fazendo uso dos predicados `peek_`. Aqui o `readlnStr(Alias,S)` utiliza um predicado que lê uma lista de códigos Ascii e depois converte a lista para um átomo.

10.6.5 Trabalhando com arquivos no Prolog ISO

No Prolog ISO, podemos abrir vários arquivos e trabalhar com todos eles ao mesmo tempo. Uma forma de trabalhar é sempre usar o nome do arquivo nos predicados de entrada e saída, como no predicado `cria2`.

Outra forma de trabalhar é usar o predicado que seleciona a fita antes de iniciarmos as operações de entrada e saída. Esta segunda forma tem a vantagem de podermos testar os predicados nas fitas de entrada e saída e somente quando eles funcionam são associados aos arquivos. Abaixo, temos um exemplo de um predicado `escreve2` que escreve duas linhas. Se executado, ele escreve as duas linhas no terminal de vídeo. Em seguida, o predicado `fwrite` torna o arquivo `teste.pl` como a fita de saída padrão; e, com o mesmo predicado `escreve2` as duas linhas são escritas no

arquivo. Esta forma de trabalho, em que os predicados escrita são independentes do dispositivo de saída, é especialmente vantajosa para testar programas. Somente quando o programa funciona é que enviamos a sua saída para um arquivo — durante os testes usamos o vídeo.

```
cria2 :- open('arqui1.pl',write,_, [type(text), alias(file01)]), %% abre
         open('arqui2.pl',write,_, [type(text), alias(file02)]),
         write(file01, 'linha 1 do arquivo 1'),nl(file01),
         write(file02, 'linha 1 do arquivo 2'),nl(file02),
         write(file01, 'linha 2 do arquivo 1'),nl(file01),
         close(file01),close(file02).
fwrite :- open('teste.pl',write,_, [type(text), alias(file03)]),
          set_output(file03), escreve2, close(file03).
escreve2:-write('primeira linha'),nl,
          write('segunda linha'),nl.
```

Segue o predicado `freadN` que lê um arquivo de linhas de texto e escreve na saída o conteúdo prefixado pelo número da linha. As linhas são lidas pelo predicado `readlnStr`.

```
freadN :- open('teste.pl',read,_,[type(text),alias(alias3)]),
          set_input(alias3), procFileN(alias3,0), close(alias3).
procFileN(A,N):-at_end_of_stream,!.
procFileN(A,N):-N1 is N+1,readlnStr(A,S), write(N1:S),nl,procFileN(A,N1).
```

Segue um teste para o predicado `freadN`.

```
?- freadN.
   1:primeira linha
   2:segunda linha
   Yes
```

Exercício 10.8 *Leia um arquivo e conte o número de linhas e de caracteres. Dica: comece com o programa acima; use o predicado* `length`.

10.6.6 Arquivos binários

Examinando o conteúdo de um arquivo

Para entender como um arquivo de texto pode ser visto como uma seqüência de bytes, executamos o predicado `freadB`.

```
freadB :- open('teste.pl',read,_,[type(binary),alias(file03)]),
          set_input(file03), procFileB(0), close(file03).
procFileB(N):-at_end_of_stream,!.
procFileB(N):-N1 is N+1, get_byte(S),tab(1),write(S),procFileB(N1).
```

Basicamente, na codificação Ascii, cada caractere é representado em um byte. O predicado `get_byte` lê um byte da fita de entrada.

Um mesmo arquivo pode ser aberto como arquivo-texto ou como arquivo-binário. O conteúdo é o mesmo, o que muda é a forma como vemos o conteúdo. Supondo que o arquivo `teste.pl` tem duas linhas com conteúdo *primeira linha* `\n` *segunda linha*, vemos o resultado deste programa abaixo (a tabulação foi feita à mão, para destacar os valores codificados no final de cada linha).

```
?- freadB.
 112 114 105 109 101 105 114 97    32  108 105 110 104 97   13 10
 115 101 103 117 110 100 97        32  108 105 110 104 97   13 10
```

Estes valores para um arquivo-texto podem mudar em um sistema UNIX (o 10 é fim de linha para DOS e o 13 é para UNIX).

Mais sobre arquivos binários

O Prolog ISO permite a representação de valores numéricos binários e hexadecimais. Estes valores, quando manipulados, são considerados inteiros positivos. Num byte, podemos representar a faixa de 0 até 255 e, em dois bytes, de 0 até 65535. Seguem algumas perguntas usando a representação binária e hexadecimal.

```
?- X is 0xFFFF.  X = 65535
?- X is 0x0000.  X = 0
?- X is 0b1111111111111111.   X = 65535
?- X is 0xFFFF >> 8.  X = 255
```

Exercício 10.9 *O valor positivo 1030 pode ser representado em apenas dois bytes. Mostre como convertê-lo em dois bytes usando as operações de manipulação de bytes. Qual é o valor de cada um dos bytes?*

Solução: Podemos fazer um *and* lógico (`/\`) entre o valor 1030 e o valor hexadecimal 0xff00 para remover o valor do byte da esquerda e com 0x00ff para remover o valor do byte da direita.

```
?- X is (1030) /\  (0xff00).
   X = 1024
?- X is (1030) /\  (0x00ff).
   X = 6
?- X is ((1030) /\  (0xff00)) >> 8.
   X = 4
```

O valor do byte da esquerda ainda não pode ser representado em um byte, porque o byte da esquerda foi preenchido com zeros. Portanto, a solução é dar um deslocamento para a direita de 8. Logo, os valores dos dois bytes para formar o valor 1030 é 4 (0b100) para o byte da esquerda e 6 (0b110) para o byte da direita. Podemos recompor o número a partir destes dois bytes, como segue:

```
?- X is 0b110.
   X = 6
?- X is 0B100.
   X is 4
?- X is (0b100 << 8) \/ 0b110.
   X = 1030
?- X is (4 << 8) \/ 6.
   X = 1030
```

Exercício 10.10 *Escreva em um arquivo binário o valor 1030, usando apenas dois bytes. E, depois, leia-os.*

Solução:

```
?- open('testeb.xx',write,_,[type(binary), alias(fbin)]),
        put_byte(fbin, 0b0100),put_byte(fbin, 0b0110),
        close(fbin),!.
?- open('testeb.xx',read,_,[type(binary), alias(fbin)]),
        get_byte(fbin, A),get_byte(fbin, B),
        close(fbin),!.
   A = 4, B = 6
```

Exercício 10.11 *Escreva um predicado que lê um valor inteiro positivo entre 0 e 65535, em seguida, transforme o valor lido em dois bytes e armazene-o em um arquivo binário usando dois bytes. Depois, faça um predicado que lê do arquivo o valor armazenado em 2 bytes, em seguida, escreva o valor na tela como um valor inteiro positivo. Sugestão: estenda o exercício anterior. Uma linha para ler o valor; outra para convertê-lo em dois bytes; outra para converter dois bytes em um valor positivo e escrever o resultado.*

Parte III

Estudos de Casos e Aplicações

CAPÍTULO 11

Programação de Gramáticas

SEMÂNTICO - **Gramáticas de Atributos - GA (tipo 0)**
Definite Clause Grammar - DCG (tipo 0)
Gramáticas sensíveis ao contexto (tipo 1)
|
SINTÁTICO - **Gramáticas livres de contexto (tipo 2)**
|
LÉXICO - **Gramáticas regulares (tipo 3)**

Neste capítulo apresentamos uma classificação das gramáticas baseada na hierarquia de Chomsky. Mostramos como utilizar o formalismo gramatical embutido no Prolog, DCG, para programar gramáticas em três níveis de especificação de linguagens: léxico, sintático e semântico.

11.1 Níveis lingüísticos

Gramáticas são formalismos usados para especificar linguagens. Linguagens são descritas em diferentes níveis, visando facilitar o seu estudo. Na tabela abaixo mencionamos quatro níveis de especificação de linguagens, que se aplicam tanto a linguagens de computadores como a linguagens naturais. Esta classificação em 4 níveis é também chamada de hierarquia de Chomsky. Cada nível lingüístico está associado a um mecanismo de reconhecimento.

tipo	**nome**	**nível lingüístico**	**reconhecedor**
TIPO 0	sem restrições	semântico e pragmático	máquina de Turing
TIPO 1	sensível ao contexto	semântico	gramática de atributos
TIPO 2	livre de contexto	sintático	autômato de pilha
TIPO 3	regular	léxico	autômato

Abaixo exemplificamos estes tipos de linguagens. O objetivo destes exemplos é desenvolver uma intuição sobre a classificação de linguagens:

- regulares: $(0|1)^*$ ={[],0,1,01,10,...}; a^*b^* = {[], a, b, ab, aa, bb, ...};
- livres de contexto: a^nb^n = {ab, aabb, aaabbb,...};
- sensíveis ao contexto: $a^nb^nc^n$ = {abc, aabbcc, ...}; $a^nb^mc^nd^m$ = {abcd, abbcdd, ...}; $\{x\ x \mid x \in \{0,1\}^*\}$;
- sem restrição: $\{h^nf^{n!}\}$ onde o comprimento de f representa a computação do fatorial do comprimento de h, $\{[]f, hf, hhff, hhhfffffff, ..., h^nf^{n!}\}$.

Algumas destas linguagens serão codificadas como gramáticas nas próximas seções, onde estudamos em detalhes três destes níveis lingüísticos:

- O LÉXICO é associado à especificação das palavras de uma linguagem, também chamadas de tokens. Separar uma frase em palavras e símbolos de pontuação ou separar uma linha de comandos em tokens (identificadores, operadores, delimitadores, etc.) são atividades deste nível.
- O SINTÁTICO é associado à construção de frases e comandos. Verificar a sintaxe de uma frase (sujeito, verbo e objeto) ou verificar a sintaxe de uma linha de comandos de uma linguagem de programação são atividades deste nível.
- O SEMÂNTICO estuda a semântica, o significado ou a tradução de uma frase. Traduzir uma linha de comandos para uma linguagem de mais baixo nível ou traduzir frases de uma língua para outra são atividades deste nível.

11.2 Gramáticas em Prolog: DCG

O Prolog possui um mecanismo para processar gramáticas, embutido na sua notação de cláusulas. Este mecanismo, conhecido como DCG (*Definite Clause Grammar*), permite a codificação direta de gramáticas do tipo 1, GAs (Gramáticas de Atributos).

RESULTADO 11.1 *A classe das DCGs tem o mesmo poder computacional da classe das GAs.*

As DCGs e as GAs são baseadas em gramáticas livres de contexto. Uma DCG processa gramáticas livres de contexto **sem recursividade à esquerda**. Por exemplo, a regra `R-->Ra`, que é recursiva à esquerda, deve ser reescrita como `R-->aR` antes de ser codificada como regra DCG. Do mesmo modo, a regra `R-->[]|aR`, com uma alternativa vazia, deve ser reescrita como `R-->aR|[]` onde a alternativa vazia é a derradeira.

11.2.1 Gramática regular

A gramática $R = a^*b^*$, exemplificada anteriormente, é traduzida para as regras DCG que seguem:

```
r --> a, b.
a --> [a],a.
a --> [].
b --> [b],b.
b --> [].
```

Os símbolos terminais são representados entre colchetes. Os não terminais são representados por letras minúsculas (em Prolog maiúsculas são variáveis).

Dada uma DCG, podemos perguntar sobre as sentenças da linguagem gerada pela gramática. Sabemos que a gramática R gera as sentenças {a,b,aa,bb,ab,aab,abb,...} e que as sentenças {ba,aba,...} não são válidas. Portanto, podemos perguntar:

```
?- r([a,b,b],X).
     X=[], Yes
?- r([a,b,a],[]).
     NO
?- r([a,b,a],X).
     X=[a]   Yes
```

As sentenças são representadas em listas de símbolos terminais. Numa pergunta são passados dois parâmetros: uma cadeia de entrada e uma cadeia de saída, respectivamente. Na pergunta `?-r([a, b, b],X)` o retorno é (`X=[]`), significando que toda a entrada foi reconhecida ou consumida. Caso contrário, o conteúdo do argumento de saída é o que sobrou, deixando de ser reconhecido. Em `?-r([a,b,a],X)` o retorno é `X=[a]`, o `"a"` deixou de ser reconhecido.

Podemos testar as produções de maneira isolada, por exemplo, a pergunta `?-b([b,b,b,a,a],X)` retorna `X=[a,a]`. A pergunta é feita para a produção `b`, que reconhece uma seqüência de b(s) e não reconhece dois a(s).

11.2.2 Gramática livre de contexto

Podemos codificar a gramática $L = a^n b^n$, como segue:

```
l --> [a],l,[b].
l --> [].
```

Seguem alguns testes para L:

```
?- l([a,b],X). X=[], Yes
?- l([a,a,a,b,b,b],[]). Yes
?- l([a,b,a],[]). No
```

Podemos transformar uma gramática regular (tipo 3) em livre de contexto (tipo 2) usando equações e atributos de GAs. A gramática regular $R = a^* b^*$, apresentada anteriormente, é codificada em DCG como:

```
r --> a, b.
a --> [a],a.
a --> [].
b --> [b],b.
b --> [].
```

Se adicionarmos um parâmetro na produção principal da gramática regular R, teremos `r-->a(X),b(Y),{X=Y}`, que se torna equivalente à gramática livre de contexto L, `l-->[a],l,[b] | []`.

```
r --> a(X), b(Y),{X=Y}.
a(N+1) --> [a],a(N).
a( 0 ) --> [].
b(N+1) --> [b],b(N).
b( 0 ) --> [].
```

Seguem aqueles mesmos testes de L agora para R:

```
?- r([a,b],X). X=[], Yes
?- r([a,a,a,b,b,b],[]). Yes
?- r([a,b,a],[]). No
```

11.2.3 Gramática sensível ao contexto

A gramática sensível ao contexto $S = a^n b^m c^n d^m$ não pode ser programada por produções do tipo livre do contexto. No entanto, as regras gramaticais DCG podem ser parametrizadas, a fim de processar gramáticas sensíveis ao contexto. Segue uma DCG para a gramática S:

```
s --> a(X),b(Y),c(Z),d(W), {X=Z, Y=W}.
a(N+1)-->[a],a(N).
a(  1)-->[a].
b(N+1)-->[b],b(N).
b(  1)-->[b].
c(N+1)-->[c],c(N).
c(  1)-->[c].
d(N+1)-->[d],d(N).
d(  1)-->[d].
```

Nesta codificação, cada uma das regras `a,b,c,d` calcula o número de ocorrências dos caracteres (tokens), e a equação semântica em `s {X=Z, Z=W}` força a ocorrência do mesmo número de `a`cs e de `b`ds. Seguem alguns testes para a gramática S:

```
?- s([a,b,c,d],X). X=[], Yes
?- s([a,b,b,c,d,d],[]). Yes
?- s([a,b,a],[]). NO
```

Exercício 11.1 *Escreva uma GA pra a linguagem* $\{x\ x \mid x \in \{0,1\}^*\}$.

Solução:

```
GLC:                GA na notação DCG:
S --> X X.          s      --> x(A), x(B),{A=B}.
X --> 0 X.          x([0|N]) --> [0], x(N).
X --> 1 X.          x([1|N]) --> [1], x(N).
X --> [].           x( 0 ) --> [].
```

Seguem alguns testes:

```
?- s([0,1,0,1],X). X=[], Yes
?- s([0,1,1,1],[]). No
```

A notação GA é vista como um formalismo para especificação semântica. Para programar uma especificação em GA devemos escolher uma ferramenta de processamento de gramáticas. Uma boa escolha é o formalismo DCG do Prolog.

Exercício 11.2 *Uma gramática é definida por uma tupla (G,N,T,P). Num código DCG como identificamos G,N,T e P?*

Solução: G é o símbolo inicial, normalmente corresponde a primeira produção. P é o conjunto das produções, normalmente codifica-se cada produção em uma linha. No lado direito da produção os elementos são separados por uma conjunção lógica (,) vírgula. Cada terminal (T) é codificados entre colchetes, no lado direito de uma regra. Todos os não terminais (N) são codificados como cabeças de regras, com pelo menos uma regra, podendo ter várias regras alternativas (normalmente, uma em cada linha terminada por ponto). Duas regras podem ser codificadas com uma mesma cabeça, neste caso os dois corpos da regra são ligados por uma disjunção lógica (;).

11.3 Gramáticas de Atributos

Inicialmente devemos saber que as DCGs implementam uma subclasse bem geral de GAs, com poder suficiente para implementar processadores de linguagens de programação. A grande diferença entre uma GA e uma DCG é que, na primeira, as equações semânticas são avaliadas como atribuições, enquanto na segunda elas são avaliadas com unificação (que é bidirecional – uma atribuição nos dois sentidos).

RESULTADO 11.2 *Uma GA* **apenas** *com atributos sintetizados S-GA tem o poder computacional de uma máquina de Turing.*

Este resultado pode ser interpretado de várias formas. Sabemos que uma gramática irrestrita, na classificação de Chomsky, também equivale a uma máquina de Turing. Portanto, uma GA pode especificar qualquer gramática irrestrita.

A classe de GAs com atributos herdados e sintetizados é mais geral que a classe que contém somente atributos sintetizados. Porém, noutra perspectiva, dada uma GA com atributos herdados e sintetizados, definindo uma computação com valor prático, podemos encontrar uma GA só com atributos do tipo sintetizado que processa a mesma computação.

Na prática, este resultado não é tão útil porque podemos programar GAs com atributos herdados e sintetizados em qualquer linguagem de programação moderna, inclusive em linguagens imperativas, como veremos nos próximos capítulos.

Atributos herdados e sintetizados

Uma GA estende uma gramática livre de contexto associando atributos aos símbolos não-terminais da gramática, que em DCG são os predicados.

RESULTADO 11.3 *Uma DCG com parâmetros equivale a uma GA.*

Abaixo, temos uma S-GA para contar o número de as da fita de entrada usando apenas um atributo do tipo sintetizado. Esta versão usa o atributo M, que recebe o valor de N+1.

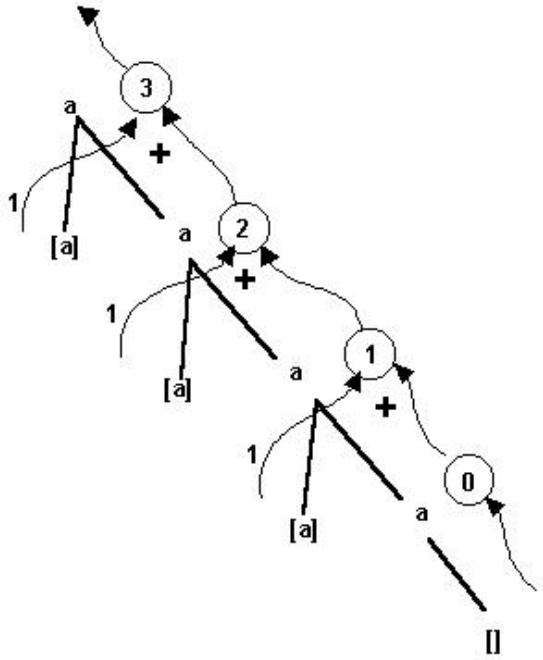


Figura 11.1: Árvore sintática com atributos sintetizados (S-GA), para contar os a(s).

```
1 %% DCG                          %% S-GA em DCG
2 a(N+1) -->[a], a(N).            a(M) --> [a], a(N), {M = N+1}.
3 a( 0 ) -->[].                   a(M) --> [], {M = 0}.
```

Seguem alguns testes para a:

```
?- a(M,[a,a,a],X).
   X=[], M=0+1+1+1,  Yes
?- a(M,[a,a,a,a],[]), Mo is M.
   Mo=4, M=0+1+1+1+1, Yes
```

Os atributos são passados do corpo (RHS) para a cabeça das produções (LHS). Se desenhamos a árvore sintática, vemos que estes atributos sobem pela estrutura da árvore, por isso são chamados de **sintetizados**, ver Figura 11.1.

Já definimos a subclasse S-GA, onde há somente atributos do tipo sintetizado. Outra subclasse importante é a L-GA (Left to right transversal GA). Uma GA é L-GA se suas equações podem ser calculadas durante a análise sintática LL(k), ou durante um caminhamento em profundidade da esquerda para a direita. Segue um exemplo de L-GA:

```
1 %% DCG                          %% L-GA em DCG
2 b0(  M )--> b(0,M).             b0(  M )--> b(0,M).
3 b(AC,M )--> [b], b(AC+1,M).     b(AC,M )--> [b], {AC1=AC+1}, b(AC1,M).
4 b(AC,AC)--> [].                 b(AC,AC)--> [].
5 %% AC é um ACumulador
```

Seguem alguns testes para b0:

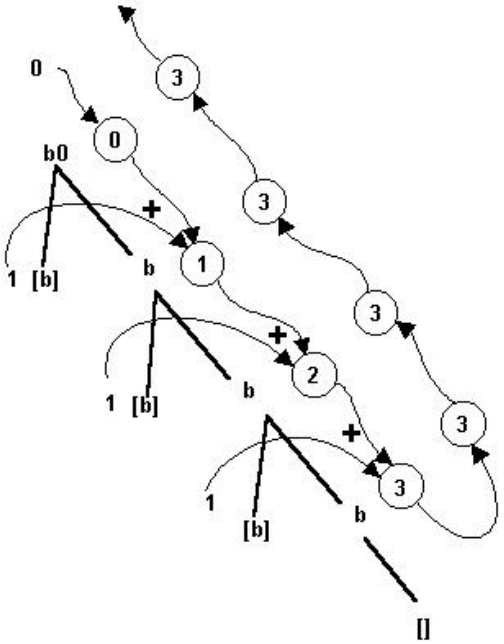


Figura 11.2: Árvore para a sentença "bbb", da gramática L-GA, com atributos herdados (descem) e sintetizados (sobem) para contar os (b)s.

```
?- b(M,[b,b,b],X).
   X=[], M=0+1+1+1,  Yes
?- b(M,[b,b,b,b],[]), Mo is M.
   Mo=4, M=0+1+1+1+1, Yes
```

Estes atributos são mais bem visualizados em uma árvore sintática da gramática. Um atributo herdado desce pela árvore e um sintetizado sobe, ver Figura 11.2. Nesta L-GA temos atributos sintetizados e herdados. O sintetizado é o M e os herdados são o AC e AC1.

EXEMPLO 11.1 (Uma linguagem para robôs)

Exercício 11.3 *Faça uma DCG para uma linguagem de robôs, onde o robô pode se mover apenas em quatro direções: trás, frente, esq, dir. Usando dois parâmetros, gere uma soma para cada um dos sentidos de deslocamento (trás/frente) e (esq/dir). Por exemplo:*

```
?-move(F,D,[esq,dir,esq,frente,frente,dir,dir,pare],[]).
  F= 0+0+0+1+1+0+0+0,
  D=-1+1-1+0+0+1+1+0 Yes
```

Solução:

```
1 move(0,0) --> [pare].
2 move(D,F) --> passo(D,F).
3 move(D+D1,F+F1)--> passo(D,F), move(D1,F1).
4 passo( 1, 0) --> [dir].
5 passo(-1, 0) --> [esq].
6 passo( 0, 1) --> [frente].
7 passo( 0,-1) --> [tras]
```

Exercício 11.4 *A gramática do robô aceita sentenças como* `[frente, frente, pare]` *mas também* `[frente, frente]`. *Reescreva-a para que aceite somente sentenças terminadas por* `[pare]`.

Exercício 11.5 *Calcule os valores para D e F, acrescentando uma regra* `cmove` *que chama* `move`, *sem modificar o* `move`.

Exercício 11.6 *Acrescente a dimensão Z, desce/sobe. Pense num braço mecânico se movimentando em 3D.*

11.4 Avaliar expressões aritméticas

Abaixo apresentamos uma gramática para expressões aritméticas, para as quatro operações básicas: soma, subtração, multiplicação e divisão.

```
1 E --> T+E | T-E | T
2 T --> F*T | F/T | F
3 F --> ( E )
4 F --> 1|2| ...
```

Com esta gramática, podemos gerar árvores abstratas, similares às árvores sintáticas, que tratam da precedência dos operadores, onde as operações de menor precedência estão no topo:

```
   +
  / \
 1   *
    / \
    2 3
```

No nível semântico, podemos definir uma GA para avaliar estas expressões. Abaixo é definida uma gramática de atributos com equações semânticas para calcular o valor de cada expressão, por exemplo, o valor de `(20+4)*4/8` que é 12.

E_1 --> T+E_2	{E_1.val:=T.val+E_2 .val}
E_1 --> T-E_2	{E_1.val:=T.val-E_2 .val}
T_1 --> F*T_2	{T_1.val:=F.val*T_2 .val}
T_1 --> F/T_2	{T_1.val:=F.val/T_2 .val}
F --> (E)	{F.val := E.val}
F --> 1	{F.val := 1}
F --> 2	{F.val := 2}
...	...

Esta gramática de atributos define uma semântica para o cálculo do valor das expressões geradas pela linguagem. Uma equação é definida para cada produção (unidade sintática). Por exemplo, a equação {F.val = 1} associada à produção F-->1 é lida como *o atributo* `val` *do* `F` *recebe* 1. De forma similar, a equação {E_1.val=T.val+E_2 .val} associada à produção E_1 --> T+E_2 é lida como *o atributo* E_1.val *recebe a soma dos atributos* T.val *e* E_2.val. Aqui, por meio das equações podemos identificar que o atributo `val` é sintetizado porque flui em direção pai (se for examinado numa árvore sintática).

Exercício 11.7 *Desenhe uma árvore sintática para a sentença* 1+2*3, *decorada com os atributos, para a gramática de atributos definida acima.*

11.4.1 Programando a GLC como DCG

Segue a gramática livre de contexto, codificada em DCG, para a linguagem de expressões; usamos a codificação E=expr, T=Termo e F=Fator:

```
1 expr --> termo,[+],expr.
2 expr --> termo,[-],expr.
3 expr --> termo.
4 termo--> fator,[*],termo.
5 termo--> fator,[/],termo.
6 termo--> fator.
7 fator --> [X],{integer(X)}.
8 fator --> ['('], expr, [')'].
```

A produção `fator --> [X],{integer(X)}` define uma regra válida para todos os inteiros da linguagem Prolog. O predicado `integer/1` testa se uma constante é do tipo inteiro, conforme exemplo: `?- integer(10). Yes ?- integer(a). No`

A gramática com símbolo inicial `expr` é um programa executável em Prolog. Podemos perguntar se uma expressão é reconhecida por ela: `?- expr([1,+,2,*,3],X). X=[], Yes.`

11.4.2 Calculando o valor com equações semânticas

Abaixo, apresentamos a versão em Prolog desta gramática, estendida com atributos e equações semânticas que calculam o valor da expressão aritmética.

A sintaxe do Prolog difere um pouco da notação de GA. Numa GA um atributo é associado a um símbolo não-terminal. Dois símbolos com mesmo nome numa produção são indexados por um dígito diferenciador. No entanto, em Prolog, estes índices são associados a variáveis, por exemplo, E1. De qualquer modo, a semântica das equações é a mesma.

```
1  expr(E)--> termo(T),[+],expr(E1),{E is T+E1}.
2  expr(E)--> termo(T),[-],expr(E1),{E is T-E1}.
3  expr(T)--> termo(T)
4 termo(T)--> fator(F),[*],termo(T1),{T is F*T1}.
5 termo(T)--> fator(F),[/],termo(T1),{T is F/T1}.
6 termo(F)--> fator(F).
7 fator(X)--> [X],{integer(X)}.
8 fator(E)--> ['('], expr(E), [')'].
```

Seguem algumas perguntas onde é retornado o valor da expressão.

```
?-expr(V,[1,+,2,*, 3],X).
   V=7, X=[]
?- expr(V,[1,+,2,*, '(',3,+,1, ')'],X).
   V=9, X=[],
?- expr(V,['(',20,+,4,')', *, 4, /, 8],X).
   V=12, X=[]
```

11.4.3 O problema da associatividade à esquerda para LL(k)

Esta implementação apresenta um problema de associatividade que aparece em operações não associativas. Por exemplo, em seqüências de divisões e em seqüências de somas e/ou subtrações, como vemos abaixo.

```
?- expr(V,[1,/,2,/,4,*,3],[]), display(V), X is V.
     /(1, /(2, *(4, 3)))
     X = 6
?- expr(V,[1,-,2,+,4,-,3],[]), display(V), X is V.
    -(1, +(2, -(4, 3)))
    X = -2
```

A primeira expressão 1/2/4*3 com parênteses é (1/(2/(4*3)))=6, porém, o certo é escrevê-la como (((1/2)/4)*3)=0.375. Este problema de associatividade acontece com gramáticas recursivas à direita, as LL(k). Estas gramáticas geram (naturalmente) árvores abstratas associativas à direita. No capítulo sobre *um compilador* é apresentada uma solução geral para este problema.

11.5 Gerando notação polonesa com ações semânticas

Abaixo, apresentamos outra versão desta gramática com **ações semânticas** de escrita. Escreve-se na saída o código em notação polonesa para a expressão.

Equações semânticas são diferentes de ações semânticas. As **equações semânticas** definem relações entre atributos, são locais a uma regra gramatical, mais formais e mais declarativas, como as usadas na gramática de atributos para cálculo do valor da expressão. Já as **ações semânticas** são mais procedurais. Por exemplo, quando elas envolvem entradas e saídas possuem efeito colateral, uma vez escrito um valor, a escrita não pode ser desfeita. Portanto, programas com ações semânticas necessariamente devem ser **fatorados** para não haver retrocesso. Por causa disso, fatoramos as produções da gramática de expressões. Cada produção começa com a parte comum (termo), seguida de uma produção com várias alternativas para a parte diferenciada (rtermo) – resto do termo.

```
1 expr --> termo,rexpr.
2 rexpr --> [+],expr, {write(some),nl}.
3 rexpr --> [-],expr, {write(subt),nl}.
4 rexpr --> [].
5 termo--> fator,rtermo.
6 rtermo--> [*],termo, {write(mult),nl}.rtermo--> [/],termo, {write(divi),nl}.
7 rtermo--> [].
8 fator --> [X],{integer(X)},{write(X), write(' enter'), nl}.
9 fator --> ['('], expr, [')'].
```

O efeito das ações semânticas é escrever uma seqüência de passos a serem executados numa calculadora do tipo HP. Esta notação para representar expressões sem parênteses é também chamada de notação polonesa, como segue:

```
?- expr([10,+,20,*,33],[]).
10 enter
20 enter
```

```
33 enter
mult
some
?- expr([1,-,2,+,4,-,3],[]).
1 enter
2 enter
4 enter
3 enter
subt
some
subt
```

Exercício 11.8 *Desenhe uma árvore sintática decorada com as ações semânticas para a gramática versão fatorada que gera a notação polonesa, para a sentença* 1+2*3.

CAPÍTULO 12

Romanos para decimais e Decimais por extenso

Neste capítulo mostramos dois exemplos de programas com gramáticas DCG. O propósito dos exemplos é didático: mostrar como funciona internamente a DCG, como um programa DCG pode ser convertido para cláusulas.

12.1 Romanos para decimais e vice versa

Uma definição simples para números romanos é valida para valores no intervalo de 1 até 3999. Não é possível expressar o zero em romanos. Seguem dois exemplos de conversão de decimal para romano e vice-versa.

```
?- romano_(1947,V).
V = 'MCMXLVII'

?- decimal_('MCMXLVII',V), Vo is V.
V = 1000+ (900+ (40+ (5+ (1+ (1+0)))))
Vo = 1947
```

O programa abaixo faz a conversão de romanos para decimais e vice-versa. O conjunto de regras serve para os dois predicados, porém eles em si variam um pouco. O programa todo não pode ser reversível pois envolve operações aritméticas que em princípio não são reversíveis.

As regras devem ser executas de cima para baixo. Note que é uma gramática não LL(1): por exemplo, existem três regras em que o corpo inicia com 'C'. Poderiam ser fatoradas, mas o exemplo serve para mostrar que as DCGs trabalham com gramáticas LL(k).

```
1 %% file:romano.pl
2 %% romano para valor
3 r(V)-->romam(V1),r(V2),{V = V1+V2}.
```

```
r(0)-->[].
decimal_(X,V):-atom_chars(X,L),r(V,L,[]),!.
 romam(1000)-->['M'].
 romam(900)-->['C'],['M'].
 romam(500)-->['D'].
 romam(400)-->['C'],['D'].
 romam(100)-->['C'].
 romam(90)-->['X'],['C'].
 romam(50)-->['L'].
 romam(40)-->['X'],['L'].
 romam(10)-->['X'].
 romam(9)-->['I'],['X'].
 romam(5)-->['V'].
 romam(4)-->['I'],['V'].
 romam(1)-->['I'].
%% valor para romano
romano_(V,R):-xr(V,L,[]),!,atom_chars(R,L).
xr(V) --> romam(V1),{V1=<V, Vo is V - V1}, xr(Vo).
xr(0) --> [].
```

Para entendermos um pouco mais sobre DCGs, vamos mostrar como codificar este programa em cláusulas, isto é, sem usar o recurso gramatical DCG.

Segue o código. Para economizar espaço não reescrevemos todas as regras. Deixamos para o leitor completar as regras bem como o predicado `roman_`, a partir da versão anterior. Existe uma correspondência de uma linha para uma linha entre a versão DCG e esta versão em cláusulas. Em cada cláusula são acrescidos dois parâmetros, as fitas de entrada e saída.

```
%% romano para valor
decimal_(X,V):-atom_chars(X,L),r(V,L,[]),!.
r(V,I,O):-romam(V1,I,I1),r(V2,I1,O),V = V1+V2.
r(0,I,I).
%% ?- decimal_('MCMII',V), Vo is V.
 romam(1000,I,O):-tok('M',I,O).
 romam(900,I,O):-tok('C',I,I1),tok('M',I1,O).
 romam(500,I,O):-tok('D',I,O).
 romam(400,I,O):-tok('C',I,I1),tok('D',I2,O).
 romam(100,I,O):-tok('C',I,O).
 romam(1,I,O)  :-tok('I',I,O).
   tok(X,I,O)  :-I=[X|O].
```

Agora, abaixo, podemos ver como são traduzidas as regras DCG para cláusulas pelo tradutor do Prolog. O código abaixo é da versão DCG do programa. Em relação a versão anterior, o Prolog substitui o `tk(X,I,O)` por `I=[X|O]` na cabeça das cláusulas.

```
?- [romano].
?- listing(romam).
romam(1000, ['M'|A], A).
romam(900, ['C'|A], B) :- A=['M'|B].
romam(500, ['D'|A], A).
romam(400, ['C'|A], B) :- A=['D'|B].
```

```
romam(100, ['C'|A], A).
romam(90, ['X'|A], B)  :- A=['C'|B].
romam(50, ['L'|A], A).
romam(40, ['X'|A], B)  :- A=['L'|B].
romam(10, ['X'|A], A).
romam(9, ['I'|A], B)   :- A=['X'|B].
romam(5, ['V'|A], A).
romam(4, ['I'|A], B)   :- A=['V'|B].
romam(1, ['I'|A], A).
```

Por fim, devemos dizer que é possível codificar uma versão do programa com `tok/3` para uma linguagem imperativa. Primeiro reescrevemos as cláusulas removendo os valores inteiros da cabeça de cada cláusula para o seu corpo, como segue:

```
1 romam(X,I,O):-tok('M',I,O),                {X=1000}.
2 romam(X,I,O):-tok('C',I,I1),tok('M',I1,O),{X=900}.
3 romam(X,I,O):-tok('D',I,O),                {X=500}.
4 romam(X,I,O):-tok('C',I,I1),tok('D',I2,O),{X=400}.
5 romam(X,I,O):-tok('C',I,O),                {X=100}.
6 romam(X,I,O):-tok('I',I,O),                {X=1}.
7   tok(X,I,O):-I=[X|O].
```

Agora o predicado `romam` pode ser visto como uma função booleana, numa linguagem imperativa, como segue:

```
1 function romam(var V:int; I:int; var O:int):bool;
2 begin  roman := tok('M',I,O)  and attr(X,1000) or
3                 tok('C',I,I1) and tok('M',I1,O) and attr(X,900) or
4                 ...
5                 tok('I',I,O) and attr(X,1); end;
6 function tok(T:char; I:int; var O:int):bool;
7 begin  tok:= (fita[I]=T) and attr(O,I+1); end;
8 function attr(var X:int; V:int):bool; begin X:=V; attr:=true;end;
```

Desse jeito se traduz um programa em DCG (Prolog) para um programa Prolog-like em Pascal. Depois que alguém passa anos programando em Prolog acaba fazendo este tipo de programa até nas linguagens imperativas. Esta tradução vale também para Java e outras linguagens imperativas. Para completar o estudo de gramáticas, veremos outro exemplo pedagógico na seção que segue.

Exercício 12.1 *Este programa procedural opera com retrocesso? É um programa LL(2)? Justifique.*

12.2 Traduzindo decimais por extenso e vice-versa

O programa abaixo traduz decimais por extenso e vice-versa. É um programa reversível: as mesmas regras podem gerar o valor decimal ou o valor por extenso, depende de qual parâmetro é fornecido na pergunta, como segue:

```
?- dx([cento,e,trinta,e,um],V,[]).
V = [1, 3, 1]

?- dx(C,[3,1,2,3],[]), dx(C,L,[]).
C = [tres, mil, e, cento, e, vinte, e, tres]
L = [3, 1, 2, 3]
```

Segue o programa, com um predicado para unidades, dezenas, centenas e milhar. Apresentamos somente parte das regras, deixando indicações de como completá-lo. Completando-se as dezenas, centenas, ... este programa funciona para valores até 9999.

```
1  dx(X)-->dddd(X);ddd(X);dd(X);d(X).
2  d([zero])-->[0].
3  d([um])-->[1].
4  d([dois])-->[2].
5  d([tres])-->[3].
6  %%..
7  dd([dez])-->[1,0].
8  dd([onze])-->[1,1].
9  %%..
10 dd([vinte])-->[2,0].
11 dd([vinte,e|D])-->[2],d(D).
12 dd([trinta])-->[3,0].
13 dd([trinta,e|D] )-->[3],d(D).
14 %%..
15 ddd([cem])-->[1,0,0].
16 ddd([cento,e|DD])-->[1],dd(DD).
17 ddd([duzentos])--->[2,0,0].
18 ddd([duzentos,e|DD])-->[2],dd(DD).
19 %%...
20 dddd([mil])-->[1,0,0,0].
21 dddd([mil, e|DDD])-->[1],ddd(DDD).
22 dddd([D, mil, e|DDD])-->d([D]),ddd(DDD).
```

Exercício 12.2 *Programe uma versão deste programa numa linguagem imperativa, tipo Pascal, C++ ou Java. Numa linguagem imperativa são necessárias duas versões, pois as funções não serão reversíveis: uma gera o extenso e a outra reconhece o extenso gerando o decimal.*

CAPÍTULO 13

Tokens para expressões aritméticas

Uma tarefa comum para processadores de linguagens é transformar uma linha de texto em uma lista de tokens. Por exemplo, abaixo temos um predicado tokens que transforma um string em uma lista de tokens.

```
?- tokens(L,"11  +3*(23)",[]).
L = [int(11), simb(+), int(3), simb(*), simb('('), int(23), simb(')')] Yes
```

Neste exemplo temos dois tipos de tokens: inteiros e símbolos. Os inteiros são seqüências de dígitos (11, 3 e 23); os símbolos gráficos de dois tipos: operadores e separadores. Exemplos de operadores são (+ *). Os separadores são os parêntese e também os caracteres não-visíveis (e.g., espaços em branco).

A gramática resultante, ver abaixo, possui sete não-terminais com 13 produções. Um token (`tok`) é um inteiro ou um símbolo. Uma fita de tokens é uma seqüência de tokens, que podem estar separados por uma seqüência de brancos(`br`). Os tokens são escritos na saída e os brancos são desprezados.

Na Figura 13.1, temos um autômato equivalente. Na passagem da gramática para o autômato, alguns não-terminais não são representados como estados: temos sete não-terminais e apenas quatro estados – não temos os estados `int`, `br` e `tok`. Estes estados podem ser criados com uma transição vazia partindo do estado `toks` – portanto também podem ser eliminados, excluindo-se as três transições vazias.

```
1   tokens  --> tok, rtokens.      int --> dig, rint.
2   tokens  --> br,  rtokens.     rint --> dig, rint.
3  rtokens  --> tok, rtokens.     rint --> [].
4  rtokens  --> br,  rtokens.       br --> branco, rbr.
5  rtokens  --> [].                rbr --> branco, rbr.
6  tok      -->     int.           rbr --> [].
7  tok      --> simbolo.
8  %%
9  branco       --> [C],{C<33}.  %% [32]; [8]; [10]; [13].
```

```
10      dig( C ) --> [C],{C>47, C<58}.
11  simbolo([D]) --> [D],{member(D, "*+/-()")}.
```

Esta gramática na notação DCG do Prolog é executável. As três últimas regras (`dig`, `simbolo` e `branco`) possuem ações semânticas que verificam restrições sobre os caracteres da fita de entrada. Na definição destas produções, assumimos que fita de entrada é uma cadeia de códigos Ascii. A maneira mais econômica de escrever cadeias de códigos Ascii é usando aspas:

```
?- "0123 89"=X.  X= [48, 49, 50, 51, 32, 56, 57]
```

Em `"0"=[48]`, o valor Ascii do zero é 48. No Prolog ISO vale também a notação `0'0=48`. Todos os dígitos estão entre os valores Ascii 48 e 57, o que resulta na produção
`dig(C)-->[C],{C>47, C<58}`.

Numa DCG, as equações semânticas são codificadas entre chaves `{}`. Assim, a produção `dig` é válida somente se o valor de `C` estiver dentro do intervalo especificado. Os separadores são os caracteres de valor Ascii menor ou igual a 32 (do branco). Este conjunto inclui o caractere de tabulação (8) e os caracteres de fim de linha (13 e 10).

Os símbolos gráficos (operadores e delimitadores) são definidos por uma lista `"+-*/()"`; com `member` testa-se se o caractere pertence a esta lista.

```
 1   int([C|W]) --> dig(C),!, rint(W).
 2  rint([C|W]) --> dig(C),!, rint(W).
 3  rint(   []) --> [].
 4   br --> branco,!, rbr.
 5  rbr --> branco,!, rbr.
 6  rbr --> [].
 7   tokens([H|T]) --> tok(H),!,rtokens(T).
 8   tokens(   T ) --> br,    !,rtokens(T).
 9  rtokens([H|T]) --> tok(H),!,rtokens(T).
10  rtokens(   T ) --> br,    !,rtokens(T).
11  rtokens(   []) --> [].
12  tok(int(T))  -->     int(L),!,{name(T,L)}.
```

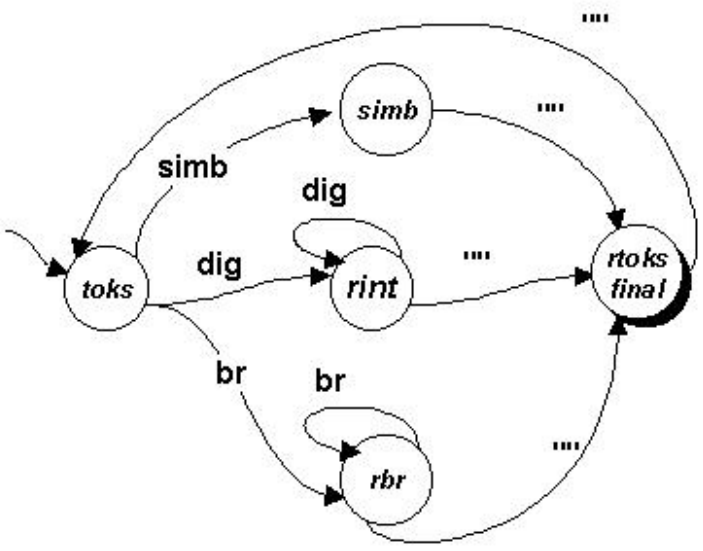


Figura 13.1: Autômato finito para tokens de expressões aritméticas.

```
13 tok(simb(T)) --> simbolo(L),!,{name(T,L)}.
14 branco       --> [C],{C<33}.  %% [32]; [8]; [10]; [13].
15     dig( C ) --> [C],{C>47, C<58}.
16 simbolo([D]) --> [D],{member(D, "*+/-()")}.
```

Na prática, sempre que implementamos uma gramática regular, são necessárias algumas ações semânticas para gerar os tokens e para manipular os atributos e parâmetros das produções. As ações semânticas são escritas entre chaves e naturalmente integradas às regras DCG do Prolog. Por exemplo, na gramática acima, é usado um parâmetro nas produções que definem os inteiros, visando acumular a seqüência de dígitos de um inteiro reconhecido. De forma similar, guarda-se o valor de um símbolo. O predicado `name` é usado para transformar uma lista de códigos Ascii num símbolo. Por exemplo, `?-name([48,49],X). X='01'.` `name` é sinônimo de `atom_codes`.

Nos predicados acima, o operador de corte foi introduzido em todas as produções, tornando o programa determinístico.

13.1 Testando o programa

Para testar uma gramática, devemos começar com produções isoladas, por exemplo, `int`, `br`, `simbolo`.

```
?- int(V,"12",D).
   V = [49, 50] D = [] Yes
?- br("  ",G).
   G = [] Yes
?- simbolo(D,"((",H).
D = [40] H = [40] Yes
```

Depois podemos testar o gerador de tokens com uma expressão aritmética.

```
?- tokens(L,"11  +3*(23)",[]).
L = [int(11), simb(+), int(3), simb(*), simb('('), int(23), simb(')')] Yes
?- tokens(L,"11  +3*(23+32*(3/45)-1)",[]).
  L=[int(11),simb(+),int(3),simb(*),simb('('),int(23),...] Yes
L = [11, +, 3, *, '(', 23, ')'] Yes
```

Exercício 13.1 *Abaixo, temos uma saída para léxico em que separamos os operadores (*,+) dos delimitadores ('(', ')'). Modifique o programa DCG para gerar esta saída.*

```
?- tokens(L,"11  +3*(23)",[]).
   L = [int(11), operador(+), int(3),
        operador(*), delimitador('('), int(23),  delimitador(')' ]
```

No capítulo sobre *um compilador* este assunto é retomado numa maior profundidade.

CAPÍTULO 14

Calcular expressões aritméticas com variáveis

Neste capítulo, a minilinguagem LET é usada para atender objetivos pedagógicos. O estudo desta minilinguagem é interessante, pois, ela exige uma tabela de símbolos com contexto para armazenar as variáveis parciais usadas numa expressão aritmética. Ela permite calcular expressões LET aninhadas como as que seguem:

```
let a=4+5, b=8+2
    in a + b
VALOR=(4+5)+(8+2) = 19

let c= (let a=4+5, b=8+2     %% aqui o parêntese é opcional
           in a + b),        %% porém facilita a leitura
    d=8+2
  in (c+d)*c+d
VALOR=(4+5+ (8+2)+ (8+2))* (4+5+ (8+2))+ (8+2)= 561
```

Abaixo, temos uma gramática para estas expressões. Primeiro codificamos dois predicados para implementar uma tabela de símbolos, como uma lista de pares `par(VAR,VAL)`:

- `lookUp/2` — retorna o valor para uma variável;
- `insert/2` — insere um `par(VAR,VAL)` na tabela de símbolos.

Trabalhamos com expressões simples, do tipo `(c+d)*c+d`, sem as operações de subtração e divisão por causa do problema da associatividade, já discutido no capítulo sobre gramáticas. Naquele capítulo vimos produções para calcular o valor de expressões numéricas sem variáveis. Aqui, para tratar variáveis, todas as produções recebem um atributo herdado: a tabela de símbolos. Ela é necessária para armazenar os valores associados às variáveis que acontecem numa expressão.

Além das produções de expressões, são usadas três novas produções `let`, `decVar` e `decVars`. A produção `let` define uma expressão composta de declaração e o corpo da expressão onde as

declarações serão usadas. A produção `decVar` declara uma variável associada a uma expressão `Var=Exp`; a variável é incluída na tabela de símbolos. A produção `decVars` chama repetidas vezes à produção `decVar`.

```
1  %% let.pl
2  lookUp(X,T):-member(X,T).
3  insert(X,Ti/To):-To=[X|Ti], write((tab:To)),nl.
4  isLetra(X):-member(X,[a,b,c,d,e,f,g,h,i,x,y,z]).
5  %%
6  let(Ti,V) --> [let], decVars(Ti/T1), [in], expr(T1,V).
7  decVars(Ti/To) --> decVar(Ti/T1), [','], decVars(T1/To).
8  decVars(Ti/To) --> decVar(Ti/To).
9  decVar(Ti/To) --> [L],{isLetra(L)}, [=], expr(Ti,E),
10                       {insert(par(L,E),Ti/To)}.
11 expr(TAB,E)--> let(TAB,E).
12  expr(TAB,E)--> termo(TAB,T),[+],expr(TAB,Eo),{E = (T+Eo)}.
13  expr(TAB,E)--> termo(TAB,E).
14 termo(TAB,T)--> fator(TAB,F),[*],termo(TAB,To),{T = (F*To)}.
15 termo(TAB,F)--> fator(TAB,F).
16 fator(TAB,X)--> [X],{integer(X)}.
17 fator(TAB,E)--> ['('],expr(TAB,E), [')'].
18 fator(TAB,V)--> [X],{member(X,[a,b,c,d,e,f,g,h,i,x,y,z])},
19                       {lookUp(par(X,V),TAB), write((look:X:V)),nl}. %% vars
20 %% ?- teste(1,LET),let([],V,LET,RESTO), VAL is V.
21 teste(1, [let, a,=,4,+,5,',',b,=,8,+,2,
22              in, a, +, b]).
23 teste(2, [let, c,=,let, a,=,4,+,5,',',b,=,8,+,2,
24                     in, a, +, b,',',
25               d,=,8,+,2,
26             in, '(', c, +, d,')', *, c, +, d ]).
```

No final do programa dois testes foram codificados como predicados a fim de facilitar a digitação das expressões LET aninhadas. Segue a execução destes dois testes. No programa incluímos dois `write(s)` para ver o que acontece com a tabela de símbolos.

```
?- teste(1,LET),let([],V,LET,RESTO),VX is V.
 tab:[par(a, 4+5)]
 tab:[par(b, 8+2), par(a, 4+5)]
 tab:[par(b, 8), par(a, 4+5)]
 tab:[par(b, 8+2), par(a, 4+5)]
 look:a:4+5
 look:a:4+5
 look:b:8+2
 look:b:8+2
 look:b:8+2
 look:b:8+2
LET = [let, a, =, 4, +, 5, (','), b, =|...]
V = 4+5+ (8+2) RESTO = []
VX = 19

?- teste(2,LET),let([],V,LET,RESTO), VX is V.
```

```
 tab:[par(d, 8+2), par(c, 4+5+ (8+2))]
 look:c:4+5+ (8+2)
 look:d:8+2
LET = [let, c, =, let, a, =, 4, +, 5|...]
V = (4+5+ (8+2)+ (8+2))* (4+5+ (8+2))+ (8+2) RESTO = []
VX = 561
```

Exercício 14.1 *Fatore o programa da gramática. Lembre que depois de fatorar é necessário ajustar as equações semânticas. Inclua corte, onde for necessário, para que ele trabalhe de forma determinística. Veja exemplo no capítulo sobre gramáticas.*

Exercício 14.2 *Implemente uma técnica de diagnóstico para erros sintáticos, que mostra até onde o programa reconheceu a fita de entrada. Trabalhe sobre uma versão fatorada do problema.*

Exercício 14.3 *Implemente um diagnóstico para erros semânticos. As variáveis declaradas em* `vars` *numa expressão* `let vars in expr` *só podem ser usadas num contexto mais interno* `in expr`*. Seguem abaixo dicas para diagnóstico de erros semânticos:*

```
  let a=b+5, b=8-2  /** em a=b+5 a variável b ainda não foi declarada **/
     in let c=a+b, d=a+a+3
            in (c+d)*(c+d)/2

 let a=b1+5, b=let k=2+3 in k+k
      in (b+c+k)           /** em b+c+k a variável k já não existe **/
                           /** ela é local ao outro let            **/
let a=5, b=8-2
     in let a=a+1
            in a+b   /** esta expressão é válida e o aqui o a=6 **/
                     /** vale a declaração mais interna, já funciona **/
```

Por fim, modifique o predicado do `lookUp` para dar o diagnóstico. Como mostrado nos exemplos.

CAPÍTULO 15

Um compilador: tradutor e interpretador

Neste capítulo, apresentamos um estudo de caso de um tradutor para uma minilinguagem baseada na linguagem Pascal. O código gerado, uma árvore sintática abstrata, é executado por um interpretador.

A idéia é explorar técnicas das disciplinas de compiladores e linguagens formais: como implementar um léxico (scanner), um analisador sintático e um interpretador. O analisador sintático é bem poderoso: ele faz uso de analisador de precedência de operadores guiado por tabela (a especificação é na forma de uma tabela). Todos os componentes fazem uso de DCG, mostrando o poder expressivo das gramáticas DCG para implementar processadores de linguagens (ferramentas para processar linguagens).

A construção de um compilador é uma tarefa complexa, por isso é usualmente decomposta em três principais subtarefas:

- análise léxica;
- análise sintática;
- análise semântica e geração de código.

Estes três níveis são apresentados neste estudo de caso. A análise léxica é feita por um componente (programa); a análise sintática é feita por dois componentes: um só para tratar expressões e o outro para os comandos em si. A geração de código está integrada aos componentes sintáticos, é guiada pela análise sintática. Por fim, o código gerado (árvore sintática) é interpretado, como se fosse por uma máquina abstrata.

Sugerimos dois livros como material complementar, o primeiro é um livro clássico sobre compiladores e o segundo é um livro sobre Prolog com um capítulo sobre compiladores: (1) A. Aho, R. Seti. e J. Ulmman. [1] *Compilers — Principles, Techniques, Tools*; (2) L. Sterling e E. Shapiro, [16] *The Art of Prolog*.

15.1 A minilinguagem

Abaixo, apresentamos um programa exemplo da minilinguagem: calcula o fatorial.

```
program fatorial;
var   fat, valor;
begin
    write('Calcula o fatorial'); nl;
    write(' valor=');, read(valor);
    fat := 1;
    while valor > 0 do
      begin
        fat := fat * valor;
        valor := valor - 1;
      end;
     write('o fatorial eh:', fat); nl;
     write('FIM');
end.
```

A minilinguagem tem três tipos de dados: inteiros, booleanos e strings. Todas as variáveis declaradas são assumidas do tipo inteiro. Expressões booleanas são usadas nos comandos `if` e `while`, não em declarações. Strings são usados nos comandos `read` e `write`, não em declarações.

Ao lado direito do comando de atribuição podemos ter expressões aritméticas complexas, que são analisadas pelo parser que verifica a precedência e a associatividade dos operadores.

Temos um comando `read` que lê valores para uma ou mais variáveis inteiras; um comando `write` que escreve uma lista de strings e/ou inteiros; e, um comando `nl` que escreve o fim de linha (como no Prolog).

15.2 O scanner: análise léxica

Além de números inteiros, delimitadores e palavras reservadas, o tokenizer deve processar os identificadores que, para a linguagem Pascal, iniciam com uma letra opcionalmente seguida de letras ou dígitos, por exemplo *aa123bb*.

A partir de um arquivo fonte, o scanner gera a lista de tokens para o programa, como a lista exemplificada abaixo (armazenada no fato `teste/2`). As palavras reservadas são representadas como `r(prog)` e `r(const)`; os identificadores como `id(fatorial)` e `id(valor)`; os números como `num(105)`; os operadores e os parênteses como: `':='`, `')'`, etc.

```
teste(fat,
[r(prog),id(fatorial),;,
r(var), id(valor),',',id(fat),;,
 r(begin),
    r(write),'(',str('Calcula o fatorial'),')',;,r(nl),;,
    r(write),'(',str(' valor='),')',;,
    r(read),'(',id(valor),')',;,
    id(fat),:=,num(1),;,
    r(while),id(valor),>,num(0),r(do),
      r(begin),
        id(fat),:=,id(fat),*,id(valor),;,
```

```
12          id(valor),:=,id(valor),-,num(1),;,
13      r(end),;,
14      r(write),'(',str('o fatorial eh:'),',',id(fat), ')',;,r(nl),;,
15      r(write),'(',str('FIM'),')',;,
16  r(end),'.']).
```

O scanner pode também gerar o número de linha associado a cada token, como no segundo exemplo abaixo. Este número serve para indicar a posição de um erro.

```
?- l1.
1, [r(program), id(fatorial), (;)]
2, [r(var), id(fat), (','), id(valor), (;)]
3, [r(begin)]
...
?- l2.
66:[ (1, r(program)), (1, id(fatorial)), (1, (;)), (2, r(var)), (2, id(fat)),
(2, ','), (2, id(valor)), (2, (;)), (3, r(begin)), (4, r(write)),
...
?- l3.
66:[r(program), id(fatorial), (;), r(var), id(fat), (','), id(valor), (;),
r(begin), r(write), '(', str('Calcula o fatorial'), ')', (;), r(nl), (;),
r(write), '(', str(' valor='), ')', (;), r(read), '(', id(valor), ')', (;),
...
```

Segue abaixo o código do programa scanner, o `readln-elf`. Ele é inspirado no código do predicado `readln` do SWI-Prolog, que lê os tokens de uma fita de entrada, ver exemplo abaixo. Em relação à versão citada nós acrescentamos reconhecimento de strings (com apóstrofos e com aspas) e de símbolos tais como <=. Outra diferença é que nossa versão é codificada em DCG.

```
?-readln(X).
 |: a:='Alo Mundo'
   X = [a, :, =, '\'', 'Alo', 'Mundo', '\'']
```

O predicado `file_to_tokens/2` lê linha a linha um arquivo, retornando o úmero da linha e uma lista de tokens. Como mostrado acima, com `l1, l2, l3`, pode-se retornar os tokens de cada linha, uma fita de tokens com número da linha ou uma fita só de tokens, respectivamente em `l1, l2, l3`.

São usados dois predicados, `zipper_dl` e `remN`, para tratar o número da linha. O `zipper_dl` propaga o número da linha para uma lista de tokens; o `remN` remove os números de linhas, retornando uma lista só com os tokens.

```
?- X=[(29, [a,b,c]), (30, [d,e]), (31, [x])],zipper_dl(X,W/[]),length(W,WL).
X = [ (29, [a, b, c]), (30, [d, e]), (31, [x])]
W = [ (29, a), (29, b), (29, c), (30, d), (30, e), (31, x)] WL = 6

?- X=[(29, [a,b,c]), (30, [d,e]), (31, [x])],remN(X,W),length(W,WL).
X = [ (29, [a, b, c]), (30, [d, e]), (31, [x])]
W = [a, b, c, d, e, x] WL = 6
```

Um identificador pode ser uma palavra reservada, por isso, no início do código são listadas as palavras reservadas, que são usadas para diferenciar uma palavra reservada de um identificador.

```
%% readln-elf.pl
%% ?- l1. ?- l2. ?- l3.
l1:- file_to_tokens('fat.pas',L),wfile(L).
l2:- file_to_tokens('fat.pas',L),zipper_dl(L,Lo/[]),length(Lo,M),writeq(M:Lo).
l3:- file_to_tokens('fat.pas',L),remN(L,Lo), length(Lo,M),writeq(M:Lo).
%% PALAVRAS RESERVADAS
res(prog).   res(program).  res(var). res(const). res(begin).  res(end).
res(write).  res(read). res(nl).
res(if).     res(then). res(else).
res(while).  res(do).
%%
remN([(N,L)|Ls],Lo):-remN(Ls,Lz),append(L,Lz,Lo).
remN([],[]).
zipper_dl([(N,L)|Ls],Wi/Wo):-zipper_one_dl(N,L,Wi/W2),zipper_dl(Ls,W2/Wo).
zipper_dl([],Wo/Wo).
zipper_one_dl(N,[L|Ls],Wi/Wo):-Wi=[(N,L)|W1],zipper_one_dl(N,Ls,W1/Wo).
zipper_one_dl(N,[],W/W).
wfile([X|Xs]):-writeq(X),nl,wfile(Xs). wfile([]).
file_to_tokens(F,L) :-
      open(F,read,_,[type(text),encoding(iso_latin_1),alias(file03)]),
      set_input(file03), procFileN(0,L), close(file03).
procFileN(N,[]):-at_end_of_stream,!.
procFileN(N,[(N1,T)|Ls]):-!,N1 is N+1,read_line_to_codes(file03,Line),
                           readln_(Line,T,_), procFileN(N1, Ls).
readln_(LN, Read, LastCh):-
    rl_readln_(LN, Read, LastCh, [10], "_0123456789", uppercase).
readln_(LN, P, EOF, StopChars, WordChars, Case) :-
    (   var(StopChars)-> Arg1 = [10];           Arg1 = StopChars),
    (   var(WordChars)-> Arg2 = "01234567890_"; Arg2 = WordChars ),
    (   var(Case )    -> Arg3 = lowercase;      Arg3 = Case ),
    rl_readln_(LN, P, EOF, Arg1, Arg2, Arg3).
rl_readln_(LN, P, EOF, StopChars, WordChars, Case) :-
       rl_blanks(LN, LL),
       rl_words(P, options(WordChars, Case), LL, []), !.
rl_words([W|Ws], Opts) --> rl_word(W, Opts),!,rl_blanks, rl_words(Ws, Opts).
rl_words([], _) --> rl_blanks, !.
rl_words([], _) --> [].
rl_word(str(W),Opts)--> [0''], rl_pair(C1, 0''), !,
                        rl_pairs(Rest, 0''),([0''],!; []), {name(W, [C1|Rest])}.
rl_word(str(W),Opts)--> [0'"],rl_pair(C1, 0''), !,
                        rl_pairs(Rest, 0''),([0'"],!; []), {name(W, [C1|Rest])}.
%% two chars symbs
rl_word(W, _, [0'<,0'=|S0], S0) :- name(W, [0'<, 0'=]).
rl_word(W, _, [0'<,0'>|S0], S0) :- name(W, [0'<, 0'>]).
rl_word(W, _, [0'>,0'=|S0], S0) :- name(W, [0'>, 0'=]).
rl_word(W, _, [0'*,0'*|S0], S0) :- name(W, [0'*, 0'*]).
rl_word(W, _, [0':,0'=|S0], S0) :- name(W, [0':, 0'=]).
%%
rl_word(num(N), _) --> [0'.], rl_num(N1), !,  rl_nums(Rest,  dot),
                        {name(N,[0'0 , 0'. , N1|Rest])}.
```

```
51 rl_word(num(N), _) --> rl_num(N1),!, rl_nums(Rest, _),
52                        {name(N,[N1|Rest])}.
53 rl_word(Wo, Opts) -->  rl_char(C1, Opts), !, rl_chars(Rest, Opts),
54                        {name(W, [C1|Rest]), (res(W)-> Wo=r(W); Wo=id(W))}.
55 rl_word(P,_) --> [C],  {name(P, [C])}, !.
56 rl_chars([A|As], Opts) -->  rl_char(A, Opts), !, rl_chars(As, Opts).
57 rl_chars([], _)-->[].
58 rl_pairs([A|As], Opts) --> rl_pair(A, Opts), !, rl_pairs(As, Opts).
59 rl_pairs([], _)--> [].
60 rl_pair(C1, Opts)--> [C1],{Opts\=C1}.
61 rl_nums([46,N|As], Dot)--> [46], {var(Dot)}, rl_num(N),!, rl_nums(As, dot).
62 rl_nums([A|As],    Dot)-->                   rl_num(A),!, rl_nums(As, Dot).
63 rl_nums([],          _) --> [].
64 rl_blanks --> rl_blank, !, rl_blanks.
65 rl_blanks --> [].
66 %% Basic Char types
67 rl_char(A, options(WChars, Case),[C|S],S):- rl_lc(C, A, WChars, Case).
68 rl_num(N, [N|R], R) :- code_type(N, digit).
69 rl_blank( [X|R], R) :- code_type(X, space).
70 rl_lc(X, X1, _,Case):- code_type(X, upper),!,rl_fix_case(Case,X,X1).
71 rl_lc(X, X, _, _) :-   code_type(X, lower).
72 rl_lc(X, X, WordChars, _) :- memberchk(X, WordChars).
73 rl_fix_case(lowercase,U,L):- !,code_type(L,lower(U)).
74 rl_fix_case(_, C, C).
```

O predicado principal é o `readln_`. Seguem dois testes para ele. No primeiro teste é passada a lista de caracteres que podem ocorrer dentro dos identificadores; a opção `lowercase` força tudo em minúscula; strings não são afetados. No segundo teste, os identificadores são mantidos case sensitive (como aparecem no texto).

```
?-readln_("IOLE1a 'SS10.1' temp", P, L, [10],"_01213456789",lowercase).
P = [id(iole1a), str('SS10.1'), id(temp)]

?-readln_("IOLE1a 'SS10.1' temp", P, L, _,_, case).
P = [id('IOLE1a'), str('SS10.1'), id(temp)]
```

O predicado `readln` chama um predicado `rl_words` que lê as palavras. Palavras são separadas por brancos `rl_blanks`. Uma palavra pode ser `id()`, `str()`, `num()` ou um símbolo com um ou mais caracteres <, <=.

Os predicados para ler números e identificadores já foram comentados no capítulo sobre tokens para expressões aritméticas. Aqui usamos o predicado `code_type` para testar o tipo de cada caractere: por exemplo, se `code_type(N, digit)` é verdadeiro então N é um dígito.

Em `rl_word(str(W), Opts)` o parâmetro `Opts` informa se espera aspas ou apóstrofo, no fim do string. E, em `rl_word(num(N), Opts)` o parâmetro `Opts` informa se já foi encontrado o ponto.

Um identificador começa com letra e pode conter todos os caracteres passados no parâmetro em `rl_readln_(... "_0123456789",..)`; neste caso pode conter o underline e os dígitos; assim vale `ID_1A12`.

Segue abaixo diversos testes que podem ser úteis para entender como funcionam certos predicados não detalhados.

```
%% ?- X=[(29, [a,b,c]), (30, [d,e]), (31, [x])],zipper_dl(X,W/[]),length(W,WL).
%% ?- X=[(29, [a,b,c]), (30, [d,e]), (31, [x])],remN(X,W),length(W,WL).
%% ?- readln_("IOle 'SS10.1' Temp", P, L, [10],"_01213456789",uppercase).
%% ?- readln_("Iole 'SS10.1' Temp", P, L, _,_,_).
%% ?- atom_codes('E12+ .2 2.35,"I"',X), rl_words(P, options("12", lowercase),X,Y).
%% ?- atom_codes('E12<=<ca,\'io\'\t,"I"',X), readln_(\t,X,R).
%% ?- rl_words(Ws,[0'a,0'b,32,32,0'c],X,[]).
%% ?- rl_chars(SS, Opts("eee12_", upper)), "a1a_bb 'bb",X).
%% ?- rl_pairs(SS, 0'', "aa'bb",X).
%% ?- rl_nums(N,D,"234.45",X).
%% ?- rl_blanks("   aa",X).
%% ?- rl_fix_case(lowercase, 0'A,X). % A=65, a=97
```

Este programa não gera erros. Em todas as situações são gerados tokens, nem que seja para apenas um caractere. Possíveis erros devem ser diagnosticados no componente sintático.

15.3 O analisador de expressões baseado em tabela

No capítulo sobre gramáticas falamos do problema da precedência e da associatividade, que não é simples resolvê-lo. Este analisador de precedência de operadores resolve este problema fazendo a correta parentização das expressões reconhecidas.

Na verdade este código é do analisador de precedência de operadores embutido na linguagem Prolog. Ele bem flexível e poderoso, permitindo a sobrecarga de operadores: isto é, um operador por exemplo o (-) pode ser prefixo e também infixo: -1 ou 1-2.

Os operadores são declarados como se faz no Prolog, aqui com o predicado ope/3 ou com o predicado mkop/3 para declarar vários ao mesmo tempo. Para entender como eles funcionam sugerimos a leitura da seção sobre operadores do capítulo sobre a sintaxe do Prolog, consultando também a tabela de operadores do Prolog.

O reconhecedor basicamente tem duas partes: uma que define a tabela de operadores e a outra que é o analisador em si (os predicados term e rTerm). Tirando a parte da tabela de operadores o analisador possui apenas uma dezena de linhas de código.

```
%% file: expr-pl.pl
:-['readln-elf'].
%% para entender sobre a posição do ERRO SINTATICO ou
%% para fazer um trace de cada token consumido:  tirar o comentário...
xx(X)-->[(P,X)],!. %% , { write(('@token@',P,X)),nl}.
xx(X)-->[X]. %%, {write(('@token@',X)),nl}.
okDecl(ID):- tab(_,ID,V),!;errNDECL(ID).
%% Parser
%% term(in/out) rTerm(in,in,in/out)
term(PP,To) --> xx(id(X)),  %% {okDecl(X)},
                            rTerm(PP,0,id(X)/To).
term(PP,To) --> xx(num(X)), rTerm(PP,0,num(X)/To).
term(PP,To) --> xx('('), term(1200,T), xx(')'),rTerm(PP,0,(T)/To).
term(PP,To) --> xx(OP), {prefix(P, FX,OP,PR),P=<PP},
                        term(PR,T), {NT=..[OP,T]},rTerm(PP,P,NT/To).
rTerm(PP,Pi,Ti/To) --> xx(OP),{infix(P,YFX,OP,PL,PR),P=<PP,Pi=<PL},
```

```
                          term(PR,T), {NT=..[OP,Ti,T]}, rTerm(PP,P,NT/To).
rTerm(PP,Pi,Ti/To) --> xx(OP),{posfix(P,XF, OP,PL  ),P=<PP,Pi=<PL,
                                    NT=..[OP,Ti]},rTerm(PP,P,NT/To).
rTerm(_ , _,Ti/Ti) --> [].
%% ---TABELA DE OPERADORES-----------------------------------------------
:-abolish(ope/3), dynamic(ope/3).
mkop(P,XFY,[X|Xs]):-assert(ope(P,XFY,X)),mkop(P,XFY,Xs). mkop(_,_,[]).
%% ope(1200,xfx, ':=').
ope(1100,xfy, 'or').
ope(1000,xfy, 'and').
ope(900, fy , 'not').
:- mkop(700, xfx, [ =, <>, >, <, <=, >=, \= ]).
:- mkop(500, yfx, [ +, -, xor ]).
:- mkop(400, yfx, [ *,  /, rem, div, mod ]).
:- mkop(200, xfx, [ ( ** ) ]).
:- mkop(200,  fy, [ +, - ]).
lope:-listing(ope).
prefix(P, fx,OP, P-1  ):- ope(P,fx,OP).
prefix(P, fy,OP, P    ):- ope(P,fy,OP).
infix(P, xfx,OP,P-1,P-1):-ope(P,xfx,OP).
infix(P,xfy,OP,P-1, P):- ope(P,xfy,OP).
infix(P, yfx,OP,P,P-1 ):- ope(P,yfx,OP).
posfix(P, xf,OP, P-1  ):- ope(P,xf,OP).
posfix(P,yf,OP, P    ):- ope(P,yf,OP).
%% ----TESTA TUDO ------------------------------------------------------
% ?- tb(1).
% ?- term(1200,T,[num(1),-,num(1),-,num(1)],P).
% ?- term(1200,T,[id(a),=,num(4),or,id(a), and, id(b), <> ,id(t)],P),display(T).
tb(S):-ss(S,Xs),readln_(Xs,L,_),parse(L),nl.
parse(L):-time(term(1200,T,L,[])),!,write((L,'\n','@OK@:')),display(T),nl.
parse(L):-time(term(1200,T,L,R)),!, write((L,'\n','@ERRO@':R,'\n')),display(T),nl.
parse(L):-write(('@ERRO TUDO@':L)),nl.
tt:-tb(X),fail. tt.  %% testa tudo
ww:-write(termht0),nl,tell('saida-termht0.pl'),tt,told.
s(X,Z):-s0(X,Y),atom_chars(Y,Z),atom_length(Y,L),
         nl,write(teste:X),write(':  '),write(Y),write('@length@'),write(L),nl.
ss(X,Z):-s0(X,Y),atom_codes(Y,Z).
s0(1,'1--1').
s0(2,'1---1*1').
s0(3,'1+a+a--1').
s0(5,'1!!*+(+(+(1)))').
s0(8,'a=b=1').
s0(12,'1@2+1').
s0(13,'+1-').
```

Para testá-lo tem várias perguntas no final código. Por exemplo, `1--1` exemplifica a sobre carga de operadores, parentizado como, `-(1-(1))`

O comentário `%{okDecl(X)}` deve ser tirado para testar se uma variável foi declarada ou não; mas, para isso, deve ser testado juntamente com o tradutor. Se for removido `%%` no predicado `xx` o analisador vai mostrando passo a passo os tokens ele vai reconhecendo:

```
term(PP,To) --> xx(id(X)),  %% {okDecl(X)},  ...
xx(X)-->[(P,X)],!.  %% , { write(('@token@',P,X)),nl}.
xx(X)-->[X].        %% , { write(('@token@',X)),nl}.
```

Segue a execução de alguns exemplos. O terceiro teste mostra um erro pois o (!) não foi definido como operador.

```
?-tt.
 [num(1), -, -, num(1)],
, @OK@:-(num(1), -(num(1)))

 [num(1), +, num(1), *, id(a), *, id(a), +, id(a), -, num(1)],
, @OK@:-(+(+(num(1), *(*(num(1), id(a)), id(a))), id(a)), num(1))

 [num(1), !, !, *, +, (, +, (, +, (, num(1), ), ), )],
, @ERRO EM@ :[!, !, *, +, (, +, (, +, (, num(1), ), ), )],
num(1)
```

15.4 O tradutor: analise sintática e geração de código

O programa tradutor tem três principais partes: a tabela de símbolos, a gramática em si e um conjunto de testes.

O programa inicia com a tabela de símbolos, que pode ter dois tipos de informação: variáveis e constantes. Abaixo, ilustramos um exemplo de tabela com duas variáveis e uma constante.

```
?- wTab.
   tab(var, fat, 0)
   tab(var, valor, 0)
   tab(const, zero, 0)
```

O predicado `initTab` inicializa a tabela como vazia, removendo possíveis entradas e declarando o predicado como dinâmico para poder ser alterado durante a execução. O predicado `mk/3` é usado para incluir uma nova entrada na tabela. Já o predicado `wTab` é usado para escrever a tabela com propósito de teste e depuração.

Os erros semânticos são tratados com as informações da tabela de símbolos. São tratados com dois tipos de testes: (1) na parte de declarações, para verificar se uma variável (ou constante) não está sendo declarada duas vezes; e (2) no corpo do programa, para verificar se uma variável foi declarada. Gera-se duas mensagens de erros semânticos: *variável não declarada* e *variável já declarada*.

```
%% file: tradutor.pl
:-['expr-pl'].
%%-----------TABELA DE SIMBOLOS--------------------
:-dynamic(tab/3).
initTab :- abolish(tab/3),dynamic(tab/3).
mk(C,X,V) :-tab(C,X,V),!;assert(tab(C,X,V)).
wTab :-tab(CV,ID,N), nl, write(tab(CV,ID,N)), fail.  wTab.
errDECL(ID) :-tab(VC,ID,_),!, write('ERRO: Ja Declarada ':VC:ID),nl.
errNDECL(ID):- \+tab(_,ID,_),!, write('ERRO: Nao declarada':ID),nl.
```

```
%%-------GRAMATICA----------------------------------
const  --> xx(r(const)),!,rConst,xx(;).
const  --> [].
rConst --> cDecl,rrConst.
rrConst--> xx(;),cDecl,rrConst.
rrConst--> [].
cDecl  --> xx(id(ID)),xx(=),xx(num(N)),
              {errDECL(ID),!;mk(const,ID,N)} .
  var --> xx(r(var)),!,rVar,xx(';').
  var --> [].
 rVar -->       xx(id(ID)),{errDECL(ID),!;mk(var,ID,0)},rrVar.
rrVar --> xx(','),xx(id(ID)),{errDECL(ID),!;mk(var,ID,0)},rrVar.
rrVar --> [].
prog(prog(X,B))    --> xx(r(program)),!,xx(id(X)),xx(;),block1(B),xx(.).
block1(B)   --> const, var, beStmt(B).
beStmt(be(S))  --> xx(r(begin)),!, stmtSeq(S), xx(r(end)).
stmtSeq([S|Ss]) --> stmt(S), rStmt(Ss).
stmtSeq([]) --> [].
rStmt([S|Ss])   --> xx(;),stmt(S), rStmt(Ss).
rStmt([])   --> xx(;).
rStmt([])   --> [].
stmt(S)  --> beStmt(S).
stmt(attr(ID,E))  --> xx(id(ID)),xx(:=),{tab(var,ID,V);errNDECL(ID)},exp(E).
stmt(if(C,T))  --> xx(r(if)),!, cond(C),xx(r(then)),stmt(T).
stmt(wh(C,S))  --> xx(r(while)),!,cond(C), xx(r(do)),stmt(S).
stmt(read(S))  --> xx(r(read )),!,xx('('),  idSeq(S), xx(')').
stmt(write(S)) --> xx(r(write)),!,xx('('), wrtSeq(S), xx(')').
stmt(nl)       --> xx(r(nl)),!.
 idSeq([ID|Ss]) --> xx(id(ID)),{tab(var,ID,V);errNDECL(ID)}, rIdSeq(Ss).
rIdSeq(S) --> xx(','),idSeq(S).
rIdSeq([]) --> [].
 wrtSeq([E|Es]) --> exp(E),rWrtSeq(Es).
 wrtSeq([str(ID)|Ss]) --> xx(str(ID)),rWrtSeq(Ss).
rWrtSeq(S) --> xx(','),wrtSeq(S).
rWrtSeq([]) --> [].

exp(T) --> term(1201,T). %%,{nl,display(T),nl}.
cond(T) --> term(1201,T). %%,{nl,display(T),nl}.
```

A gramática é parte central do tradutor. Como já falamos, uma gramática DCG é uma gramática do tipo LL(K), que funciona por retrocesso.

15.4.1 Gramática da minilinguagem

Um programa inicia pela declaração de constantes e variáveis. Num programa, podemos ter uma lista de constantes, por exemplo, `const zero=0; mil=1000;`. Cada declaração de constante, regra `const`, termina por um ponto-e-vírgula. Acrescentamos a regra, `const-->[]`, pois a declaração de constantes é opcional. A regra `rCont` define a lista de declarações de constantes. Em cada declaração de constantes, temos uma ação semântica que verifica se a variável sendo declarada já está na tabela, em caso afirmativo é erro, caso contrário, o predicado `mk/3` inclui uma

entrada na tabela.

Variáveis também são opcionais. Por exemplo, |`program vazio; begin end`| é um programa válido sem declaração de variáveis nem de constantes.

Um programa na minilinguagem é formado pela declaração de constantes, seguida da declaração de variáveis e do bloco de comandos. O bloco de comandos começa com a palavra reservada `begin` e termina com `end`. O conteúdo do bloco é uma lista de comandos, podendo conter outros comandos do tipo `begin end`. Além do comando composto são implementados outros cinco comandos: o de atribuição, `:=`; dois condicionais, `if` e `while`; o de leitura, `read`, e o de escrita, `write`.

É difícil desenvolver uma gramática do tipo LL(k) para expressões aritméticas e expressões booleanas que trate dos problemas de precedência e associatividade. Por isso, foi criado o componente já descrito onde as expressões são especificadas por um conjunto de operadores definidos numa tabela. Este componente é chamado em duas produções:

```
exp(T) --> term(1201,T) . %% ,{nl,display(T),nl}.
cond(T) --> term(1201,T). %% ,{nl,display(T),nl}.
```

15.4.2 Testes para o tradutor

Segue abaixo o texto que faz parte do programa tradutor, mas que contém apenas os testes. Para programas complexos, como um compilador (tradutor), é necessário definir estratégias para fazer testes sistemáticos de todas as produções da linguagem. Uma possibilidade é testar pequenos grupos de produções que executam determinadas *funcionalidades*, por exemplo, a declaração de variáveis.

Aqui, os predicados `do/2` e `do2/2` foram definidos para testar produções isoladamente. Com eles se definem testes específicos para declaração de constantes `c1` e variáveis `v1`, `v2`. Por exemplo, o teste `v1` declara três variáveis (que são incluídas na tabela de símbolos) e o teste `v2` tenta declarar duas vezes a mesma variável. Segue a execução do teste `v2`.

```
?- v2.
  [r(var),id(valor),(,),id(valor),(;)] : >>>
  ERRO: Ja Declarada  : var : valor   sobrou : []
  tab(var,valor,0)
```

De forma similar também existem testes para as expressões aritméticas `e2`, `e3` e condicionais `co1`, `co2`.

É importante deixar registrados os testes a serem executados, pois quando modificamos uma parte da gramática rapidamente podemos testar apenas aquele fragmento. Além disso, fica registrado como documentação para lembrarmos depois como se faz o teste.

Temos também testes mais completos, baseados na fita do programa fatorial, codificada em `teste/2`, que são `p`, `pcv`, `terro` e `tfat`. Por fim, temos os testes com os arquivos de entrada `p1` e `t1`.

```
1 do(P,X):-initTab, DO=..[P,X,R],call(DO),write('sobrou': R),nl,wTab.
2 do2(P,X):-initTab, DO=..[P,T,X,R],call(DO),write(T:'\nsobrou': R),nl,wTab.
3 c1:-do(const,[r(const),id(zero),=,num(0),;,;,id(tres2),=,num(3),;]).
4 v1:-do(var,[r(var), id(valor),',',id(maior),',',id(tot),;]).
5 v2:-do(var,[r(var), id(valor),',',id(valor),;]).
6 e1:-assert(tab(var,tot,_)),do2(exp,[num(1), <> ,id(tot)]).
7 e2:-assert(tab(var,tot,_)),do2(exp,[id(l1), <> ,id(tot)]).
```

```
e3:-assert(tab(var,valor,_)),do2(exp,[num(1),*,id(valor),-,num(3),*,num(4)]).
co1:-assert(tab(var,valor,_)),do2(cond,[id(valor), <> , id(zero)]).
co2:-assert(tab(var,valor,_)),do2(cond,[id(valor), <> , num(1), *, id(abc)]).
%-----------------------------------------------------------------
% fitas de um programa
p:-prog(T,[r(program),id(test),;,r(begin), r(end),'.'],X),write(T:X),nl.
pcv:-initTab, prog(T,[r(program),id(test),;,
           r(const),id(zero),=,num(0),;,
           r(var), id(valor),',',id(maior),',',id(total),;,
           r(begin), r(end),'.'],X),write(T:X),wTab,nl.
teste(erro, [r(program),;, r(var), id(valor),',',id(fat),;,r(end),'.']).
teste(fat,
[r(program),id(fatorial),;,
r(var), id(valor),',',id(fat),;,
 r(begin),
    r(write),'(',str('Calcula o fatorial'),')',;,r(nl),;,
    r(write),'(',str(' valor='),')',;,
    r(read),'(',id(valor),')',;,
    id(fat),:=,num(1),;,
    r(while),id(valor), > ,num(0),r(do),
      r(begin),
        id(fat),:=,id(fat),*,id(valor),;,
        id(valor),:=,id(valor),-,num(1),;,
    r(end),;,
    r(write),'(',str('o fatorial eh:'),',',id(fat), ')',;,r(nl),;,
    r(write),'(',str('FIM'),')',;,
    r(end),'.']).
stdout(F,CALL):-tell(F),call(CALL),told.
tfat   :- teste(fat, P),initTab,prog(Po,P,R),nl,writeq(Po:R), nl.
terro  :- teste(erro,P),initTab,prog(Po,P,R),nl,writeq(Po),nl, writeq(er:R).
%% usinf files
%% ?- t1. ?- p1.
t1:- callComp('fat.pas').
p1:- callComp('prog.pas').
callComp(F):- file_to_tokens(F,L), wfile(L),
     zipper_dl(L,Lo/[]),!,         writeq(M:Lo),
     Lo=P,initTab,prog(Po,P,R),!, nl,writeq(Po),nl,writeq(sobrou:R).
```

Comparando-se o tamanho da gramática com o tamanho do arquivo de testes, vemos que são do mesmo tamanho. Isto mostra que devemos nos preocupar com os testes, principalmente em programas complexos como um compilador.

15.5 O interpretador: análise semântica

O interpretador roda sobre o código gerado pelo tradutor, logo, ele inicia carregando o tradutor. O tradutor é ativado pelo predicado `prog/3` que retorna a árvore sintática para ser interpretada. A entrada do tradutor é à saída do léxico, que é ativado pelo predicado `file_to_tokens`.

Temos também, abaixo, um teste que não depende destes dois componentes, mas apenas de uma árvore sintática codificada como o fato `testeInt/2`. Isto é bom, pois, às vezes não queremos

depender dos outros componentes para trabalhar com o interpretador. Segue um exemplo de execução do programa fatorial.

```
?- tfati.
executando..fatorial
Calcula o fatorial
 valor=entre com o valor:valor
|    5. <enter>
valor=0 novo@@ valor=5
  fat=0 novo@@ fat=1
  fat=1 novo@@ fat=5
valor=5 novo@@ valor=4
  fat=5 novo@@ fat=20
valor=4 novo@@ valor=3
 fat=20 novo@@ fat=60
valor=3 novo@@ valor=2
 fat=60 novo@@ fat=120
valor=2 novo@@ valor=1
fat=120 novo@@ fat=120
valor=1 novo@@ valor=0
o fatorial eh:120
FIM
```

O interpretador é codificado como uma função recursiva com um predicado para cada tipo de construção do programa. Temos um predicado `aval` que chama `avalif`, `avalwh`, `avalwrL` para respectivamente os comandos if, while e write. Depois temos o predicado `avalExp` que tem uma regra (linha) para cada operador, se chama recursivamente para cada operando. A recursividade termina quando é encontrado um identificador ou uma constante. O valor de um identificador é buscado na tabela de símbolos.

```
1  %% file:inter.pl
2  :-['tradutor'].
3  callCompInt(F):- file_to_tokens(F,L),   % wfile(L),
4       zipper_dl(L,Lo/[]),length(Lo,M),writeq(M:Lo),!,Lo=P,
5       initTab,prog(Po,P,R),    % nl,writeq(Po),nl,writeq(Po),
6       !,aval(Po).
7  t1i:- callCompInt('fat.pas').
8  p1i:- callCompInt('prog.pas').
9  %% EXEMPLO DE ÁRVORE SINTATICA ABSTRATA DO FAT
10 tint   :- initTab, testeInt(fat,P),aval(P).
11 testeInt(fat,
12     prog(fatorial, be([write([str('Calcula o fatorial')]), nl,
13     write([str(' valor=')]), attr(valor, num(5)),
14     attr(fat, num(1)),
15     wh(id(valor)>num(0),
16     be([attr(fat, id(fat)*id(valor)), attr(valor, id(valor)-num(1))])),
17     write([str('o fatorial eh:'), id(fat)]), nl, write([str('FIM')])])) ).
18 %% testa os exemplos do tradutor
19 tprogi :- teste(prog,P),initTab,prog(Po,P,[]),nl,writeq(Po),nl, !,aval(Po).
20 tfati  :- teste(fat, P),initTab,prog(Po,P,[]),nl,writeq(Po),nl,!,aval(Po).
21 %% ?- avalExp( id(valor)>id(maior)+id(total),X).
```

```
%%-------------------------------------------------------------
aval([]).
aval(prog(ID,B)):-nl,write('executando..'),write(ID),nl,aval(B).
aval(be(X)):-!,aval(X).
aval(if(C,B)):-!,avalif(C,B).
avalif(C,B):-avalExp(C,Co),Co=true,!,aval(B).
avalif(C,B).
aval(wh(C,B)):-!,avalwh(C,B).
avalwh(C,B):-avalExp(C,Co),Co=true,!,aval(B),avalwh(C,B).
avalwh(C,B).
aval([X|Xs]):-aval(X),aval(Xs).
aval(read([X])):- write('entre com o valor':X),nl,!,
                  read(Xo),!,save(tab(_,X,Xo)).
aval(write(X)):-!,avalwrL(X).
avalwrL([X|Xs]):-!,avalExp(X,Xo),write(Xo),avalwrL(Xs).
avalwrL([]).
aval(nl):-nl.
aval(attr(X,E)):-avalExp(E,Eo),save(tab(_,X,Eo)).
avalExp(str(X),X).
avalExp(id(X),Y):-tab(_,X,Y).
avalExp(num(X),X).
avalExp(<(X,Y),V):-avalExp(X,Xo),avalExp(Y,Yo),!, ((Xo>Yo, V=true,!;V=false)).
avalExp(>(X,Y),V):-avalExp(X,Xo),avalExp(Y,Yo),!, ((Xo>Yo, V=true,!;V=false)).
avalExp( <>(X,Y),V):-avalExp(X,Xo),avalExp(Y,Yo),!, ((Xo\=Yo, V=true,!;V=false)).
avalExp(+(X,Y),V):-avalExp(X,Xo),avalExp(Y,Yo), (V is Xo+Yo).
avalExp(-(X,Y),V):-avalExp(X,Xo),avalExp(Y,Yo), (V is Xo-Yo).
avalExp(*(X,Y),V):-avalExp(X,Xo),avalExp(Y,Yo), (V is Xo*Yo).
avalExp(/(X,Y),V):-avalExp(X,Xo),avalExp(Y,Yo), (V is Xo/Yo).
save(tab(X,Y,Z)):-tab(X,Y,Vi), asserta(tab(X,Y,Z)),
                          write(Y=Vi),write(' novo@@ '),write(Y=Z),nl,!.
save(tab(X,Y,Z)):-asserta(tab(X,Y,Z)),write(' novo@@@ '),write(Y=Z),nl,!.
```

Como foi visto, em Prolog, podemos codificar um sistema complexo como um compilador em componentes modulares, que podem ser testados isoladamente. Cada componente tem menos de uma centena de linhas de código fonte, assim fica fácil dominá-lo mentalmente.

Aqui termina nosso estudo de caso sobre compiladores. Restam vários desafios para os alunos. Por um lado, como estender o léxico para trabalhar com números de ponto flutuante e outros tipos de inteiros (binários e hexa) como os da linguagem Java; como incluir declarações variáveis destes diferentes tipos: bool, char, etc; como incluir outros comandos: if-then-else, for, etc; como incluir vetores; como incluir funções tipo Java; etc; Por outro lado, como gerar assembler para a árvore sintática, apenas substituindo o interpretador.

Bom estudo e bom trabalho.

CAPÍTULO 16

Processamento de Linguagem Natural com DCG

Neste capítulo apresentamos alguns conceitos de Processamento de Linguagem Natural (PLN) fazendo uso da notação gramatical DCG do Prolog.

O material deste capítulo pode ser aprofundado em livros, tais como, M. A. Covington, [8] *Natural Language Processing for Prolog Programmers*, Prentice Hall, New Jersey, 1994.

16.1 Introdução ao PLN

Processamento de Linguagem Natural é uma subárea da Inteligência Artificial que estuda o uso do computador para processar linguagens usadas na comunicação entre seres humanos, tais como português e inglês.

Exemplos de aplicações em PLN incluem: *Tradução automática*; *Extração automática de conhecimento* a partir de textos; *Reconhecimento de voz*; *Dicionários* eletrônicos e *Corretores* ortográficos; etc. Estas aplicações, no estado da arte atual, ainda deixam muito a desejar, por exemplo, para um tradutor, a qualidade da tradução ainda é bem pobre.

Do ponto de vista da Lingüística (ciência que estuda as linguagens naturais), a estrutura de uma linguagem natural é classificada em cinco níveis: Fonologia (som); Morfologia (formação das palavras); Sintaxe (estrutura das frases); Semântica (significado das frases) e Pragmática (uso da linguagem num contexto).

Em uma perspectiva ortogonal à Lingüística, o estudo de linguagens em computação faz referência à área da computação conhecida como linguagens formais (de computador). Uma linguagem formal é descrita e processada em três níveis de conhecimento: *Léxico* (ou morfológico) trata dos tokens e palavras; *Sintático* trata das sentenças (erros de gramática); *Semântico* trata do significado das sentenças ou tradução.

Neste capítulo, apresentamos noções de PLN, enfocando as áreas de morfologia, sintaxe e semântica. A tecnologia desenvolvida para linguagens formais de computador é usada em PLN, principalmente em aplicações sobre linguagem escrita.

16.2 Léxico – Morfologia

A morfologia estuda a formação das palavras. Uma linguagem tem dois principais tipos de formação de palavras: (1) Inflexão: dada a raiz de uma palavra geram-se várias formas, e.g., mesa, mesas; canto, canta, cantas, cantem; (2) Derivação: juntar palavras existentes para formar outras, e.g., guarda-chuva.

Palavras são formadas por caracteres. O Prolog possui um conjunto de facilidades para se trabalhar com caracteres formando palavras. Um símbolo atômico, ou palavra básica do Prolog, pode ser decomposto em uma lista de códigos Ascii. Cada código Ascii representa um caractere. A maioria dos caracteres é visível na tela e possui um significado simbólico. Existem também caracteres de controle, por exemplo, caracteres de nova linha e de tabulação.

O Prolog possui o predicado `name(ATOMO, LISTA_CHAR)` que converte um átomo numa lista de códigos Ascii, onde cada caractere é representado por um código. Seguem dois exemplos:

```
?- name('ABC',X).
   X = [65,66,67]
?- name('abc',X).
   X = [97,98,99]
```

No Prolog também podemos escrever a lista dos valores Ascii de um átomo, colocando-o entre aspas. Na primeira pergunta, abaixo, vemos que os caracteres "AZaz" valem [65,90,97,122], o que significa que o conjunto das letras maiúsculas("AZ") está no intervalo entre 65 e 90, o conjunto das minúsculas ("az") está no intervalo entre 97 e 122, e que temos 26 letras, incluindo o Y e W. Na segunda pergunta, temos os valores para os dígitos decimais que estão entre 48 e 57. Na terceira pergunta, temos uma lista de símbolos gráficos normalmente usados no Prolog para programação. O espaço em branco é o valor 32.

```
?- "AZaz"=X.
   X = [65,90,97,122]
?- "0123456789" = Y.
   Y = [48,49,50,51,52,53,54,55,56,57]
?- " !@$%$^&*()-+" = Z.
   Z = [32,33,64,36,37,36,94,38,42,40,41,45,43]
```

16.2.1 Flexão em gênero

Um tipo de processamento morfológico (que manipula palavras) é a flexão em gênero: *dada a forma no masculino, gerar a forma correspondente no feminino*. Trabalha-se no nível dos caracteres, por exemplo, na palavra `menino`, como substituir o "o" por um "a" para gerar a palavra `menina`. Decompomos a palavra numa lista de caracteres, usando `name/2`, trocamos o último caractere por um "a", como segue:

```
?- Z=menina, name(Z,NZ),append(X,[97],NZ),
                        append(X,[111],NZ1), name(Z1,NZ1).
Z = menina ,
NZ = [109,101,110,105,110,97] ,
X = [109,101,110,105,110] ,
NZ1 = [109,101,110,105,110,111] ,
Z1 = menino
```

O valor Ascii do "o"é 111 e do "a"é 97. Para usar o predicado `name`, temos que codificar os valores Ascii dos caracteres nas regras. O append tem dois usos, um para separar o "o"da palavra menino ("menin"+"o") e outro para juntar o "a"formando menina ("menin"+"a"). Para evitar a manipulação de valores Ascii, que são um tanto obscuros, fazemos dois predicados: `separa` e `junta`. Estes predicados, escondendo o trabalho com os valores Ascii, permitem átomos nos seus parâmetros, como segue.

```
junta(X,Y,Z):-nonvar(X),nonvar(Y),name(X,NX),name(Y,NY),
              append(NX,NY,NZ),name(Z,NZ),!.
separa(X,Y,Z):-nonvar(Z),name(Z,NZ),
               append(NX,NY,NZ),name(Y,NY),name(X,NX),!.
genero(LEMA,LEX):- var(LEX ),separa(RAIZ,o,LEMA),junta(RAIZ,a,LEX ).
genero(LEMA,LEX):- var(LEMA),separa(RAIZ,a,LEX ),junta(RAIZ,o,LEMA).
```

O `junta` exige os dois primeiros parâmetros instanciados; o `separa` exige o último parâmetro instanciado. A partir dos predicados `junta` e `separa` é codificada uma regra que faz a conversão entre `genero`: se damos a forma feminina ela devolve a masculina e vice-versa. Seguem alguns exemplos:

```
?- separa(M,o,menino).
   M = menin
?-junta(menin,o,M).
   M = menino
?- genero(menino, X).
   X = menina
?- genero(X, menina).
   X = menino
```

16.3 Um exemplo de minilinguagem

Para mostrar como podemos trabalhar com a sintaxe de uma linguagem natural, devemos focalizar a aplicação num domínio bem restrito. Assumimos o domínio de um pequeno ambiente para uma família de cães, gatos e ratos. Neste domínio podemos fazer frases tais como:

- *O gato persegue o Mikey.*
- *As gatas comem o queijo do Mikey.*
- *Ele persegue o gato. Ele come uns ratos.*
- *Bidu persegue Mimi. Bidu dorme.*

Abaixo, apresentamos algumas regras gramaticais para gerar frases como as exemplificadas, com até dois participantes. Uma oração (ou sentença) é formada por uma frase nominal (`fraseNom`)[1] seguida de uma frase verbal (`fraseVerbal`). Uma frase nominal é um nome opcionalmente precedido por um artigo ou um pronome. Uma frase verbal é um verbo seguido de uma frase nominal.

[1] É comum em Português usar o termo sintagma nominal. Nós preferimos usar o termo frase em vez de sintagma. Frase é uma seqüência de palavras que forma um sentido completo (por exemplo: exclamação, oração simples, oração complexa). Sintagma é uma seqüência de palavras que forma um constituinte. Uma oração também é um constituinte gramatical.

```
sent        --> fraseNom, fraseVerbal.
fraseNom    --> artigo, nome.
fraseNom    --> nome.
fraseNom    --> pronome.
fraseVerbal --> verbo, fraseNom.
fraseVerbal --> verbo.
%%
nome   --> [bidu] ; [mimi] ; [mikey].
nome   --> [cao] ; [gato] ; [gata] ; [gatas] ; [rato] ; [queijo].
pronome--> [ele] ; [ela].
verbo  --> [late] ; [persegue] ; [come]; [comem]; [dorme].
artigo --> [A], {member(A,[a,as,o,os,um,uns,uma,umas])}.
```

A segunda parte do código da gramática define um *léxico* para o domínio do problema. Neste léxico, todas as palavras usadas nas sentenças devem ser classificadas em categorias léxicas, que incluem nomes próprios (Bidu, Mikey), nomes comuns (gato, rato, queijo), pronomes (ele), verbos (persegue, dorme) e artigos (o, os).

Para testar as regras, criamos uma base de testes e um predicado que chama cada um dos testes (`run`).

```
teste([o, gato, dorme]).
teste([o, gato,persegue,mikey]).
teste([o, gato,come,o,queijo,do,mikey]).
teste([a, gata,come,o,queijo]).
teste([o, gatas,come,umas,queijo]).
teste([ela, persegue, o, rato]).
teste([ele, persegue, um, mikey]).
teste([ele, persegue, o, mikey]).
teste([eles, come, um, rato]).
teste([ele, come, uns, rato]).
teste([o,ele,come,o,rato]).
%%
wOK(R,T):-R=[]-> write((’   ok’:T)); write((erro1:T:sobrou:R)).
wErro(T):-write((erro2:T)).
run(T):-sent(T,R)-> wOK(R,T); wErro(T ).    run.
```

Cada produção codificada como uma regra gramatical DCG, quando executada, retorna um dos três estados:

- falhou completamente, não reconhecendo nada (`erro2`);
- reconheceu uma parte da sentença, retornando, no segundo parâmetro, à parte não reconhecida (`erro1`);
- reconheceu a sentença toda, retornando, no segundo parâmetro, a lista vazia (`ok`).

Estas regras estão codificadas no predicado `run/1`. O predicado `run/0`, por retrocesso, chama o teste para todas as entradas da base de testes.

Segue o resultado dos testes. Podemos ver que a gramática apresenta vários problemas, incluindo falta de tratamento para a concordância nominal e verbal, por exemplo, */*o gatas /*eles come /*um Mikey/*. Estes problemas são analisados a seguir.

```
?- run.
   ok : [o,gato,dorme]
   ok : [o,gato,persegue,mikey]
erro1 : [o,gato,come,o,queijo,do,mikey] : sobrou : [do,mikey]
   ok : [a,gata,come,o,queijo]
   ok : [o,gatas,come,umas,queijo]
   ok : [ela,persegue,o,rato]
   ok : [ele,persegue,um,mikey]
   ok : [ele,persegue,o,mikey]
   ok : [eles,come,um,rato]
   ok : [ele,come,uns,rato]
erro2 : [o,ele,come,o,rato]
yes
```

16.4 Árvore sintática

Alguns conceitos sobre sintaxe estão diretamente relacionados à construção de árvores sintáticas para sentenças. Abaixo, temos uma árvore sintática para a frase *Ele come o queijo*. A raiz da árvore é o nó `sent`; na fronteira da árvore, temos os nós folhas (e.g., `ele`, `come`); internamente na árvore temos os nós `fn` e `fv`, que denotam frase nominal e verbal, respectivamente. Estes nós são também chamados de constituintes gramaticais. Para cada constituinte temos uma frase associada, por exemplo, o `fn` mais interno é associado à frase *o queijo*. Cada palavra é classificada numa das categorias léxicas: pron(ome), verbo, art(igo) e nome comum.

```
       fn -- pron -- ele
      /
 sent      verbo -- come
      \   /
       fv      art -- o
          \   /
           fn
              \
               comum -- queijo

sent(fn(pron(ele)),fv(verbo(come),fn(art(o),comum(queijo))))
```

A árvore possui também uma representação na forma de termo, como é mostrado acima. Esta representação na forma de termo é facilmente gerada a partir das regras DCG da gramática. Cada nó interno da árvore é uma regra DCG. Se o nó tem dois filhos, então no corpo da regra temos uma chamada para duas sub-regras.

Abaixo, apresentamos uma versão da gramática que gera árvores sintáticas. Acrescentamos um sufixo (_) em cada predicado para diferenciá-los da versão anterior. As regras para nome, pronome e verbo, aqui, são codificadas com uso do member, como havíamos codificado a regra-artigo na versão anterior.

```
1 sent_(sent(FN,FV) ) --> fraseNom_(FN), fraseVerbal_(FV).
2    fraseNom_(fn(A,N) ) --> artigo_(A), nome_(N).
3    fraseNom_(fn(N)   ) --> nome_(N).
```

```
4     fraseNom_(fn(P)    ) --> pronome_(P).
5  fraseVerbal_(fv(V,FN)) --> verbo_(V), fraseNom_(FN).
6  fraseVerbal_(fv(V)    ) --> verbo_(V).
7  %%
8  nome_(propr(N))    --> [N], {member(N, [bidu,mimi,mikey])}.
9  nome_(comum(N))    --> [N], {member(N, [cao, gato, rato, queijo])}.
10 pronome_(pron(P)) --> [P], {member(P, [ele,ela])}.
11   verbo_(verbo(V))--> [V], {member(V, [late, persegue, come])}.
12  artigo_( art(A)) --> [A], {member(A, [a,as,o,os,um,uns,uma,umas])}.
```

Podemos testar os diferentes constituintes isoladamente. Por exemplo, perguntar sobre o nome próprio *Mikey*: `?-nome_(N,[mikey],[])`. O resultado é `N=propr(mikey)`. Uma pergunta sobre uma frase nominal é `?- fraseNom_(A, [o, cao],[])`, que resulta em `A=fn(art(o), comum(cao))`.

Uma regra DCG, quando traduzida para uma cláusula, é acrescida de dois parâmetros (os dois mais à direita): (1) lista de entrada e (2) lista de saída. Quando a gramática reconhece todos os tokens da lista de entrada, a lista de saída é vazia. A árvore é retornada no primeiro dos três parâmetros.

Seguem algumas perguntas sobre árvores sintáticas:

```
?- fraseNom_(A, [mikey],[]).
   A = fn(propr(mikey))
?- fraseNom_(A, [o, cao],[]).
   A = fn(art(o),comum(cao))
?- sent_(A, [o, cao, late],[]).
   A = sent(fn(art(o),comum(cao)),fv(verbo(late)))
?- sent_(A, [ele, come, o, queijo],[]).
   A = sent(fn(pron(ele)),fv(verbo(come),fn(art(o),comum(queijo))))
```

Na geração de árvores, fizemos somente testes que são válidos, pois quando uma pergunta falha não retorna valores nas variáveis.

16.5 Incluindo regras para concordância

A concordância pode acontecer em diversos níveis, por exemplo, nas frases */*eles compra livro* /*Mimi perseguir Mikey/* temos problemas de concordância verbal. E nas frases */*o livros* /*a livros/* temos problemas concordância nominal.

Para trabalhar com concordância é necessário estender as regras gramaticais com parâmetros associados aos traços das palavras de cada frase. Traços lingüísticos são informações lingüísticas que caracterizam uma palavra. Seguem alguns traços para as palavras, *cão, ele, Bidu*:

- *cão* – nome comum, gênero masculino e número singular;
- *ele* – pronome pessoal, gênero masculino e número singular, na terceira pessoa;
- *Bidu* – nome próprio (um cão), gênero masculino.

Os traços de uma palavra são representados num fato. Por exemplo, para a palavra *ele* o fato é `tracos(ele, t(pronome, masc/sing, s3))`. Um ou mais valores de traços são passados como parâmetros entre as regras gramaticais para verificar as regras de concordância.

Abaixo, na regra `sent1`, temos em T os valores dos traços da frase nominal (`fraseNom(T)`). Verificamos que estes mesmos valores acontecem na frase verbal (`fraseVerbal(T)`).

Numa frase nominal, devemos garantir (ou estabelecer) a concordância entre gênero e número. A concordância em gênero é estabelecida pela variável MF, resolvendo erros do tipo */*a gato /*o gata/*. Ao mesmo tempo, a concordância em número é estabelecida pela variável SP, resolvendo erros do tipo: */*o gatos /*os gato/*. Além disso, um artigo indefinido deve preceder um nome comum, resolvendo erros do tipo: */*um Mikey/*. Um nome próprio ou um pronome, também, são frases nominais.

Numa frase verbal, temos apenas o verbo ou o verbo seguido de uma frase verbal. Nesta gramática, na frase verbal, não existe concordância entre o verbo e o objeto. Tipicamente, o verbo só concorda com o sujeito.

```
sent1       --> fraseNom(T), fraseVerbal(T).
fraseNom(t(_, MF/SP,P)) --> artigo(t(_,MF/SP,_)), nome(t(comum,MF/SP,P)).
fraseNom(t(_, MF/SP,P)) --> artigo(t(_,MF/SP,def)), nome(t(propr,MF/SP,P)).
fraseNom(t(_, MF/SP,s3)) --> nome(t(C,MF/SP,_)), {C=propr; C=pronome}.
fraseVerbal(T) --> verbo(T), fraseNom(_).
fraseVerbal(T) --> verbo(T).
%%
nome(t(propr,  MF/SP,_)) --> [X],{tracos(X,t(propr,  MF/SP,_)) }.
nome(t(comum,  MF/SP,_)) --> [X],{tracos(X,t(comum,  MF/SP,_)) }.
nome(t(pronome,MF/SP,_)) --> [X],{tracos(X,t(pronome,MF/SP,_)) }.
verbo(t(_,     MF/SP,P)) --> [X],{tracos(X,t(verbo,  MF/SP,P)) }.
artigo(t(_,    MF/SP,P)) --> [X],{tracos(X,t(artigo, MF/SP,P)) }.
%%
do(S,T):- CALL=..[S,T,R], CALL-> wOK(R,T); wErro(T).
do1:-teste(T),do(sent1,T),nl,fail.   do1.
```

Para esta versão da gramática, o léxico é codificado como uma tabela de palavras associadas a traços lingüísticos. Todas as palavras usadas numa frase devem constar no léxico. Para aplicações não-triviais, o léxico é uma fonte de conhecimento grande e dispendiosa, portanto a codificação manual de um léxico é uma atividade tediosa. Seguem os fatos do léxico:

```
tracos0(cao,   t(comum, masc/sing,  _)).
tracos0(gato,   t(comum, masc/sing,  _)).
tracos0(gata,   t(comum, fem/sing,  _)).
tracos0(rato,   t(comum, masc/sing,  _)).
tracos0(queijo,   t(comum, masc/sing,  _)).
%%
tracos0(ele,   t(pronome, masc/sing, s3)).
tracos0(eles,  t(pronome, masc/plur, p3)).
tracos0(ela,   t(pronome, fem/sing, s3)).
%%
tracos0(come,     t(verbo, _/sing, s3)).
tracos0(persegue, t(verbo, _/sing, s3)).
tracos0(comem,    t(verbo, _/plur, p3)).
%%
tracos0(bidu,  t(propr, masc/_, _)).
tracos0(mikey, t(propr, masc/_, _)).
```

```
tracos0(mimi,  t(propr,  fem/_, _)).
%%
tracos0(o,   t(artigo, masc/sing, def)).
tracos0(os,  t(artigo, masc/plur, def)).
tracos0(um,  t(artigo, masc/sing, ind)).
tracos0(uns, t(artigo, masc/plur, ind)).
tracos0(a,   t(artigo,  fem/sing, def)).
tracos0(as,  t(artigo,  fem/plur, def)).
tracos0(uma, t(artigo,  fem/sing, ind)).
tracos0(umas, t(artigo, fem/plur, ind)).
%%
tracos(P, T) :- tracos0(P, T),!.
```

Abaixo, temos os testes executados para a gramática `sent1`, que verifica também as regras de concordância. Foram identificados vários novos erros, os de concordância:
*/*o gatas /*um Mikey /*eles come /*uns rato/.*

```
?- do1.
do1.
   ok : [o,gato,dorme]
   ok : [o,gato,persegue,mikey]
erro1 : [o,gato,come,o,queijo,do,mikey] : sobrou : [do,mikey]
   ok : [a,gata,come,o,queijo]
erro2 : [o,gatas,come,umas,queijo]
   ok : [ela,persegue,o,rato]
erro1 : [ele,persegue,um,mikey] : sobrou : [um,mikey]
   ok : [ele,persegue,o,mikey]
erro2 : [eles,come,um,rato]
erro1 : [ele,come,uns,rato] : sobrou : [uns,rato]
erro2 : [o,ele,come,o,rato]
yes
```

Na prática, esta gramática sintática pode ser vista como uma espécie de corretor gramatical, que verifica erros comparando os traços das palavras que compõem as sentenças.

16.6 Gerando uma árvore sintática

Aqui, vemos como juntar o código que gera uma árvore (`sent_`) ao código que faz a concordância (`sent1`). Basicamente, é só acrescentar um parâmetro em cada não-terminal da gramática `sent1`, como segue.

```
sent2(sent(FN,FV)          )   --> fraseNom(FN,T), fraseVerbal(FV,T).
fraseNom(fn(A,N),t(_,MF/SP,P))-->artigo(A,t(_,MF/SP,_)),nome(N,t(comum,MF/SP,P)).
fraseNom(fn(A,N),t(_,MF/SP,P))-->artigo(A,t(_,MF/SP,def)),nome(N,t(propr,MF/SP,P)).
fraseNom(fn(N), t(_, MF/SP,s3)) --> nome(N,t(C,MF/SP,_)), {C=propr; C=pronome}.
fraseVerbal(fv(V,FN),T) --> verbo(V,T), fraseNom(FN,_).
fraseVerbal(fv(V),T) --> verbo(V,T).
%%
nome(propr(X), t(propr,  MF/SP,_)) --> [X],{tracos(X,t(propr,  MF/SP,_)) }.
```

```
nome(comum(X), t(comum,  MF/SP,_)) --> [X],{tracos(X,t(comum,  MF/SP,_)) }.
nome( pron(X), t(pronome,MF/SP,_)) --> [X],{tracos(X,t(pronome,MF/SP,_)) }.
verbo(verbo(X),t(_,      MF/SP,P)) --> [X],{tracos(X,t(verbo,  MF/SP,P)) }.
artigo(art(X), t(_,      MF/SP,P)) --> [X],{tracos(X,t(artigo, MF/SP,P)) }.
%%
do2(S,T,A):- CALL=..[S,A,T,R], CALL-> wOK(R,T); wErro(T).
do2:-teste(T),do2(sent2,T,A),nl,write((arvor:A)),nl,fail. do2.
```

Abaixo, temos uma lista de testes para a gramática `sent2/1`. Num erro do tipo `erro1`, uma árvore parcial é gerada, porém, num erro do tipo `erro2`, nada é gerado para a variável da árvore sintática.

```
?-do2.
   ok : [o,gato,dorme]
arvor : sent(fn(art(o),comum(gato)),fv(verbo(dorme)))
   ok : [o,gato,persegue,mikey]
arvor : sent(fn(art(o),comum(gato)),fv(verbo(persegue),fn(propr(mikey))))
erro1 : [o,gato,come,o,queijo,do,mikey] : sobrou : [do,mikey]
arvor : sent(fn(art(o),comum(gato)),fv(verbo(come),fn(art(o),comum(queijo))))
   ok : [a,gata,come,o,queijo]
arvor : sent(fn(art(a),comum(gata)),fv(verbo(come),fn(art(o),comum(queijo))))
erro2 : [o,gatas,come,umas,queijo]
arvor : _00074A42
   ok : [ela,persegue,o,rato]
arvor : sent(fn(pron(ela)),fv(verbo(persegue),fn(art(o),comum(rato))))
erro1 : [ele,persegue,um,mikey] : sobrou : [um,mikey]
arvor : sent(fn(pron(ele)),fv(verbo(persegue)))
...
erro2 : [o,ele,come,o,rato]
arvor : _00074A42
yes
```

Exercício 16.1 *Assuma uma frase com dois participantes. Faça uma regra para gerar a forma passiva dada a forma na ativa. Por exemplo, dada a frase* Ele persegue Mimi *deve ser gerada a frase* Mimi é perseguida por ele.

16.7 Semântica para linguagem natural

O nível semântico trabalha com o significado de uma frase. Podemos pensar numa aplicação que usa uma representação lógica (ou próxima de) para frases. Estamos falando de lógica dos predicados (ou de primeira ordem). O vocabulário da lógica dos predicados consiste em predicados, em quantificadores (existencial e universal) e em variáveis (ver capítulo 2). Seguem alguns exemplos de frases, cada uma com uma possível representação lógica:

- *Um cão late.*
  `[existe(x), cao(x), [late(x)]]`
- *Bidu persegue Mikey.*
  `[existe(x), x=bidu, [existe(y), y=mikey, persegue(x, y)]]`

- *Bidu persegue todo gato.*
  `[existe(x), x=bidu, [todo(y), gato(y), persegue(x, y)]]`
- *Todo gato persegue todo rato.*
  `[todo(x), gato(x), [todo(y), rato(y), persegue(x, y)]]`
- *Todo rato come um queijo.*
  `[todo(x), rato(x), [existe(y), queijo(y), come(x, y)]]`
- *Algum cão late.*
  `[existe(x), cao(x), [late(x)]]`

Nesta aplicação, o estudo da semântica de linguagem natural consiste na tradução de frases para fórmulas lógicas. Várias frases podem ter uma mesma representação lógica, por exemplo, as frases */O cão late /Um cão late /Algum cão late/* são traduzidas para `[existe(x), cao(x), [late(x)]]`.

Abaixo, temos uma versão de uma gramática que descreve a tradução de frases para formas lógicas, como exemplificado.

Quando temos um nominal composto por um nome próprio, por exemplo, *Bidu*, geramos `[existe(x), x=bidu, [fv/x]]`. Aqui, `vf/x` significa que temos uma frase verbal incompleta pela falta de um valor `x`. Na representação de uma frase nominal, usamos o esquema `FN/X/V`, onde X é a variável que faz a ligação entre a frase nominal e uma frase verbal (`V`). Dentro da frase nominal o `V` ocupa a última posição da lista. Por exemplo,

`FN/X/V = [existe(X), cao(X), V]/X/V` se temos a frase verbal
`V/X=[late(X)]`, juntando as duas temos
`[existe(X), cao(X), [late(X)]]`.

Um esquema similar de programa também é usado para tratar verbos transitivos, isto é, com dois argumentos.

Na gramática `sent3`, o operador (`=..`) é usado para criar um termo parametrizado com uma variável, por exemplo, `cao(X)=..[cao,X]`.

```
1  sent3(FN) --> fraseNom3(FN/X/FV), fraseVerbal3(FV/X).
2  fraseNom3([existe(X),X=N, V]/X/V ) --> artigo3(A), nome3(proprio(N)).
3  fraseNom3([existe(X), NX, V]/X/V ) --> artigo3(A), nome3(comum(N)),{NX=..[N,X]}.
4  fraseNom3([  todo(X), NX, V]/X/V )--> quant3(todo(X)), nome3(comum(N)),{NX=..[N,X]}.
5  fraseNom3([ existe(X),NX, V]/X/V ) --> quant3(algum(X)), nome3(comum(N)),{NX=..[N,X]}.
6  fraseNom3([existe(X),X=N, V]/X/V ) --> nome3(proprio(N)).
7  fraseVerbal3(FN/X) --> verbo_dois3(V), fraseNom3(FN/Y/VXY), {VXY=..[V,X,Y]}.
8  fraseVerbal3([VX]/X)  --> verbo_um3(V),{VX=..[V,X]}.
9  %%
10 quant3(todo(X))   --> [todo].
11 quant3(todo(X))   --> [qualquer].
12 quant3(algum(X))  --> [algum].
13 nome3(proprio(N)) --> [N], {member(N, [bidu,mimi,mikey])}.
14 nome3(comum(N))   --> [N], {member(N, [cao, gato, rato, queijo])}.
15   verbo_dois3(V)  --> [V], {member(V, [late, persegue, come])}.
16     verbo_um3(V)  --> [V], {member(V, [late])}.
17   artigo3(art(A)) --> [A], {member(A, [a,o,um,uma])}.
```

Segue uma base de testes para a gramática acima. Assumimos, neste exemplo, que as entradas são sintaticamente corretas.

```
1 teste3([um, cao, late]).
2 teste3([bidu, persegue, mikey]).
3 teste3([bidu, persegue, todo, gato]).
4 teste3([todo, gato, persegue, todo, rato]).
5 teste3([todo,rato, come, um, queijo]).
6 teste3([algum, cao, late]).
```

Podemos testar uma sentença isoladamente. Seguem os testes para duas sentenças. Nos sistemas Prolog, é usual mostrar as variáveis com nomes pouco legíveis, como _G476 e _G529.

```
?- sent3(A, [todo,rato, come, um, queijo],[]).
   A = [todo(_G476), rato(_G476), [existe(_G529), queijo(_G529), come(_G476, _G529)]]
Yes
?- sent3(A, [todo,rato, come, um, queijo],[]), limpa(A).
   A = [todo(x), rato(x), [existe(y), queijo(y), come(x, y)]]
Yes
```

Para as formas lógicas, fizemos um predicado `limpa/1`, que substitui cada variável por uma constante, letras minúsculas (`x,y,z...`). Este predicado faz uso do predicado `free_variables`, que retorna todas as variáveis livres de um termo. Por exemplo:

```
?- free_variables(t(X,Y,Z),V).
X = _G297, Y = _G298, Z = _G299
V = [_G297, _G298, _G299]
```

Segue o código para o predicado `limpa`. O predicado `escolhe_xyz` seleciona uma constante para cada uma das variáveis livres.

```
1 limpa(A):-free_variables(A,F), escolhe_xyz(F,[x,y,z]).
2 escolhe_xyz([X|Xs],XYZ):-select(X,XYZ,XYZ1),!,escolhe_xyz(Xs,XYZ1).
3 escolhe_xyz([],_).
4 %%
```

A partir do predicado `limpa`, fizemos a versão `do3a`, que gera as formas lógicas, já limpas, como segue:

```
?-do3a.
   ok:[um, cao, late]
forma:[existe(x), cao(x), [late(x)]]
   ok:[bidu, persegue, mikey]
forma:[existe(x), x=bidu, [existe(y), y=mikey, persegue(x, y)]]
   ok:[bidu, persegue, todo, gato]
forma:[existe(x), x=bidu, [todo(y), gato(y), persegue(x, y)]]
   ok:[todo, gato, persegue, todo, rato]
forma:[todo(x), gato(x), [todo(y), rato(y), persegue(x, y)]]
   ok:[todo, rato, come, um, queijo]
forma:[todo(x), rato(x), [existe(y), queijo(y), come(x, y)]]
   ok:[algum, cao, late]
forma:[existe(x), cao(x), [late(x)]]
Yes
```

PLN compreende processamento em três principais níveis: Léxico, sintático e semântico. Usando os recursos do Prolog principalmente DCG é fácil trabalhar nestes três níveis.

CAPÍTULO 17

IA: Classificação KNN

Em muitos cursos de Inteligência Artificial (IA) os alunos apenas utilizam simuladores e/ou softwares de aprendizagem de máquina onde os algoritmos são caixas pretas. O objetivo deste e dos próximos capítulos e desmistificar algumas destas caixas pretas mostrando o interior de alguns métodos de aprendizagem de máquina.

A partir deste capítulo temos uma seqüência de estudos de casos em IA. O objetivo é explorar a programação de técnicas das disciplinas de IA e Mineração de Dados. Serão programados os algoritmos de: 1) classificação via KNN e Bayes; 2) classificação via indução de regras; 3)agrupamento (kmeans) e 4)ngramas para comparação. Nestes capítulos não apresentamos uma teoria para cada um destes métodos, o leitor deve procurar estudá-los em livros especializados de IA ou de Mineração de Dados.

Além de explorarmos as facilidades do Prolog para codificar as soluções em si, exploramos também três formas alternativas de representar os dados de entrada para estes algoritmos: como fatos, como listas de instâncias ou uma forma que combina estas duas. Ainda, nestes capítulos são explorados os diferentes tipos de dados utilizados em algoritmos de aprendizagem e mineração: dados discretos e dados contínuos. São apresentadas diferentes métricas de distância e similaridade para ambos os tipos de dados. Por fim, são apresentadas também métricas relacionadas com quantidade de informação e entropia.

17.1 KNN: K vizinhos mais próximos

O nome KNN vem do termo inglês *K Nearest Neighbors* que pode ser traduzido para *K vizinhos mais próximos*. A idéia do algoritmo é classificar uma determinada instância (ou exemplo) a partir de uma base de instâncias (exemplos) de treinamento. Assim, a entrada do algoritmo é um conjunto de instâncias de treinamento e uma instância de teste. A saída é a classificação para a instância de teste.

17.2 Base de dados

Para se fazer múltiplos testes com o algoritmo, a entrada é uma base de exemplos, que é dividida em duas partições, uma para treinamento e outra para testes. Uma forma de particionar a base é 2/3 treinamento e 1/3 para testes. A cada vez que se roda o algoritmo um dos exemplos dos testes é avaliado.

Segue abaixo uma pequena base para ilustrar o funcionamento do algoritmo. Para cada instância temos uma lista de pares `atributo=valor` e uma classificação: `approve` ou `reject` (crédito aprovado ou rejeitado). Cada exemplo é enumerado com um identificador único.

```
1  %% ia-dados.pl   /* data set 2/3 - 1/3 split */
2  /* training set */
3  example(1,  approve, [emp=yes, buy=comp, sex=f, married=no]).
4  example(2,  reject,  [emp=no,  buy=comp, sex=f, married=yes]).
5  example(3,  approve, [emp=yes, buy=comp, sex=m, married=no]).
6  example(4,  approve, [emp=yes, buy=car,  sex=f, married=yes]).
7  example(6,  approve, [emp=yes, buy=comp, sex=f, married=yes]).
8  example(7,  approve, [emp=yes, buy=comp, sex=f, married=no]).
9  example(8,  approve, [emp=yes, buy=comp, sex=m, married=no]).
10 example(11, reject,  [emp=no,  buy=comp, sex=m, married=yes]).
11 /* test set */
12 test(5,  reject,  [emp=yes, buy=car,  sex=f, married=no]).
13 test(9,  approve, [emp=yes, buy=comp, sex=m, married=yes]).
14 test(10, approve, [emp=yes, buy=comp, sex=m, married=yes]).
15 test(12, reject,  [emp=no,  buy=car,  sex=f, married=yes]).
```

Um classificador KNN tem como objetivo inferir a classificação de um dos exemplos de testes a partir da base de treinamento. Para um dado valor K e uma instância teste, ele busca encontrar os K vizinhos mais similares à instância de teste; a classe inferida é a classe que mais ocorre entre os k vizinhos. Assim o `knn/3` tem como parâmetros uma instância e o valor de K, retornando a classe inferida.

Para saber quem são os k vizinhos mais similares (`neighbors`) precisamos de um predicado que mede a distância (`dist`) ou a similaridade; quanto maior a distância menor é a similaridade. Para atributos discretos uma forma simples de calcular a distância é contar o número de atributos com valor diferente entre duas instâncias. Seguem alguns testes para os predicados `knn`, `dist` e `neighbors`.

```
?- knn([emp=yes, buy=car, sex=f, married=no], 3,X).
   X = approve
?- neighbors([emp=yes, buy=car, sex=f, married=no], 3,X).
   X = [1-approve, 1-approve, 1-approve]
?- dist( [emp=yes, buy=car,  sex=f], [emp=yes, buy=comp, sex=m],X).
   X = 2
```

A base de teste é para validar o resultado do KNN: saber em quantos testes ele acerta e em quantos ele erra: para `K`=3 o teste 9 acerta e o teste 5 falha, ver abaixo.

```
?- test(5,C,X),knn(X,3,C1).
   C = reject  ... C1 = approve
```

```
?- test(9,C,X),knn(X,3,C1).
  C = approve  ... C1 = approve
?- test(9,_,X),neighbors(X,3,L).
   X = [emp=yes, buy=comp, sex=m, married=yes]
   L = [1-approve, 1-approve, 1-approve]
?- test(5,C,X),example(N,P,Y),dist(X,Y,D),write(N:[D-P]),nl,fail.
   1:[1-approve]
   2:[3-reject]
   3:[2-approve]
   4:[1-approve]
   6:[2-approve]
   7:[1-approve]
   8:[2-approve]
  11:[4-reject]
```

Existem duas formas de inferir a classe a partir dos K vizinhos: 1) simplesmente pegar a classe mais freqüente (moda) ou 2)considerar também a distância como um peso, pois a distância varia de 0 a 4 (de todos iguais à todos diferentes). Fizemos duas versões do KNN, numa delas a distância é considerada: calcula-se o somatório do peso de cada classe $1/(1+(D_i)^2), i:1..K$; quanto maior a distância menor é o peso.

Para calcular a freqüência de cada classe nos K vizinhos usamos o predicado `selClass/4` que seleciona uma lista com os pares Dist-Classe. A freqüência de cada classe é calculada pelo predicado `sumClass` que retorna uma lista de pares Soma-Classe; não confundi-los com os pares Dist-Classe retornados pelos outros predicados. Para o `knn` simples o `sumClass` apenas pega o tamanho da lista de cada classe, pois a distância é ignorada. Já para para o `knnW` com peso, usando o predicado `maplist`, aplicamos o peso para cada distância e depois somamos os valores. Assim, o `sumClassW` devolve um valor ponderado pela distância de cada instância. Veja abaixo exemplos de execução destes predicados.

```
?- maplist(funcw,[1,2,5,7],X),sumlist(X,S).
   X = [1/ (1+1*1), 1/ (1+2*2), 1/ (1+5*5), 1/ (1+7*7)]
   S = 0.758462
?- sumClass([1-approve,  2-approve, 2-approve, 3-reject, 4-reject],S).
      S = [3-approve, 2-reject]
?- selClass(approve,[2-approve,  2-approve, 1-reject, 4-reject],S,Lo).
   S = [2-approve, 2-approve]
   Lo = [1-reject, 4-reject]
?- test(5,C,X),neighbors(X,6,L),sumClassW(L,S).
   C = reject X = ...
   L = [1-approve, 1-approve, 1-approve, 2-approve, 2-approve, 2-approve]
   S = [2.1-approve]
?- sumClassW([1-approve,  2-approve, 2-approve, 3-reject, 4-reject],S).
   S = [0.9-approve, 0.158824-reject]
```

17.3 O algoritmo KNN

Segue abaixo o programa principal. Os predicados `knn` e `knnw` foram comentados acima. O programa em si é pequeno, aproximadamente 30 linhas, mostrando que o Prolog é uma linguagem compacta e expressiva. Para rodar o programa é necessário carregar também o arquivo de dados (`ia-dados.pl`).

```
%% ia-knn.pl %% Versão adaptada do original de Zdravko Markov
:- [ia-dados.pl].
knn(Instance,K,Class) :-
     neighbors(Instance,K,Neighbors),
     sumClass(Neighbors,Sum),max(Sum,_-Class).
knnW(Instance,K,Class) :-
     neighbors(Instance,K,Neighbors),
     sumClassW(Neighbors,Sum), max(Sum,_-Class).
neighbors(Instance,K,Neighbors) :-
    findall(D-C,(example(_,C,E),dist(Instance,E,D)),Ds),
    keysort(Ds,L),
    first(K,L,Neighbors).
%%
dist([],[],0). %% nro de A=V diferentes
dist([X|Xs],[X|Ys],N) :- !,dist(Xs,Ys,N).
dist([_|Xs],[_|Ys],N) :-   dist(Xs,Ys,M), N is M+1.
sumClass([],[]).
sumClass([N-C|Ls],[S-C|R]) :- sumOneClass(C,[N-C|Ls],Lo,S), sumClass(Lo,R).
sumOneClass(C,Li,Lo,S):-selClass(C,Li,Co,Lo),zipper(Co,Ns,Cs),length(Ns,S).
selClass(X,L,[No-X|Co],Lo) :-select(No-X,L,L1),selClass(X,L1,Co,Lo).
selClass(X,L,[],L).
sumClassW([],[]).
sumClassW([N-C|Ls],[S-C|R]) :- sumOneClassW(C,[N-C|Ls],Lo,S), sumClassW(Lo,R).
sumOneClassW(C,Li,Lo,S):-selClass(C,Li,Co,Lo),zipper(Co,Ns,Cs),
                                        maplist(funcw,Ns,Nsw),sumlist(Nsw,S).
funcw(X, 1/(1+X*X) ).
%%
zipper([(X-Y)|XYs],[X|Xs],[Y|Ys]):-!,zipper(XYs,Xs,Ys).
zipper([],[],[]).
max(L,M):-msort(L,Lo),last(Lo,M).
first(K,L,KL):-length(KL,K),append(KL,_,L).
```

A partir deste código podem-se fazer vários projetos interessantes, alguns estão nos exercícios que seguem.

Exercício 17.1 *Adaptar o algoritmo para trabalhar com dados discretos e contínuos usando métricas de distância para dados contínuos, as quais são apresentadas nos próximos capítulos.*

Exercício 17.2 *Ler um arquivo de testes do software para mineração de dados do Weka[18] e processá-lo gerando as mesmas saídas que o Weka. Use o formato do Weka: ARFF.*

Exercício 17.3 *Dada uma base de testes, permitir a seleção automática do melhor K para uma faixa de valores. Ele deve fazer uma avaliação sistemática do seu treinamento.*

Exercício 17.4 *Na seleção automática do melhor K, implementar o método de testes* N-fold cross-validation *descrito abaixo. O N é um parâmetro.*

N-fold Cross Validation: divida os dados em N partições; rode N iterações de treinamenos+testes; cada iteração usa N-1 partições para treinamento e uma para teste. A performance do classificador é medida pela média dos resultados das N iterações. Um caso especial deste método é chamado de *Leave-one-out cross-validation*: divide um conjunto com M dados em M partições.

CAPÍTULO 18

IA: Agregação Kmeans

Iniciamos apresentando uma biblioteca de funções utilitárias; depois falamos sobre como criar uma população para testes e por fim mostramos o algoritmo de agrupamento kmeans.

18.1 Medidas de similaridade e distância

Neste capítulo vamos trabalhar com dados do tipo contínuo. Para trabalhar com estes dados usam-se funções da área de estatística, tais como média, desvio padrão, correlação, etc. Além disso, usa-se outras que são da matemática, tais como coseno entre dois ângulos num espaço vetorial de dimensão N.

```
1  %% ia-similar.pl
2  sum([X|Xs],S):-sum(Xs,Sx),S is Sx+X.
3  sum([],0).
4  medv(V,M):-sum(V,S),length(V,L),M is S/L.
5  sumDifXY([X|Xs],[Y|Ys], S):- sumDifXY(Xs,Ys, Ss), S is abs(X-Y)+Ss.
6  sumDifXY([],[],0).
7  prodX([X|Xs], S):- prodX(Xs, Ss), S is (X*X)+Ss.
8  prodX([],0).
9  prodXY([X|Xs],[Y|Ys], S):- prodXY(Xs,Ys, Ss), S is (X*Y)+Ss.
10 prodXY([],[],0).
11 prodDifXY([X|Xs],[Y|Ys], S):- prodDifXY(Xs,Ys, Ss), S is (X-Y)*(X-Y)+Ss.
12 prodDifXY([],[],0).
13 prodDifMXY([X|Xs],[Y|Ys],Mx,My,S):-
14                 prodDifMXY(Xs,Ys,Mx,My, Ss), S is (X-Mx)*(Y-My)+Ss.
15 prodDifMXY([],[],_,_,0).
16 prodDifMX([X|Xs],Mx, S):- prodDifMX(Xs,Mx, Ss), S is (X-Mx)*(X-Mx)+Ss.
17 prodDifMX([],_,0).
18 prodMinkXY([X|Xs],[Y|Ys],P, S):- prodMinkXY(Xs,Ys,P, Ss), S is abs(X-Y)^P+Ss.
19 prodMinkXY([],[],P,0).
20 cut0([X|Xs],[Y|Ys], Xo,Yo):-
```

```
    (X\==0, Y\==0,!,Xo=[X|Xso],Yo=[Y|Yso];Xo=Xso,Yo=Yso),cut0(Xs,Ys,Xso,Yso).
cut0([],[],[],[]).
%%-----------------------------------------------------
%% estatistica
var(X,V):- medv(X,Mx),prodDifMX(X,Mx,Sx),length(X,N),V is Sx/N.
std(X,D):-var(X,V),D is sqrt(V).
covar(X,Y,V):- medv(X,Mx),medv(Y,My),prodDifMXY(X,Y,Mx,My,Sxy),
               length(X,N),V is Sxy/(N-1).
covar2(X,Y,V):- medv(X,Mx),medv(Y,My),prodXY(X,Y,Sxy),
                length(X,N),V is (Sxy/N)-(Mx*My).
%% regressão
beta(X,Y,Mx,My,B):- prodDifMXY(X,Y,Mx,My,Sxy),prodDifMX(X,Mx,Sx),B is Sxy/Sx.
alfa(X,Y,Mx,My,A):- beta(X,Y,Mx,My,B), A is My - B*Mx.
regress(X,Y,A,B):- medv(X,Mx),medv(Y,My),beta(X,Y,Mx,My,B),alfa(X,Y,Mx,My,A).
%%-----------------------------------------------------
%% distância & similaridade
distEu(X,Y,D):- prodDifXY(X,Y,P), D is sqrt(P).
distMink(X,Y,D,P):-prodMinkXY(X,Y,P,DP),D is DP^(1/P).
coseno(X,Y,P):- prodXY(X,Y,Sxy),prodX(X,Sx),prodX(Y,Sy),
                P is Sxy/(sqrt(Sx)*sqrt(Sy)).
pearsMed(X,Y,Mx,My,P):- prodDifMXY(X,Y,Mx,My,Sxy),sumDifX(X,Mx,Sx),
             prodDifMX(Y,My,Sy),P is Sxy/(sqrt(Sx)*sqrt(Sy)).
pearson(X,Y,C):-length(X,N),sum(X,Sx), sum(Y,Sy),prodXY(X,Y,Sxy),
                prodX(X,Sxx),prodX(Y,Syy),
                C is (N*Sxy-Sx*Sy)/(sqrt(N*Sxx-Sx*Sx)*sqrt(N*Syy-Sy*Sy)).
pearson0(X,Y,P):- cut0(X,Y,X0,Y0),pearson(X0,Y0,P).
coseno0(X,Y,P):- cut0(X,Y,X0,Y0),pearson(X0,Y0,P).
d1Mink(X,Y,P):- distMink(X,Y,P,1).
d2Mink(X,Y,P):- distMink(X,Y,P,2).
d3Mink(X,Y,P):- distMink(X,Y,P,3).
zipper([(X,Y)|XYs],[X|Xs],[Y|Ys]):-!,zipper(XYs,Xs,Ys).
zipper([(X-Y)|XYs],[X|Xs],[Y|Ys]):-!,zipper(XYs,Xs,Ys).
zipper([],[],[]).
%%-----------------------------------------------------
%% base de testes
v1([1,0,6, 4,3,1, 2,0,0,2]).
v2([1,5,6, 3,2,1, 2,1,3,0]).
%vet(Med,NM, [ABC,  DEF    GHIJ])
vet(_,luis, [1,0,0, 2,1,7, 0,0,2,3]).
vet(_,edu,  [5,1,2, 2,1,7, 1,0,6,5]).
vet(_,mari, [1,5,6, 4,3,1, 2,0,0,2]).
vet(_,mara, [4,2,2, 0,0,6, 1,6,0,6]).
vet(_,lau,  [6,5,4, 1,2,0, 0,5,0,3]).
%
cos(X,Y,C):- vet(_,X,Vx),vet(_,Y,Vy),coseno(Vx,Vy,C).
dds(D):-member(D,[d1Mink, d2Mink, d3Mink, distEu, coseno, pearson]).
%% compara todos
do(D,Y):- bagof(X,X^V^vet(_,X,V),CAB),tab(10),writeln(CAB),
          bagof((V,X),(X^C^Vx^Vy^(vet(_,X,Vx),vet(_,Y,Vy),
          C=..[D,Vx,Vy,V],C)),M),write((D,Y:' ')),wl(M).
wl(M):-zipper(M,M1,M2),wl0(M1).
```

```
72 wl0([X|Xs]):-format('~2f|',X),wl0(Xs).
73 wl0([]):-nl.
```

O código acima inicia com várias funções do tipo somatório que são auxiliares para codificar as funções estatísticas, como a variância, o desvio padrão, a correlação, etc. O leitor pode testar cada uma delas isoladamente, passando um ou dois vetores conforme for o caso. Abaixo mostramos a execução de alguns predicados estatísticos.

```
?- var([40,56,38, 38,63,59, 52,49,46],V).  V = 76.2222
?- std([40,56,38, 38,63,59, 52,49,46],D).  D= 8.73053
?- covar([1,2,3,4,5,6],[6,5,4,3,2,1],V).    V = -3.5
?- covar2([1,2,3,4,5,6],[6,5,4,3,2,1],V).   V = -2.91667
```

Temos também a função de *regressão linear simples*. Basicamente codifica-se as fórmulas: `Y=a+bX`, `a=my-b(mx)` e `b=sum((x-mx) *(Y-my)) / sum((x-mx) *(x-mx))`. Onde `sum` é somatório, `mx` e `my` são as medias dos vetores `X` e `Y`. Segue abaixo um exemplo de teste, onde são retornados os valores `A` e `B` com os quais podemos montar a equação: `Y = 23.209 + 3.53748*X`.

```
?- regress([3,8,9, 13,3,6, 11,21,1,16], [30,57,64, 72,36,43, 59,90,20,83],A,B).
   A = 23.209,   B = 3.53748
```

Ainda temos duas medidas de *similaridade*: correlação e coseno. A **correlação** de Person (ou só correlação) entre dois vetores retorna um valor entre 0 e 1. Se for 1, eles estão fortemente correlacionados, isto é, os valores de um vetor podem predizer os valores do outro. Se for 0, não existe correlação. E se for -1, existe uma correlação inversamente proporcional.

O **coseno** é similar à correlação devolvendo valores entre 0 e 1. Ele mede o ângulo entre dois vetores num espaço vetorial. Quanto mais próximo de 1 for o valor, mais similares são os dois vetores.

Ao comparar dois vetores com a correlação é bom excluir os itens com zeros em um dos vetores, pois os zeros afetam bastante a correlação. Para isso, usa-se o predicado `cut0/3`. Diferente da correlação, o coseno não é tão sensível à presença de zeros.

Depois, temos as medidas de distância. Quanto menor à distância mais similares são dois objetos. As medidas de distância num espaço de dimensão N, podem ser generalizadas pela medida de Minkowski, onde um parâmetro é o valor da potência; com potência de dois corresponde à distância Euclidiana.

Seguem alguns testes para estas medidas de similaridade e distância. Os dois últimos testes comparam coseno e pearson com e sem os zeros.

```
?- pearson([1,2,3],[3,2,1],X).    X = -1.0
?- pearson([1,2,3],[1,2,3],X).    X = 1.0
?- coseno([1,2,3],[1,2,3],X).    X = 1.0
?- coseno([1,2,3],[3,2,1],X).    X = 0.714286
?- d2Mink([1,2,3],[1,2,3],X).    X=0.0
?- v1(X),v2(Y),d2Mink(X,Y,P).    P = 6.40312
?- v1(X),v2(Y),cut0(X,Y,Xo,Yo),coseno(X,Y,P),coseno(Xo,Yo,Po),write(P:Po),nl.
   0.750587 : 0.988399
?- v1(X),v2(Y),cut0(X,Y,Xo,Yo),pearson(X,Y,P),pearson(Xo,Yo,Po),write(P:Po),nl.
0.42823 : 0.963952
```

No final do programa `ia-similar.pl` existe uma pequena tabela de dados de clientes. Cada cliente possui um vetor de atributos. Podemos compará-los todos com todos, como mostrado abaixo.

```
?- dds(M), do(M,luis).
             [luis, edu, mari, mara, lau]
d1Mink, luis: 0.00|14.00|26.00|23.00|30.00|
...
distEu, luis: 0.00|6.48|10.68|8.54|12.08|
coseno, luis: 1.00|0.86|0.31|0.67|0.21|
pearson, luis: 1.00|0.78|-0.33|0.41|-0.50|
...
```

18.2 Criando uma população

Antes de apresentarmos o método de agrupamento (clustering) `kmeans` vamos mostrar como criar uma população, isto é, um conjunto de fatos relativos a pessoas, fazendo uso de predicados já apresentados aqui e no capítulo sobre programação aritmética.

Pensamos em minerar um grupo de indivíduos do sexo masculino ou feminino. Inicialmente teremos apenas três dados para cada indivíduo: sexo, peso e altura. Segue abaixo uma tabela com dados estatísticos sobre peso e altura de homens e mulheres.

Para testar um algoritmo de cluster vamos criar uma população fictícia baseada nesta tabela de peso e altura. Temos três colunas: altura em centímetros, peso (min-max) para homem e peso (min-max) para mulher. A menor altura é 147 e a maior é 190. Dentro desta faixa, a tabela tem valor zero na coluna altura para homens com altura menor que 155 e também para mulheres com altura maior que 180. Estes campos tem valor zero pois estatisticamente as ocorrências são poucas.

```
1  %% ia-pop.pl %% gerar uma população
2  :- [stat].   %% cap prog aritmética
3  peso(147, 0,     45-59).
4  peso(150, 0,     45-60).
5  peso(152, 0,     46-62).
6  peso(155, 55-66, 47-63).
7  peso(157, 56-67, 49-65).
8  peso(160, 57-68, 50-67).
9  peso(162, 58-70, 51-69).
10 peso(165, 59-72, 53-70).
11 peso(167, 60-74, 54-72).
12 peso(170, 61-75, 55-74).
13 peso(172, 62-77, 57-75).
14 peso(175, 63-79, 58-77).
15 peso(177, 64-81, 60-78).
16 peso(180, 65-83, 61-80).
17 peso(182, 66-85, 0).
18 peso(185, 68-87, 0).
19 peso(187, 69-89, 0).
20 peso(190, 71-91, 0).
21 hHomem(Xs,N):-findall(X,(between(1,N,_),X is normal(172,11)),Xs).
```

```
hMulher(Xs,N):-findall(X,(between(1,N,_),X is normal(163,10)),Xs).
popn(POP,N):- pop(P), length(POP,N),append(POP,_,P).
pop(P):-hHomem(H,1000),hMulher(M,1000),
        criaPar(h,H,HP),criaPar(m,M,MP),append(HP,MP,P).
criaPar(SX,[X|Xs],[Y|Ys]):- Y=(X-P)-SX,criaPeso(SX,X,P),criaPar(SX,Xs,Ys).
criaPar(_,[],[]).
ajusta(h,X,155):-X<155.
ajusta(h,X,190):-X>190.
ajusta(m,X,147):-X<147.
ajusta(m,X,180):-X>190.
criaPeso(SX,X,P):-ajusta(SX,X,Xo),criaPeso0(SX,Xo,P),!;criaPeso0(SX,X,P).
criaPeso0(h,X,P):-peso(X,N-M,_),P is 1+N+random(M-N),!.
criaPeso0(m,X,P):-peso(X,_,N-M),P is 1+N+random(M-N),!.
criaPeso0(SX,X,P):- X1 is X-1, criaPeso0(SX,X1,P).
```

O problema inicial é: a partir das informações desta tabela gerar uma população de 1000 individuos (ou mais), com dados mais ou menos reais de peso e altura. Assumimos que a altura das pessoas segue uma distribuição normal e usando a função `normal`, apresentada no capítulo sobre programação aritmética, geramos uma população de 1000 indivíduos. Inicialmente calculamos a média e o desvio padrão da altura, como mostrado abaixo. Com estes valores de médias e de desvios podemos gerar randomicamente quantos indivíduos quisermos, para cada sexo, como exemplificado abaixo.

```
?- bagof(A,(H^M^peso(A,H,M),H\=0),L),medv(L,Med),std(L,Std).
   Med = 172.267   Std = 10.8041
?- bagof(A,(H^M^peso(A,H,M),M\=0),L),medv(L,Med),std(L,Std).
   Med = 163.5   Std = 10.1119
?- findall(X,(between(1,20,_),X is normal(163,10)),L0).
   L0 = [172, 159, 156, 168, 175, 155, 167, 171, 150|...]
?- criaPeso(h,176,P). P = 70
?- criaPeso(m,176,P). P = 66
?- popn(P,5).
   P = [169-62-h, 166-67-h, 191-72-h, 169-68-h, 172-65-h]
```

Ainda, usando a tabela de pesos e alturas, dada uma altura e o sexo podemos inferir um peso. Como a tabela tem um valor mínimo e máximo de peso, usamos a função `random` para chutar um valor entre o mínimo e o máximo. O predicado `criaPeso` codifica esta funcionalidade. Ele trata também os valores para os homens e as mulheres que estão fora da faixa; se a altura for menor que o mínimo da tabela é atribuído o valor mínimo de peso; se for maior que o máximo da tabela, é atribuído o valor máximo do peso; por fim, para uma altura dentro da faixa, mas não cadastrada, e.g., 161, é atribuído o valor do peso do individuo menor mais próximo, 160.

O predicado `criaPar` gera uma lista de triplas altura-peso-sexo, a partir de uma lista de alturas associada a um dado sexo (`h/m`). O predicado `pop` gera uma população inicial de 1000 indivíduos. E o predicado `popn` permite selecionar um grupo com qualquer número de indivíduos, pois quando estamos testando o programa KNN pelas primeiras vezes é bom trabalhar com um número mínimo de indivíduos (tipo 5), facilitando o trace do programa. A medida que o programa se torna estável vamos aumentando o grupo de testes até testarmos com a população toda.

Exercício 18.1 *Como vemos o predicado* `popn/2`*, para valores menores que 500, retorna só homens pois foram criadas duas listas, uma com 500 homens e outra com 500 mulheres, que*

foram concatenadas com `append`. *Modifique o programa com um predicado que intercala um a um, os homens e as mulheres.*

Exercício 18.2 *Utilizando o predicado de regressão linear simples e os dados da tabela, crie as funções* `pesoMin`, `pesoMax` *para homens e para mulheres. Permitindo calcular os pesos para qualquer valor positivo de altura.*

Exercício 18.3 *Incorpore as funções* `pesoMin` *e* `pesoMax` *no código* `ia-pop.pl`.

18.3 Kmeans: Agrupamento

O método kmeans é um método de agrupamento (clustering) bastante difundido. Espera-se que o código abaixo sirva de base para o desenvolvimento de projetos de programação interessantes, sobre as inúmeras variações do kmeans. Esta versão trabalha apenas com dois atributos (peso e altura) para cada indivíduo, porém o método funciona para um número qualquer de atributos.

Seja uma população de tamanho N e seja um valor K. O kmeans cria K agrupamentos para a população. Ele é baseado em uma métrica de similaridade (ou de distância). Cada agrupamento é feito em torno de um indivíduo (ou dos atributos dele) chamado centro (ou centróide), com os indivíduos mais similares.

Esta versão do algoritmo kmeans segue as etapas:

- **(1) inicialização:** gera-se K valores randômicos (entre 1 e N) para selecionar K indivíduos iniciais, que são os centros dos K grupos;
- (2) se houver um centro repetido volta-se à etapa 1; senão inicia-se o passo que gera os agrupamentos, tendo como entrada a população e os K centros;
- **(3) agrupamentos:** para cada um dos N indivíduos calcula-se à distância contra os K centros; escolhe-se como grupo o de menor distância; no final desta etapa temos K grupos de indivíduos;
- (4) calcula-se a média dos valores dos indivíduos de cada um dos K grupos; para cada atributo calcula-se uma média; seleciona-se K novos centros a partir dos valores das médias;
- **(5) testa-se o fim:** compara-se os novos centros com os anteriores, se são diferentes retorna-se a etapa 3; senão termina-se o processo, mostrando a média e o desvio padrão da distância entre o centro e os elementos de cada grupo.

Segue abaixo a execução do `kmeans` para agrupar 50 indivíduos em 3 grupos. Foram necessárias 4 iterações. No final temos, para cada grupo, a média e o desvio padrão das distâncias dos elementos ao seu centro. Se rodarmos o programa novamente outros resultado serão produzidos pois os valores iniciais dos grupos são diferentes. As médias e desvios mostram a qualidade de cada grupo.

Normalmente o kmeans converge rápido, mas para grandes populações o número de iterações é bem maior. Além disso, na prática os métodos de mineração exploram outros aspectos do processo, por exemplo, para descobrir qual é o melhor valor para K; ou eliminar dados identificados como *desvios* ou excepcionais, melhorando assim a coesão do grupo.

```
?- popn(P,50),kmeans(P,3).
Centros iniciais:[146-58, 168-61, 178-81]:[27, 39, 8]
---Passo:1: ---
cluster:[28*1, 29*1, 31*1]
cluster:[3*2, 4*2, 5*2, 6*2, 7*2, 10*2, 12*2, ... 7*2, 49*2, 50*2]
cluster:[1*3, 2*3, 8*3, 9*3, 11*3, 13*3, 15*3, ... 43*3, 44*3, 48*3]
Novos Centros:[146-59, 167-64, 182-78]
---Passo:2: ---
cluster:[28*1, 29*1, 31*1]
cluster:[3*2, 4*2, 5*2, 6*2, 7*2, ... 45*2, 46*2, 47*2, 49*2, 50*2]
cluster:[1*3, 2*3, 8*3, 9*3, 11*3, ... 43*3, 44*3, 48*3]
Novos Centros:[146-59, 169-65, 182-78]
---Passo:3: ---
cluster:[28*1, 29*1, 31*1, 46*1]
cluster:[3*2, 4*2, 5*2, 6*2, 7*2, ...  42*2, 45*2, 47*2, 49*2, 50*2]
cluster:[1*3, 2*3, 9*3, 11*3, 13*3, ... 41*3, 43*3, 44*3, 48*3]
Novos Centros:[156-59, 169-65, 182-78]
---Passo:4: ---
cluster:[19*1, 21*1, 28*1, 29*1, 31*1, 46*1]
cluster:[3*2, 4*2, 5*2, 6*2, 7*2, 8*2, ...  47*2, 49*2, 50*2]
cluster:[1*3, 2*3, 9*3, 11*3, 13*3, 15*3, ...  41*3, 43*3, 44*3, 48*3]
Novos Centros:[156-59, 169-65, 182-78]
Fim no Passo:4
[156-59, 169-65, 182-78]
Qualidade:[cluster:1:3.28153:1.76254, cluster:2:4.11917:1.65603,
           cluster:3:5.2229:2.33628]
```

O código é fortemente baseado no predicado `findall(_,between(1,LN,I),..)`. A população é uma lista `POP` de comprimento `LN`; este comando de controle permite percorrer a lista e fazer algum tipo de processamento para cada indivíduo. Uma estrutura paralela à lista de indivíduos é a lista dos agrupamentos chamada de `CLU`; nesta, `10*3` denota que o individuo 10 está no agrupamento 3. Assim, o mesmo comando de controle serve para percorrer também os agrupamentos. Associado ao `findall` temos o predicado `nth1(I,POP,IND)`; neste caso, estamos acessando o Iésimo INDivíduo da POPulação.

De forma similar, com `findall(_,(between(1,K,I),..)` estamos percorrendo a lista dos CENtros; com `nth1(I,CEN,C)` estamos acessando o centro C de índice `I`.

O predicado `kmeansSTEP` tem duas versões de centros: `CEN` são os centros de entrada e `CENo` são os novos centros, criados na corrente etapa. Se eles são iguais então o processo termina.

```
1  %% ia-kmeans.pl   %% Cluster Kmeans
2  wCluster(C/LN/K):- findall(K,  (I^between(1,K,I),  wCluster(C/LN,I)),_).
3  wCluster(C/LN,K):- findall(I*K,(I^between(1,LN,I), nth1(I,C,I*K)),CLU),
4                     writeln(cluster:CLU),!.
5  kmeans(Pi,K):-
6        length(Pi,LN), zipper(Pi,POP,_),
7        findall(XY,(between(1,K,_),R is random(LN),nth0(R,POP,XY)),CEN),
8        (dupl(CEN)->kmeans(Pi,K);
9           writeln('Centros iniciais':CEN:Ci),
10          kmeansSTEP(1,POP/LN,CEN/K)).
11 kmeansSTEP(Pas,POP/LN,CEN/K):-
```

```
12      writeln('---Passo':Pas:'---'),
13      findall(I*C,(between(1,LN,I),
14             nth1(I,POP,IND),getIdvCLU(IND,CEN/K,C)),CLU),
15      wCluster(CLU/LN/K),
16      findall(Ck,(Ki^between(1,K,Ki),getCENm(POP/LN, Ki,CLU,Med),
17                                    getCENi(POP/LN, Med,Ck)),CENo),
18      writeln(('Novos Centros':CENo)),
19      (CEN=CENo->writeln('Fim no Passo':Pas),writeln(CEN),
20          getCENms(POP/LN, CEN/K,CLU,LMS),writeln('Qualidade':LMS)
21       ;Pas1 is Pas+1,kmeansSTEP(Pas1,POP/LN,CENo/K)).
22  getCENm(POP/LN, K,CLU,Meo):-
23       findall(IND,(between(1,LN,I), nth1(I,CLU,I*K),nth1(I,POP,IND)),INDs),
24       zipper(INDs,H,P),medv(H,Malt),medv(P,Mpeso),Meo=Malt-Mpeso,!.
25  getCENi(POP/LN, Med,Ci):-
26       findall(D,(between(1,LN,I),nth1(I,POP,IND),dist0(Med,IND,D)),Ds),
27       minL(Ds,Min),nth1(Io,Ds,Min), nth1(Io,POP,Ci),!.
28  getIdvCLU(IND,CEN/K,CLU):-
29       findall(D,(between(1,K,I), nth1(I,CEN,CEi),dist0(CEi,IND,D)),Ds),
30       minL(Ds,Min),nth1(CLU,Ds,Min),!.
31  %% Avalia a qualidade
32  getCENms(POP/LN, CEN/K,CLU,LMS):-
33      findall((cluster:I:Meo:Std),(between(1,K,I),
34                    getCENms0(POP/LN, CEN/I,CLU,Meo,Std)),LMS).
35  getCENms0(POP/LN, CEN/K,CLU,Meo,Std):-
36       nth1(K,CEN,CEi),
37       findall(D,(between(1,LN,I),
38           nth1(I,CLU,I*K),nth1(I,POP,IND),dist0(CEi,IND,D)),Ds),
39           medv(Ds,Meo), std(Ds,Std),!.
40  dist0(X-Y,X1-Y1,D):-distEu([X/2,Y],[X1/2,Y1],D). %% normalizar X/2
41  minL([L|Ls],M):-minL(Ls,L,M).
42  minL([L|Ls],ACC,M):- (ACC<L->minL(Ls,ACC,M);minL(Ls,L,M)).
43  minL([],M,M).
44  dupl([X|Xs]):-member(X,Xs);dupl(Xs).
```

Os predicados do kmeans não são facilmente testados de forma isolada, pois é trabalhoso criar para eles os dados de entrada. O melhor método para depurar um código deste tipo é iniciar uma população mínima, pode ser 3 indivíduos e um cluster; faz-se um trace passo a passo do programa, vendo o que entra e o que sai para identificar os possíveis erros.

No predicado `kmeans(Pi,K)` entra uma população e um valor K inteiro. Com o predicado `zipper` jogamos fora o atributo sexo ficando apenas o peso e a altura. Depois, se geram K valores randômicos para selecionar os K indivíduos que são os K centros iniciais.

No predicado `kmeansSTEP(Pas,POP/LN,CEN/K)` entra o número da iteração (Passo), a população `POP/LN` e os centros `CEN/K`; `LN` e `K` representam respectivamente o tamanho da população e o número de grupos.

O predicado `getCENm(POP/LN, K,CLU,Meo)` recupera, em `Meo`, as duas médias dos atributos para o grupo K. Ele precisa percorrer o `CLU` e pegar os valores reais na população. O predicado `getCENi(POP/LN, Med,Ci)` retorna o indivíduo mais próximo das médias de um grupo.

O predicado `getCENms(POP/LN, CEN/K,CLU,LMS)` retorna uma lista com a média e o desvio padrão de cada cluster; ele chama o `getCENms0(POP/LN, CEN/K,CLU,Meo,Std)` que retorna a média e o desvio de um cluster.

O predicado `dist0(X-Y,X1-Y1,D)` chama o predicado que calcula a distância euclidiana; ele passa os valores da altura normalizados, isto é, altura dividida por 2. Neste ponto, pode-se substituir a função que calcula a distância por uma outra, como as mostradas anteriormente.

A partir deste código podem ser feitos vários projetos novos, tais como, os dos exercícios que seguem.

Exercício 18.4 *Ler um arquivo de entrada com os dados a serem agrupados; gerar num arquivo os agrupamentos.*

Exercício 18.5 *Gerar uma população e armazená-la num fato; depois para um K de 2 até 10 rodar o* `kmeans` *salvando os cluster e as avaliações; por fim, escolher qual é o melhor K.*

Exercício 18.6 *Generalizar: a)permitindo definir a métrica de similaridade (distância) num parâmetro; b) permitir dados discretos e contínuos; c)trabalhar com N atributos para cada indivíduo.*

Exercício 18.7 *Tratar automaticamente da normalização dos atributos.*

CAPÍTULO 19

IA: Classificação Naive Bayes

O tema Naive Bayes mostra como fazer inferências utilizando probabilidades condicionais entre eventos.

Antes de apresentar o algoritmo vamos mostrar como representar os dados como fatos a partir de listas de valores.

19.1 Os conjuntos de dados

Abaixo temos o programa que codifica a forma de tratar os dados. Ele inicia com dois conjuntos de dados, descritos por uma linha de cabeçalho que define o nome de cada coluna e depois por N linhas de dados.

```
1  %% ia-mkdata.pl
2   %% DATA SETS
3  pre_test_data(10,[
4    [outlook, temp, humidity, wind, playTennis],
5  [sunny, hot,   high,weak,  no],
6  [sunny, hot,   high,strong,  no],
7  [overcast,hot, high,weak,  yes],
8  [rain, mild,   high,weak,  yes],
9  [rain, cool,   nml,weak,  yes],
10 [rain, cool,   nml,strong,  no],
11 [overcast,cool,nml,strong,  yes],
12 [sunny, mild,  high,weak,  no],
13 [sunny, cool,  nml,weak, yes],
14 [rain,  mild,  nml,weak,  yes],
15 [sunny, mild,  nml,strong, yes],
16 [overcast,mild,high,strong, yes],
17 [overcast,hot,  nml,weak,  yes],
18 [rain,   mild,  high,strong, no]]).
19 %%
```

```
pre_test_data(12,[
[age, income ,student ,credit_rating  ,buys_computer],
[30,  high   ,no     ,fair       ,no],
[30,  high   ,no     ,excellent  ,no],
[35,  high   ,no     ,fair       ,yes],
[40,  medium ,no     ,fair       ,yes],
[40,  low    ,yes    ,fair       ,yes],
[40,  low    ,yes    ,excellent  ,no],
[35,  low    ,yes    ,excellent  ,yes],
[30,  medium ,no    ,fair        ,no],
[30,  low    ,yes    ,fair       ,yes],
[40,  medium ,yes    ,fair       ,yes],
[30,  medium ,yes    ,excellent  ,yes],
[35,  medium ,no     ,excellent  ,yes],
[35,  high   ,yes    ,fair       ,yes],
[40,  medium ,no    ,excellent   ,no]]).
%%-------------------------------------------------------------
%% MK DATA SETS
mkdata(_):- abolish(data/2),dynamic(dom/3),fail.
mkdata(_):- dynamic(data/2),dynamic(dom/3),fail.
mkdata(X):-pre_test_data(X,[Y|Ys]),!, mkDom(1,Y,DOM),
           zipper(DOM,D1,D2),assert(dom(X,DOM,D1)),mktuple(1, Ys).
mktuple(N,[X|Xs]):-assert(data(N,X)),N1 is N+1, mktuple(N1,Xs).
mktuple(_,[]):-listing(dom),listing(data).
mkDom(N,[X|Xs],[(N,X)|NXs]):-N1 is N+1, mkDom(N1,Xs,NXs).
mkDom(_,[],[]).
getD((Attr,V),(An,V)):- dom(_,VA,_),!, nth1(K,VA,(An,Attr)).
getDom([X|L],[DX|DL]):- getD(X,DX),getDom(L,DL).
getDom([],[]).
l:-listing(dom/3),listing(data/2).
getAttV((K,IDs)):- setof(V,Row^V^ID^(data(ID,Row),nth1(K,Row,V)),IDs).
getAttDom(L):- dom(N,_,Header),bagof((K,Vs),K^Vs^(member(K,Header),getAttV((K,Vs))),L).
zipper([(X,Y)|XYs],[X|Xs],[Y|Ys]):-zipper(XYs,Xs,Ys).
zipper([],[],[]).
```

Vamos trabalhar com fatos na base do Prolog, como mostrado na pergunta `mkdata(12)`, abaixo. Esta pergunta instancia o conjunto de dados 12 para ser processado. Ela cria dois fatos dinâmicos: um para as informações do domínio `dom/3` e outro para os dados em si, `data/2`. O `mkdata` chama o `mkDom` e o `mkTuple`. O `mkDom` simplesmente enumera a lista de colunas a partir do um, pois é mais fácil trabalhar com a numeração das colunas. Porém, caso for preciso trabalhar com nome da coluna é possível: o domínio permite o mapeamento entre número e nome; o `getDom/2` faz o mapeamento de atributo para número.

O `mkTuple` enumera as linhas a partir do um; as linhas dos conjuntos de dados não possuem um campo chave, logo, um conjunto de dados pode ter linhas duplicadas. Com a enumeração das linhas temos um valor id(entificador) de cada linha.

```
?- mkdata(12).
  :-dynamic dom/3.
   dom(12,[(1,age),(2,income),(3,student),(4,credit_rating),(5,buys_computer)],
   [1,2,3,4,5]).
  :- dynamic data/2.
```

```
    data(1, [30, high, no, fair, no]).
    data(2, [30, high, no, excellent, no]).
    data(3, [35, high, no, fair, yes]).
    data(4, [40, medium, no, fair, yes]).
    data(5, [40, low, yes, fair, yes]).
    data(6, [40, low, yes, excellent, no]).
    data(7, [35, low, yes, excellent, yes]).
    data(8, [30, medium, no, fair, no]).
    data(9, [30, low, yes, fair, yes]).
    data(10, [40, medium, yes, fair, yes]).
    data(11, [30, medium, yes, excellent, yes]).
    data(12, [35, medium, no, excellent, yes]).
    data(13, [35, high, yes, fair, yes]).
    data(14, [40, medium, no, excellent, no]).
?- mkDom(1, [aaa,bbb,ccc,2],X).
   X = [ (1, aaa), (2, bbb), (3, ccc), (4, 2)]
?- getDom( [(age,m30),(income, medium),(student,yes)],L).
   L = [ (1, m30), (2, medium), (3, yes)]
?- getAttV((2,Vs)).
   Vs = [high, low, medium]
?- getD((buys_computer,yes), T).
   T = 5, yes
?- getAttV((2,Vs).
   Vs = [high, low, medium]
?- getAttDom(L).
   L=[(1,[30,35,40]), (2,[high,low,medium]), (3,[no,yes]),
   (4,[excellent,fair]), (5,[no,yes])]
```

O predicado `getD` retorna o valor da coluna. O predicado `getAttV` retorna uma lista com todos os valores possíveis para uma determinada coluna. E, por fim, o predicado `getAttDom` que retorna para cada coluna todos os valores possíveis (o domínio de cada coluna).

19.2 A inferência Bayesiana

O método de classificação baseado em inferência Bayesiana trabalha com dados contínuos e discretos. Para dados contínuos, ele assume que os valores seguem uma função de distribuição normal, assim, as probabilidades são inferidas a partir da média e do desvio padrão de grupos de indivíduos. Para dados discretos os valores de probabilidades são coletados pela contagem nos grupos de indivíduos.

Seguem abaixo alguns testes para os predicados de probabilidade condicional para valores discretos e contínuos.

O predicado `probC_CC` é para valores contínuos. Dado que o valor da primeira coluna é 30: a probabilidade da classe (5ta coluna) ser `yes` é 0.00719275 e de ser `no` é 0.0347488.

O predicado `probC` é para valores discretos. Para os mesmos dados os valores são 0.222222 para `yes` e 0.6 para `no`; assim, em ambos casos a probabilidade de ser `no` é maior que a probabilidade de ser `yes`. O que muda é a escala de valores.

```
?- mkdata(12). ...
?- normal_df(110, 54.54, 120,X).
```

```
   X = 0.00719275
?- probC_CC([1,2,3,4,5,6,7,8,9,10,11,12,13,14],(1,30),(5,yes),P).
   P = 0.0347488
   probC_CC([1,2,3,4,5,6,7,8,9,10,11,12,13,14],(1,30),(5,no),P).
   P = 0.0583498
?- probC([1,2,3,4,5,6,7,8,9,10,11,12,13,14],(1,30),(5,yes),P), Po is P.
   P = 2/9  Po = 0.222222
?- probC([1,2,3,4,5,6,7,8,9,10,11,12,13,14],(1,30),(5,no),P), Po is P.
   P = 3/5   Po = 0.6
%%
?-probC([1,2,3,4,5,6,7,8,9,10,11,12,13,14],(1,40),(2,low),P), Po is P.
   P = 2/4  Po = 0.5
?- probC([1,2,3,4,5,6,7,8,9,10,11,12,13,14],(2,low),(1,40),P), Po is P.
   P = 2/5 Po = 0.4
```

Nas duas ultimas perguntas, acima, vemos que a **probabilidade condicional não é simétrica.**

O predicado principal `infer` é o que faz a inferência, que é a própria classificação. Para esse predicado de inferência, a solução apresentada aqui só trabalha com valores discretos. Para integrarmos com os valores contínuos devemos primeiro ter na descrição do domínio uma informação dizendo se ele é discreto ou contínuo; esta integração é deixada como exercício.

Assumindo que todos os dados são a base de treinamento, queremos saber a classificação para `[(age,30), (income,medium),(student,yes),(credit_rating,fair)]`. Isto é, o valor do atributo `buys_computer` é `yes` ou é `no`? O resultado retornado nas perguntas abaixo é 0.0282187 para `yes` e 0.00685714 para `no`; portanto, a classificação deste exemplo é `yes`. Propositadamente retornamos também os valores numa expressão simbólica sem calcular o valor.

```
?- infer([(age,30),(income, medium),(student,yes),(credit_rating,fair)],
                                        (buys_computer,yes),I), Io is I.

   I = 9/14* (1* (6/9)* (6/9)* (4/9)* (2/9))
   Io = 0.0282187
?- infer([(age,30),(income, medium),(student,yes),(credit_rating,fair)],
                                          (buys_computer,no),I), Io is I.
   I = 5/14* (1* (2/5)* (1/5)* (2/5)* (3/5))
   Io = 0.00685714
```

Segue abaixo o código dos predicados. Este programa chama o programa `estat.pl` para reutilizar os predicados `medv`, `std` e `normal_df`.

```
1  %% ia-bayes.pl
2  %% Inferencia Naive Bayes
3  :-[stat].
4  prob((K,V),P/Q):- bagof(ID,Row^(data(ID,Row),nth1(K,Row,V)),IDs),length(IDs,P),
5                    bagof(T,R^data(T,R),Ts),length(Ts,Q).
6  prob(L,(K,V),P/Q):- bagof(ID,Row^(data(ID,Row),nth1(K,Row,V),
7                      member(ID,L)),IDs),length(IDs,P),length(L,Q).
8  prob0(L/Lo,(K,V),P/Q):- bagof(ID,Row^(data(ID,Row),nth1(K,Row,V),
9                          member(ID,L)),Lo),length(Lo,P),length(L,Q).
10 probC(L,(K,V),(K1,V1),Po/Qo):- prob0(L/Lo,(K1,V1),_), prob(Lo,(K,V),Po/Qo).
11 probCond(L,A,B,P):-getD(A,Ao),getD(B,Bo),probC(L,Ao,Bo,P).
```

```
12 infer(L,C,I):-getDom(L,LN),getD(C,C1),infer0(LN,C1,I).
13 infer0(L,(A,V),Io):-bagof(T,R^data(T,R),IDs),
14                     prob0(IDs/IDo,(A,V),P/Q),inferL(IDo,L,I),Io = (P/Q)*I.
15 inferL(IDs,[L|Ls],Io):-prob(IDs,L,P/Q),inferL(IDs,Ls,I),Io = I*(P/Q).
16 inferL(_,[],1).
17 %%
18 getAttV(L,(K,Vs)):- bagof(V,Row^V^ID^(member(ID,L),data(ID,Row),nth1(K,Row,V)),Vs).
19 %% probabilidade condicional para coluna com valores contínuos
20 %% usa a densidade da função normal
21 probC_CC(L,(K,V),(K1,V1),P):- prob0(L/Lo,(K1,V1),_),getAttV(Lo,(K,Vs)),
22                               medv(Vs,M),std(Vs,Std),normal_df(M,Std,V,P).
```

Abaixo, mostramos a execução dos predicados auxiliares, usados para codificar o predicado de inferência, `infer`. O predicado `prob/2` retorna a probabilidade à priori de um par (coluna, valor); ele considera todos os exemplos da base; o `prob/3` faz o mesmo mas considerando só os exemplos coletados na lista passada como primeiro parâmetro. O predicado `prob0/3` é similar a este, porém retorna em `Lo` todos os identificadores das linhas da tabela que foram usados no denominador do cálculo da probabilidade; ele é usado no cálculo da probabilidade condicional.

Por fim, os predicados `probC/4` e `probCond/4` retornam a probabilidade condicional de uma coluna em relação à outra. No primeiro a coluna é passada como um número e no segundo pelo nome do atributo.

```
?- prob((5,yes),P).
   P = 9/14
?- prob((5,no),P).
   P = 5/14
?- prob([1,2,3,4,5,6,7],(5,yes),P).
   P = 4/7
?- prob0([1,2,3,4,5,6,7]/Lo,(5,yes),P).
   Lo = [3, 4, 5, 7]     P = 4/7
?- probC([1,2,3,4,5,6,7,8,9,10,11,12,13,14],(1,30),(5,yes),P), Po is P.
   P = 2/9   Po = 0.222222
?- probCond([1,2,3,4,5,6,7,8,9,10,11,12,13,14],
         (age,30),(buys_computer,yes),P), Po is P.
   P = 2/9  Po = 0.222222
```

Exercício 19.1 *Modifique o algoritmo para trabalhar com valores discretos e contínuos. Comece com a informação do tipo de dados para cada atributo. Use o formato do Weka: ARFF. Para um atributo numérico deve ser chamada o predicado da probabilidade condicional continua.*

Exercício 19.2 *A versão apresentada não consegue fazer inferência se um dos atributos tem probabilidade zero; pois, como é um produto, basta um fator ser zero para o resultado ser zero. Existem técnicas para contornar o problema, implemente uma delas.*

Dica: Uma da técnicas é a *m-estimativa*, que modifica a fórmula do cálculo de probabilidade de n_c/n para $(n_c + mp)/(n + m)$; na ausência de outra informação, pode-se assumir $p = 1/k$, se o atributo pode assumir k valores distintos; p é a estimativa de probabilidade que se deseja determinar; m é uma constante denominada *tamanho da amostra equivalente*: determina um peso dado a p em relação aos dados observados n_c; se m for zero, então a m-estimativa é n_c/n; m pode ser interpretada como um aumento de m exemplos virtuais distribuídos de acordo com p; logo para pequenas amostras (10 até 100 exemplos), faça $m = 2$.

Exercício 19.3 *Para bases de dados maiores, acima de 500 instâncias. Faça um comparativos dos métodos de classificação: KNN e Bayes. Rode primeiro as bases no Weka, para saber como fazer o comparativo. Utilize um método de testes sistemático como o N-fold cross validation.*

CAPÍTULO 20

IA: Similaridade via nGramas

Ngramas podem ser utilizados como uma medida de similaridade. Uni-gramas de palavras são utilizados nos engenhos de busca para recuperar documentos. Bi-gramas ou tri-gramas de caracteres servem para medir a similaridade de duas palavras (ou frases). Tri-gramas de palavras servem para medir o plágio entre dois textos.

Nos testes abaixo utilizamos bi e tri-gramas de letras para comparar duas palavras `eloi X eloy`. O número de tri-gramas iguais é 3 de um total de 6, resultando em um coeficiente de 0.5. Com bi-gramas o coeficiente vai para 0.6; e com uni-gramas vai para 0.75 (três letras iguais de 4). Aparentemente o valor de 0.75 é o mais correto, porém este valor também é obtido se invertermos a ordem das letras de um dos nomes: `eloi X yole`. Os bi-gramas e tri-gramas permitem medir similaridade considerando a ordem entre os elementos.

```
?-wr_ngr(char_tri,eloi,eloy).
    X:,6:[(32,32,e),(32,e,l),(e,l,o),(l,o,i),(o,i,32),(i,32,32)]
    Y:,6:[(32,32,e),(32,e,l),(e,l,o),(l,o,y),(o,y,32),(y,32,32)]
  X^Y:,3:[(32,32,e),(32,e,l),(e,l,o)]
  2*3/(6+6)=0.5
?-wr_ngr(char_big,eloi,eloy).
    X:,5:[(32,e),(e,l),(l,o),(o,i),(i,32)]
    Y:,5:[(32,e),(e,l),(l,o),(o,y),(y,32)]

  X^Y:,3:[(32,e),(e,l),(l,o)]
  2*3/(5+5)=0.6
?-wr_ngr(char_uni,eloi,eloy).
   X:,4:[e,l,o,i]
   Y:,4:[e,l,o,y]
 X^Y:,3:[e,l,o]
  2*3/(4+4)=0.75
?-wr_ngr(char_uni,eloi,yole).
    X:,4:[e,l,o,i]
    Y:,4:[y,o,l,e]
  X^Y:,3:[e,l,o]
```

```
2*3/(4+4)=0.75
```

Para gerar os tri-gramas, uma palavra é acrescida de dois brancos iniciais e dois finais (foi usado o valor Ascii do branco, 32). Para os bi-gramas foi acrescentado um branco inicial e um branco final. O propósito disso é favorecer a comparação de palavras pequenas (por exemplo, 3 letras), com prefixos ou sufixos iguais. Por exemplo, `elo X eli` se forem comparados como tri-gramas iguais o resultado é zero, porém com os dois brancos iniciais o resultado é 2/5. Para algumas aplicações, quando os prefixos e sufixos não tem um valor maior, deve-se remover estes brancos inicias para que sejam contabilizados os bi e tri-gramas puros.

Esta discussão vale também para comparar textos, nestes casos os ngramas são de palavras. Um método bom para identificar plágio entre dois textos é medir a quantidade de tri-gramas de palavras que eles tem em comum.

Seja `X^Y` a intersecção da lista `X` com a lista `Y` e seja `|X|` o comprimento da lista `X`. Definimos o coeficiente de *Dice* como `(2*|X^Y|) /(|X|+|Y|)`, e.g., `dice(aabbcc,abc) =(2*3)/9 =6,66`; e o coeficiente de *Sobreposição* como `|X^Y| /min(|X|, |Y|)`, e.g., `sobrep(aabbcc,abc) =3/3 =1`. Nesta definição de intersecção a ordem não é considerada, e.g., `dice(aabbcc,abc) =dice(abcabc,cba) =6,66`. Estas duas medidas são boas quando `X` e `Y` têm poucos elementos repetidos. Quando eles têm muitos elementos repetidos devemos considerar outras métricas, tais como, o coseno das freqüências dos elementos.

Segue abaixo o programa `ngram-07.pl` que codifica predicados para gerar ngramas (n=1,2,3) de caracteres e de palavras. Para palavras fizemos duas versões: uma considerando os tokens retornados pelo predicado `tokenize_atom` do SWI-Prolog e outra que usa o predicado `atom_to_stem_list`, também do SWI-Prolog, que remove os sufixos das palavras. Este *stemmer* (em-raizador) é para a língua inglesa, mas serve para ilustrar o uso dos ngramas. Na versão com *stemmer* são removidas as palavras de classe fechada também chamadas de *stop words*; para isso, elas foram cadastradas num fato.

Os predicados `get_chars` e `get_words` transformam uma string em letras ou palavras, respectivamente. Eles também removem os acentos e deixam tudo com letras minúsculas.

```
1  %% ngram07-elf.pl
2  char_uni(X,Y,Val):- ngram(char_uni,Y,X,Val).
3  char_big(X,Y,Val):- ngram(char_big,Y,X,Val).
4  char_tri(X,Y,Val):- ngram(char_tri,Y,X,Val).
5  word_uni(X,Y,Val):- ngram(word_uni,Y,X,Val).
6  word_big(X,Y,Val):- ngram(word_big,Y,X,Val).
7  word_tri(X,Y,Val):- ngram(word_tri,Y,X,Val).
8  word_uni_stem(X,Y,Val):- ngram(word_uni_stem,Y,X,Val).
9  word_big_stem(X,Y,Val):- ngram(word_big_stem,Y,X,Val).
10 ngramTypes( [word_uni, char_big, char_tri,
11              word_uni, word_big, word_tri,
12              word_uni_stem, word_big_stem, word_tri_stem]).
13 get_chars(F,Ns):-downcase_atom(F,Fd),unaccent_atom(Fd,Fu),atom_chars(Fu,Ns).
14 get_words(F,Ns):-downcase_atom(F,Fd),unaccent_atom(Fd,Fu),tokenize_atom(Fu,Ns).
15 char_uni(F,B):- get_chars(F, B).
16 char_big(F,B):- get_chars(F, Ns),!,mk_big(Ns, B).
17 char_tri(F,B):- get_chars(F, Ns),!,mk_tri(Ns, B).
18 word_uni(F,B):- get_words(F, B).
19 word_big(F,B):- get_words(F, Ns),!,mk_big(Ns, B).
20 word_tri(F,B):- get_words(F, Ns),!,mk_tri(Ns, B).
```

```
word_uni_stem(F,B):-atom_to_stem_list(F,Bo), filter_sw(Bo,B),  !. %%c\stemmer
word_big_stem(F,B):-atom_to_stem_list(F,Nso),filter_sw(Nso,Ns),!,mk_big(Ns,B).
word_tri_stem(F,B):-atom_to_stem_list(F,Nso),filter_sw(Nso,Ns),!,mk_tri(Ns,B).
%%
ngram(CALL,X,Y,Val):-CX=..[CALL,X,XB],CX,! ,CY=..[CALL,Y,YB],CY,!,
                      listDice(XB,YB,Val).
wr_ngr(CALL,X,Y):- CX=..[CALL,X,XB],CX, !,CY=..[CALL,Y,YB],CY,!,
           nl,writeln(CALL),
           interS(XB,YB,XYB),length(XB,Xm),length(YB,Ym),length(XYB,XYm),
           Val is 2*XYm/(Xm+Ym),
           FR=7,front(FR,XB,XBF), front(FR,YB,YBF), front(FR,XYB,XYBF),
           nl,write(('X:',Xm:XBF)),
           nl,write(('Y:',Ym:YBF)),
           nl,write(('X^Y:',XYm:XYBF)),
           nl,write(2*XYm/(Xm+Ym)=Val),nl.
%% bi-tri-gramas
mk_big(F,B):- !,big_([32|F], B).
mk_tri(F,B):- !,tri_([32,32|F], B).
big_([X,Y,Z|XYZs],[XY,YZ|Bs]):-!, XY=(X,Y),YZ =(Y,Z),big_([Z|XYZs],Bs).
big_([X,Y],       [XY,YB]   ):-!, XY=(X,Y),YB =(Y,32).
big_([X],         [XB]      ):-!, XB=(X,32).
big_([],          []        ).
tri_([X,Y,Z,W|XYZs],[XY,YZ|Bs]):-!, XY=(X,Y,Z),YZ =(Y,Z,W),tri_([Z,W|XYZs],Bs).
tri_([X,Y,Z],       [XY,YB,ZB]):-!, XY=(X,Y,Z),YB =(Y,Z,32), ZB=(Z,32,32).
tri_([X,Y],         [XB,YB]   ):-!, XB=(X,Y,32),YB=(Y,32,32).
tri_([X],           [XB]      ):-!, XB=(X,32,32).
tri_([],            []      ).
%% STOP WORDS
stop(1,'a o as os de do da em no um uns uma umas que te ele ela se por \c
 para pela ser nao com e cada ou desta dessa daquela daquele mesmo outra \c
 mas mais menos deve pode ja seja ter varios varias nenhum nenhuma outro').
stop(2,'; . , - = + - : ( ) [ ] \'').
:-abolish(stopw/1), dynamic(stopw).
mk_stop:-bagof(L,(LX^X^stop(X,LX),
                atom_to_stem_list(LX,L)),B),flatten(B,BF), mkw(BF).
mkw([X|Xs]):-assert(stopw(X)),mkw(Xs). mkw([]).
:-mk_stop.
%%
filter_sw([L|LS], OS):- stopw(L),!,filter_sw(LS,OS).
filter_sw([L|LS], [L|OS]):-      !,filter_sw(LS,OS).
filter_sw([],[]).
%% similaridade para ngramas: dice, sobrep, coseno
listSobrep(X,Y,D):-length(X,Xm),length(Y,Ym),
                    interS(X,Y,XY),length(XY,XYm), D is XYm/min(Xm,Ym).
listDice(X,Y,D):-length(X,Xm),length(Y,Ym),
                 interS(X,Y,XY),length(XY,XYm), D is 2*XYm/(Xm+Ym).
%%
interS(X,Y,XY):-msort(X,Xs),msort(Y,Ys),inter0(Xs,Ys,XY),!.
inter0([X|Xs],[X|Ys],[X|Is]):-     !,inter0(Xs,    Ys,Is).
inter0([X|Xs],[Y|Ys],   Is):- X@<Y,!,inter0(Xs,[Y|Ys],Is).
inter0([X|Xs],[Y|Ys],   Is):- X@>Y,!,inter0([X|Xs],Ys,Is).
```

```
inter0([],_,[]):-!.
inter0(_,[],[]).
%%
prodX([X|Xs], S):- prodX(Xs, Ss), S is (X*X)+Ss.
prodX([],0).
prodXY([X|Xs],[Y|Ys], S):- prodXY(Xs,Ys, Ss), S is (X*Y)+Ss.
prodXY([],[],0).
coseno0_fd(A,B,C):- bins(A,Ad), bins(B,Bd),merge0_fd(Ad,Bd,AB),
                    zipper(AB,_,BXY),zipper(BXY,Xo,Yo),coseno(Xo,Yo,C).
coseno_fd(A,B,C):- bins(A,Ad), bins(B,Bd),merge_fd(Ad,Bd,AB),
                   zipper(AB,_,BXY),zipper(BXY,Xo,Yo),coseno(Xo,Yo,C).
coseno(X,Y,P):- prodXY(X,Y,Sxy),prodX(X,Sx),prodX(Y,Sy),
                P is Sxy/(sqrt(Sx)*sqrt(Sy)).
%% bins+merge para calcular a distribuição da interseção
sortbins(L,Xo):- zipper(L,L1,L2),zipper(Lo,L2,L1),msort(Lo,Los),reverse(Los,Xo).
bins(L,Do):-msort(L,[X|LS]),!,freq(LS,X,1,Do).
freq([X|Xs],X,N,Ls):-!,N1 is N+1, freq(Xs,X,N1,Ls).
freq([X|Xs],Y,N,[(Y,N)|Ls]):-!,freq(Xs,X,1,Ls).
freq([],Y,N,[(Y,N)]).
%%
merge0_fd(X,Y,XY):-mrg0(X,Y,XY). %% assume ordem asc
mrg0([(X,Fx)|Xs],[(X,Fy)|Ys],[(X,Fx-Fy)|Is]):-     !,mrg0(Xs,        Ys,Is).
mrg0([(X,Fx)|Xs],[(Y,Fy)|Ys],[(X,Fx-0)|Is]):- X@<Y,!,mrg0(Xs,[(Y,Fy)|Ys],Is).
mrg0([(X,Fx)|Xs],[(Y,Fy)|Ys],[(Y,0-Fy)|Is]):- X@>Y,!,mrg0([(X,Fx)|Xs],Ys,Is).
mrg0([(X,Fx)|Xs],[],         [(X,Fx-0)|Is]):-      !,mrg0(Xs,         [],Is).
mrg0([],         [(Y,Fy)|Ys],[(Y,0-Fy)|Is]):-      !,mrg0([]         ,Ys,Is).
mrg0([],[],[]).
merge_fd(X,Y,XY):-mrg(X,Y,XY).  %% assume ordem asc
mrg([(X,Fx)|Xs],[(X,Fy)|Ys],[(X,Fx-Fy)|Is]):-  !,mrg(Xs, Ys,Is).
mrg([(X,Fx)|Xs],[(Y,Fy)|Ys],Is):- X@<Y,!,mrg(Xs,[(Y,Fy)|Ys],Is).
mrg([(X,Fx)|Xs],[(Y,Fy)|Ys],Is):- X@>Y,!,mrg([(X,Fx)|Xs],Ys,Is).
mrg([],_,[]):-!.
mrg(_,[],[]).
%% utils lists
zipper([(X-Y)|XYs],[X|Xs],[Y|Ys]):-!,zipper(XYs,Xs,Ys).
zipper([(X,Y)|XYs],[X|Xs],[Y|Ys]):-!,zipper(XYs,Xs,Ys).
zipper([],[],[]).
front(N,L,F):-length(F,N),append(F,_,L),!.  front(_,L,L).
top(N,L,T):-  msort(L,S),reverse(L,LR),front(N,LR,T).
```

O predicado `interS` verifica a intersecção de duas listas, retornando uma lista com todos os elementos que pertencem as duas listas. Para tornar o processamento mais eficiente, primeiro ordenam-se as duas listas sem remover os duplicados.

Com já foi apresentado, existem dois predicados para medir o coeficiente de similaridade baseado na intersecção: *Sobreposição* e *Dice*. A *Sobreposição* favorece a comparação de listas de comprimento diferente sem penalizar a menor lista. O *Dice* é mais apropriado para comparar listas de tamanhos similares; se as listas têm tamanhos desiguais este coeficiente é baixo.

```
?- interS([e,l,o,i],[l,i,o,y],X).
   X = [i, l, o]
?- listDice([e,l,o,i,l,u,i,z,f,a,v,e,r,o],[e,l,o,y],Y).
```

```
   Y = 0.333333
?- listSobrep([e,l,o,i,l,u,i,z,f,a,v,e,r,o],[e,l,o,y],Y).
   Y = 0.75
```

Para comparar cadeias com alta freqüência de elementos repetidos, sugerimos o uso do coseno. Neste caso, o predicado `bins` é usado para coletar a freqüência dos elementos. Similar ao `interS`, o predicado `merge_fd` coleta os valores das freqüências dos elementos das duas listas. É um pré-processamento para fazer o cálculo do coseno. Existem duas versões: uma ignora todos os pares que tem freqüência zero em um dos elementos e a outra não. O `merge0_fd` coleta todos os pares, mesmo os que têm zeros.

```
?- get_chars(aaaabbbbcccddd,A),bins(A,AB).
   A = [a, a, a, a, b, b, b, b, c|...]
   AB = [ (a, 4), (b, 4), (c, 3), (d, 3)]
?- get_chars(aaaabbbbcccddd,A),bins(A,AB),
   get_chars(aaabbeeefffccddd,B),bins(B,BB),merge_fd(AB,BB,MM).
   A = [a, a, a, a, b, b, b, b, c|...]
   AB = [ (a, 4), (b, 4), (c, 3), (d, 3)]
   B = [a, a, a, b, b, e, e, e, f|...]
   BB = [ (a, 3), (b, 2), (c, 2), (d, 3), (e, 3), (f, 3)]
   MM = [ (a, 4-3), (b, 4-2), (c, 3-2), (d, 3-3)]
?- get_chars(aaaabbbbcccddd,A),bins(A,AB),
   get_chars(aaabbeeefffccddd,B),bins(B,BB),merge0_fd(AB,BB,MM).
   A = [a, a, a, a, b, b, b, b, c|...]
   AB = [ (a, 4), (b, 4), (c, 3), (d, 3)]
   B = [a, a, a, b, b, e, e, e, f|...]
   BB = [ (a, 3), (b, 2), (c, 2), (d, 3), (e, 3), (f, 3)]
   MM = [ (a, 4-3), (b, 4-2), (c, 3-2), (d, 3-3), (e, 0-3), (f, 0-3)]
```

Podemos pensar numa busca de 5 palavras dentro de um texto com centenas de palavras; não tem sentido coletar centenas de pares com 0 de freqüência; neste caso usa-se o `merge_fd`. A saída do `merge_fd` é uma lista de triplas, `(token, frqX-frqY)`; por exemplo, `[(a, 5-4), (b, 4-2), (c, 3-2), (d, 3-3)]`. Usa-se o `zipper` para desfazer estas triplas em duas listas: os `Xs` e os `Ys`; eles são entrada da função coseno. Esta seqüência de passos é codificada em `conseno_fd`. De modo similar, para trabalhar também com zeros temos `coseno0_fd`.

Seguem abaixo alguns testes para o coseno. Nas duas primeiras perguntas, compara-se a versão com zeros com a versão sem zeros; nas duas últimas, compara-se o coseno com zeros com a métrica de Dice.

```
?-get_chars(aaaabbbbcccddd,A),get_chars(aaabbeeefffccddd,B),coseno0_fd(A,B,C).
  A = [a, a, a, a, b, b, b, b, c|...]
  B = [a, a, a, b, b, e, e, e, f|...]
  C = 0.746203
?- get_chars(aaaabbbbcccddd,A),get_chars(aaabbeeefffccddd,B),coseno_fd(A,B,C).
   A = [a, a, a, a, b, b, b, b, c|...]
   B = [a, a, a, b, b, e, e, e, f|...]
   C = 0.970725
?- char_tri(aaaabbbbcccddd,A),char_tri(aaabbeeefffccddd,B),coseno0_fd(A,B,C).
   A = [ (32, 32, a), (32, a, a), (a, a, a), (a, a, a), (b, b, b),  ...)|...]
   B = [ (32, 32, a), (32, a, a), (a, a, a), (a, a, b), (a, b, b),  ...)|...]
   C = 0.579751
```

```
?- ngram(char_tri, aaaabbbbcccddd, aaabbeeefffccddd,Z).
   Z = 0.588235
```

20.1 O programa de testes

O programa dos ngramas, em essência, codifica os predicados para gerar os ngramas e mais os três predicados de métricas. Diversos predicados utilizados nele já são nossos conhecidos tais como `coseno`, `prodX`, `prodXY`, `bins`, `freq`, `zipper`, `front`, etc. Estes predicados já conhecidos não serão relatados aqui.

Abaixo mostramos mais predicados usados para testar os predicados dos ngramas; alguns deles trabalham com arquivos. Inicialmente temos 5 frases como fatos. Depois temos o predicado `randCh/1` que gera um caractere randomicamente; a partir desse predicado codificou-se `randChs/2` que gera uma lista de caracteres randômicos de tamanho N. Depois temos os predicados `x4, x16, x32` que multiplicam um string Xvezes. Seguem testes ilustrativos para estes predicados.

```
?- frase(1,F1),frase(2,F2),ngram(char_tri,F1,F2,V12).
   F1 = 'em orientação a objetos o conceito de generalização'
   F2 = 'generalização é um dos conceitos de orientação a objetos'
   V12 = 0.720721
?- randCh(X).     X = w
?- randCh(X).     X = h
?- randChs(10,L). L = [k, o, f, q, z, u, q, w, y|...]
?- time((randChs(10000,L),bins(L,B))).
   % 59,990 inferences, 0.06 CPU in 0.09 seconds (66% CPU, 959840 Lips)
   L = [a, o, ... B = [ (a, 425), (b, 406), (c, 370), ...]
?- frase(1,F1),frase(2,F2),x16(F1,F1X),x16(F2,F2X),
   time(ngram(char_tri,F1X,F2X,V12)).
   % 4,629 inferences, 0.00 CPU in 0.00 seconds (?% CPU, Infinite Lips)
   F1X = "em ... F2X = ...
   V12 = 0.745921
```

Por fim, temos um conjunto de testes baseados em dados de arquivos. Assume-se que existem os arquivos `teste-big.txt`, `teste1.txt` e `teste2.txt`. São arquivos de texto; os dois últimos servem para medir a quantidade de plágio, sugerimos que eles compartilhem uma boa parte do texto.

```
1  %% testa predicados
2  frase(1,'em orientação a objetos o conceito de generalização').
3  frase(2,'generalização é um dos conceitos de orientação a objetos').
4  frase(3,'uma teoria de linguagens formais').
5  frase(4,'as linguagens de programacao').
6  frase(5,'as linguagens de programação').
7  randCh(CH):-C is 0'a+random(26),char_code(CH,C).
8  randChs(M,L):-findall(C,(between(1,M,_),randCh(C)),L).
9     x4(F,F4):-string_concat(F,F,F2),string_concat(F2,F2,F4).
10   x16(P,MP):- x4(P,P4),  x4(P4, MP).
11 x1616(P,MP):-x16(P,P16),x16(P16,MP).
12 %%
```

```
13 t1 :- frase(1,F1),frase(2,F2),frase(3,F3),frase(4,F4),frase(5,F5),
14     ngram(char_tri,F1,F2,V12),writeln(val:V12),
15     ngram(char_tri,F2,F3,V23),writeln(val:V23),
16     ngram(char_tri,F3,F4,V34),writeln(val:V34).
17 t3:- frase(1,F1),frase(2,F2), x16(F1,F1M),x16(F2,F2M), wr_ngr(char_tri,F1M,F2M).
18 t5:- frase(1,F1),frase(2,F2), x1616(F1,F1M),get_chars(F1M,F2W),bins(F2W,V),
19      length(F2W,L), writeln(len:K), writeln(V).
20 %% FILES
21 do1:-F1='teste1.txt', F2='teste2.txt', fl2str(F1,X),fl2str(F2,Y),
22      ngramTypes(L),forall(member(NGR,L),wr_ngr(NGR,X,Y)).
23 rd2tokens(F,L):-fl2str(F,S),!,downcase_atom(S,Fd),
24                                unaccent_atom(Fd,Fu),tokenize_atom(Fu,L).
25 fl2str(F,Str):- see(F),current_input(Stream),
26                 read_stream_to_codes(Stream, Codes),
27                 seen,atom_codes(Str,Codes).
28 plagio1(F1,F2):-fl2str(F1,X),fl2str(F2,Y),ngram(char_tri,X,Y,V),writeln(val:V).
29 plagio2(F1,F2):-fl2str(F1,X),fl2str(F2,Y),ngram(word_tri,X,Y,V),writeln(val:V).
30 x:-plagio1('teste1.txt','teste2.txt').
31 y:-plagio1('teste1.txt','teste1.txt').
32 %%
33 get_f0(T):-rd2tokens('text-big.txt',T).
34 get_f1(T):-rd2tokens('teste1.txt',T).
35 get_f2(T):-rd2tokens('teste2.txt',T).
36 set_f0(T):- tell('text-big.ddd'),wrtbins(T),told.
37 set_f1(T):- tell('teste1.ddd'),wrtbins(T),told.
38 set_f2(T):- tell('teste2.ddd'),wrtbins(T),told.
39 wrtbins([]) :- !.
40 wrtbins([D|Ds]):-!,D=(X-Y),writeq(X),writeq('|'),writeq(Y),nl,wrtbins(Ds).
41 %%
42 b0 :- get_f0(T), mk_big(T,T1), bins(T1,D), set_f0(D).
43 b1 :- get_f1(T), mk_big(T,T1), bins(T1,D), sortbins(D,Ds), set_f1(Ds).
44 b2 :- get_f2(T), mk_big(T,T1), bins(T1,D), sortbins(D,Ds), set_f2(Ds).
45 t1 :- get_f1(T), mk_tri(T,T1), bins(T1,D), sortbins(D,Ds), set_f1(Ds).
46 t2 :- get_f2(T), mk_tri(T,T1), bins(T1,D), sortbins(D,Ds), set_f2(Ds).
```

Exercício 20.1 *Dado um dicionário, considere uma lista com todas as palavras que iniciam com a letra (a); a partir da lista crie 10 palavras com uma letra duplicada: abóbora para* abbobora*; 10 sem uma letra:* abbora*; e 10 com duas letras consecutivas trocadas:* aobbora*. I) Agora gere uma tabela, num arquivo, comparando cada um destas palavras via métricas de uni, bi e tri-gramas com todas as palavras corretas. Liste na saída em ordem decrescente de similaridade. II) Faça o mesmo modificando os bi e tri-gramas, tirando os códigos 32 no início e fim de cada palavra. III) Examinando os valores procure fazer uma fórmula para descobrir cada um destes tipos de erros. Imagine que este processo de inferência é para implementar um corretor ortográfico.*

Exercício 20.2 *Utilizando uma ou mais métricas de ngramas, proponha um método para identificar plágio entre dois textos. Faça um experimento, com um conjunto de amostras.*

Exercício 20.3 *Utilizando uma ou mais métricas de ngramas, proponha um método para identificar plágio entre dois programas Java. Faça um experimento, com um conjunto de amostras.*

Exercício 20.4 *A partir de uma lista de palavras raízes (palavras fechadas, verbos no infinitivo (sem ar er ir) e nomes comuns no masculino singular) faça um programa stemmer baseado em métricas (1-2-3-gramas de caracteres). Reduza cada palavra de um texto para a palavra raiz mais próxima. Faça um estudo comparativo com o melhor stemmer para o Português. Teste situações como* `casamento -> cas(ar); casarão -> casa; canto -> cant(ar); canto -> canto`.

CAPÍTULO 21

IA: Teoria da Informação

Aqui apresentamos algumas métricas relacionadas com a teoria da informação. Elas são úteis para explorar a manipulação da informação. Por exemplo, para induzir um classificador a partir de uma lista de atributos estas métricas podem validar quais os atributos que influenciam mais na classificação. Assim, se temos 30 atributos podemos selecionar somente os 7 mais significativos para serem utilizados no desenvolvimento do classificador.

21.1 Bits e quantidade de informação

O **bit** é a unidade mínima de informação. Com um bit podemos codificar dois valores de uma moeda (1-Kara,0-coroa). Para codificar os valores na faixa de 0 até 15, totalizando 16 possibilidade, precisamos 4 bits, pois $log_2(16) = 4$. Assim, com n bits podemos codificar valores na faixa de 0 até $2^n - 1$.

Supondo, agora, uma moeda viciada, que gera a seqüência e lançamentos `KCCCKCCCCC`, isto é, gerou 2 Karas e 8 Coroas. Se queremos transmitir uma mensagem para esta moeda podemos pensar numa codificação simples de K=1 e C=0, resultando em 1000100000. Neste caso estamos utilizando 10 bits para os 10 lançamentos. Porém a **teoria da informação** diz que podemos transmitir esta mensagem de um modo mais econômico, com apenas 0.722 bits para cada lançamento, isto é, podemos transmitir os 10 lançamentos em menos de 7.22 bits; pois, $-(2/10)log_2(2/10) - (8/10)log_2(8/10) = 0.721928$.

Com é feita esta economia de bits? Primeiro, assumimos que a mensagem é longa, por exemplo, mil lançamentos. Segundo, utilizamos uma codificação mais inteligente: `CC=0`, `KC=10`, `CK=110`, `KK=1110`. Assim `KCCCKCCCCC = 1001000`; neste caso utilizamos apenas 7 bits. Se `K` é um valor pouco freqüente, a probabilidade dele ocorrer duas vezes seguido `KK` é bem baixa. Logo, o código maior `1110` do `KK` tem pouca probabilidade de acontecer; em mil lançamentos provavelmente aconteça. Aqui, a entropia diz quantos bits em média são necessários para transmitir um lançamento da moeda.

A idéia da economia depende de um código de tamanho variável onde os valores mais freqüentes recebem os códigos menores. Intuitivamente, a entropia aumenta com a variedade dos valores e também com uma distribuição equilibrada. Para uma moeda não viciada `K=5/10`, `C=5/10` a

entropia é `H([5/10, 5/10])=1` bit. Para uma moeda totalmente viciada `K=0/10, C=10/10` a entropia é `H([0, 1])=0` bit. Ainda, `H(KKKKKKKKKK) < H(KKKKCKKKKC) < H(KKKKKCCCCC); 0 < 0.72 < 1`. E com variedade: `H(KKKKKCCCCC) < H(ABCDABCDAB); 1 < 1.97095`; são 2 valores versus 4 valores.

Supondo X uma variável aleatória com m valores $v_1...v_m$ e seja $P(X = v)$ a probabilidade de X ter o valor v. Para $P(X = v_1) = p_1...P(X = v_m) = p_m)$ a **entropia** de X é:
$H(X) = -p_1 log_2(p_1)... - p_m log_2(p_m)$.

Supondo um dado não viciado, cada face tem `1/6` de probabilidade, `H([1/6, 1/6, 1/6, 1/6, 1/6, 1/6]) = 2.58496`. Significa que para transmitir um lançamento do dado precisamos de quase 2.6 bits. Suponha agora um objeto com três faces ABC, não viciado, a entropia é `H([1/3, 1/3, 1/3]) = 1.58496`. Como codificar os 10 lançamentos `ABC ACB CBAA` em menos de 16 bits. Uma codificação próxima da solução é `A=0 B=10 C=11`, que resulta em `01011 01101 111000`.

Em resumo, a entropia mede a quantidade de bits necessários para transmitir um evento. Para o número ser realista o evento deve ser repetido muitas vezes.

Neste capítulo codificamos alguns predicados para manipular métricas baseadas na entropia. Nos predicados calculamos $-p*log_2(p)$ como $-p*(log(p)/log(2))$; pois, $log_2(n) = (log(n)/log(2))$.

```
?- inf(8,8,X), Xo is X.
   X = - (8/ (8+8))*log(8/ (8+8))/log(2)-8/ (8+8)*log(8/ (8+8))/log(2)
   Xo = 1.0
?- inf(2,8,X), Xo is X.
   X = - (2/ (8+2))*log(2/ (8+2))/log(2)-8/ (8+2)*log(8/ (8+2))/log(2)
   Xo = 0.721928
?- inf(0,8,X), Xo is X.
   X = 0, Xo = 0
```

As perguntas acima exploram o cálculo da quantidade de informação para eventos binários `P`-positivos e `N`-negativos, como o lançamento de uma moeda. `P` e `N` são as freqüências de ocorrências. No cálculo, usam-se as probabilidades `P/(P+N)` e `N/(P+N)`. O resultado é mostrado simbolicamente como uma fórmula aritmética e também como um valor. Este modo de apresentar o cálculo em uma fórmula facilita o entendimento. Quando o leitor já o entendeu, o predicado pode ser modificado para devolver apenas o valor final.

Abaixo, a idéia da quantidade de informação é generalizada para uma variável aleatória representada numa lista de probabilidades cujo somatório é 1, e.g., `[6/14, 3/14, 5/14] = 1`. Aqui, *entropia é sinônimo de quantidade de informação*. Seguem abaixo vários testes.

```
?- inf([1/2,1/2],X), Xo is X.
   X = - (1/2)* (log(1/2)/log(2))+ (- (1/2)* (log(1/2)/log(2))+0)
   Xo = 1.0
?- inf([10/10, 0/10],X).    X = 0.0
?- inf([2/10,8/10],X), Xo is X.    Xo = 0.721928
?- inf([1/3,1/3,1/3],X), Xo is X.    Xo = 1.58496
?- inf([6/14,3/14,5/14],X), Xo is X.  Xo = 1.53062
```

Segue abaixo o programa principal com os predicados sobre teoria da informação.

```
1 %% ia-entropia.pl
2 %% Ganho de Informação & Entropia
3 inf([X|Xs],I):- X>0,!, inf(Xs,Is), I = -X*(log(X)/log(2))+Is.
4 inf([X|Xs],I):-     !, inf(Xs,Is), I = Is.
```

```
inf([],0).
inf(P,N,I):- FP=(P/(N+P)), FN=(N/(N+P)),FN>0,FP>0,!,
             I= -FP*log(FP)/log(2)-FN*log(FN)/log(2).
inf(P,N,0).
freqAttr(    K,LF):-bagof(V,ID^Row^(data(ID,Row),nth1(K,Row,V)),Vs),bins(Vs,LF).
freqAttr(IDs,K,LF):-bagof(V,ID^Row^(data(ID,Row),nth1(K,Row,V),member(ID,IDs)),Vs),
                    bins(Vs,LF).
freqAttr2(    K,L,LF):-bagof((V,Vl),ID^Row^(data(ID,Row),
      nth1(K,Row,V),nth1(L,Row,Vl)),Vs),bins(Vs,LF).
freqAttr2(IDs,K,L,LF):-bagof((V,Vl),ID^Row^(data(ID,Row),
      nth1(K,Row,V),nth1(L,Row,Vl),member(ID,IDs)),Vs),bins(Vs,LF).
entrXY0(IDs,K,L,Io):-freqAttr2(IDs, K,L,F),freqUm(F,LF),inf(LF,I), Io is I.
entrXY0(    K,L,Io):-freqAttr2(K,L,F),freqUm(F,LF),inf(LF,I), Io is I.
entrCondV(L,K,(K1,V1),Eo):- bagof(ID,Row^(data(ID,Row),nth1(K1,Row,V1),
                                    member(ID,L)),Lo), entrX(Lo,K,Eo).
entrX(IDs,K,I):-freqAttr(IDs, K,F),freqUm(F,LF),inf(LF,I).
entrX(    K,I):-freqAttr(K,F),freqUm(F,LF),inf(LF,I).
entrX0(IDs,K,Io):-freqAttr(IDs, K,F),freqUm(F,LF),inf(LF,I), Io is I.
entrX0(    K,Io):-freqAttr(K,F),freqUm(F,LF),inf(LF,I), Io is I.
%%  entropia cond, inf mutua & inf mutua relativa
iMutuaRel(L,X,Y,Io):-entrX(L,X,Ex), entrCond(L,X,Y,Eo), Io=(Ex-Eo)/Ex.
iMutua(L,X,Y,Io):- entrX(L,X,Ex), entrCond(L,X,Y,Eo), Io is Ex-Eo.
iMutuaRel(X,Y,Io):- entrX(X,Ex), entrCond(X,Y,Eo), Io=(Ex-Eo)/Ex.
iMutua(   X,Y,Io):- entrX(X,Ex), entrCond(X,Y,Eo), Io is Ex-Eo.
iMutua_(IDs,X,Y,Mo):-entrXY0(IDs,X,Y,XYo),  % outra forma para calcular
                     entrX0(IDs,X,Xo), entrX0(IDs,Y,Yo), Mo is Xo-XYo.
iMutua_(X,Y,Mo):-entrXY0(X,Y,XYo),entrX0(X,Xo), entrX0(Y,Yo), Mo is Xo-XYo.
entrCond(L,X,Y,Eo):- entrXY0(L,X,Y,Exy),entrX0(L,Y,Ey),Eo is Exy - Ey.
entrCond(  X,Y,Eo):- entrXY0(  X,Y,Exy),entrX0(  Y,Ey),Eo is Exy - Ey.
%% utils
getIDs(L):-setof(ID,Row^data(ID,Row),L).
divx(X,Y,Y/X).
freqUm(Ls,Lf):-zipper(Ls,As,Bs),sumlist(Bs,S),maplist(divx(S),Bs,Lf).
lids(K,Ls,IDs/IDso):- zipper(Ls,As,Bs), bagof((V,ID),Row^(data(ID,Row),
                      nth1(K,Row,V),member(ID,IDs),member(V,As)),IDx),
                      msort(IDx,[(A,I)|Dxo]),mergeX(Dxo,(A,[I]),IDso).
mergeX([(A,I)|AVs],(A,IDs),IDso):-!, mergeX(AVs,(A,[I|IDs]),IDso).
mergeX([(A,I)|AVs],(B,IDs),[(B,IDs)|IDso]):- mergeX(AVs,(A,[I]),IDso).
mergeX([],(A,IDs),[(A,IDs)]).
bins(L,Do):-msort(L,[X|LS]),!,freq(LS,X,1,Do).
freq([X|Xs],X,N,Ls):-!,N1 is N+1, freq(Xs,X,N1,Ls).
freq([X|Xs],Y,N,[(Y,N)|Ls]):-!,freq(Xs,X,1,Ls).
freq([],Y,N,[(Y,N)]).
zipper([(X-Y)|XYs],[X|Xs],[Y|Ys]):-!,zipper(XYs,Xs,Ys).
zipper([(X,Y)|XYs],[X|Xs],[Y|Ys]):-!,zipper(XYs,Xs,Ys).
zipper([],[],[]).
```

21.2 Entropia e Entropia conjunta

Sejam X e Y duas variáveis aleatórias. H(Y) é a **entropia** de X e H(X,Y) é a **entropia conjunta** das variáveis X e Y. H(X,Y) coleta a freqüência dos pares (x,y) e calcula a quantidade de informação necessária para transmitir cada par evento. A entropia conjunta é simétrica: H(X,Y)=H(Y,X).

Abaixo mostramos a execução do predicado freqAttr. Este predicado coleta as freqüências de atributo, e.g., F=[(comp,3), (hist,2), (math,4)]. Dada esta lista, o predicado freqUm coleta as probabilidades, e.g., FU=[3/9, 2/9, 4/9]. Esta lista de probabilidades é a entrada para a função inf que calcula a entropia. O predicado freqAttr2 é similar, porém, coleta os pares (x,y).

Os predicados entrX e entrXY implementam, respectivamente, H(X) e H(X,Y). Eles trabalham sobre uma representação de tabela tipo relacional, ver abaixo. Existem duas versões do predicado entrX: a primeira calcula a entropia da coluna toda e a segunda, com um parâmetro a mais, permite a escolha das linhas que devem ser consideradas no cálculo. Por exemplo, abaixo, numa das perguntas calculamos H(X) só para as linhas pares.

Existem também as versões do predicados entrX0 e entrXY0 que retornam apenas um valor numérico no lugar da expressão aritmética.

```
?- mkdata(0).
data(1, [math, yes]).
data(2, [hist, no]).
data(3, [comp, yes]).
data(4, [math, no]).
data(5, [math, no]).
data(6, [comp, yes]).
data(7, [hist, no]).
data(8, [math, yes]).
%%
?- freqAttr(2,F).      F = [ (no, 4), (yes, 4)]
?- freqAttr(2,F),freqUm(F,FU).   F = [ (no, 4), (yes, 4)]  FU = [4/8, 4/8]
?- freqAttr(1,F),freqUm(F,FU).
   F = [ (comp, 2), (hist, 2), (math, 4)] FU = [2/8, 2/8, 4/8]
?- freqAttr([2,4,6,8],1,F),freqUm(F,FU).
   F = [ (comp, 1), (hist, 1), (math, 2)]  FU = [1/4, 1/4, 2/4]
?- freqAttr2([2,4,6,8],1,2,F).
   F = [ ((comp, yes), 1), ((hist, no), 1), ((math, no), 1), ((math, yes), 1)]
?- ?- entrX(2,X), Xo is X.
   X = - (4/8)* (log(4/8)/log(2))+ (- (4/8)* (log(4/8)/log(2))+0)
   Xo = 1.0
?- entrX([2,4,6,8],1,X), Xo is X.
X=-(1/4)*(log(1/4)/log(2))+(-(1/4)*(log(1/4)/log(2))+(-2/4)*(log(2/4)/log(2))))
   Xo = 1.5
?- entrXY0(1,2,X). X = 2.0
?- entrXY0(2,1,X). X = 2.0
?- entrX0(1,X).  X = 1.5
?- entrX0(2,X).  X = 1.0
```

21.3 Entropia condicional e Informação mútua

Sejam `X` e `Y` duas variáveis aleatórias. `H(X|Y)` é a **Entropia condicional** e `I(X;Y)` é a **Informação mutua**. Em `H(X|Y)`, a entropia de `X` condicionada a `Y` *é o número de bits necessários para transmitir X se ambos os lados já conhecem Y*. Ela é calculada a partir da entropia conjunta: `H(X|Y) = H(X,Y)-H(Y)`.

A informação mútua, `I(X;Y)`, é sinônimo de **ganho de informação**: se já conhecemos X para codificar Y economizamos `I(X;Y)` de informação.

Ela também é simétrica, `I(X;Y)=I(Y;X)`. Ela pode ser calculada a partir da entropia condicional, `I(X;Y)= H(X)-H(X|Y)`, ou a partir da entropia conjunta, `I(Y;X)= H(X)+H(Y)-H(X,Y)`. Ainda, relacionada com a informação mútua temos a **informação mutua relativa**, `IR(X|Y)= (H(X)-H(X|Y))/H(X)`. Ela diz o percentual de `X` que é necessário transmitir se ambos já conhecem `Y`.

Um fato bem importante é que as medidas de *entropia condicional* e *informação mutua relativa* são assimétricas. Em IA temos muitas medidas simétricas e poucas assimétricas. Em muitos problemas precisamos de medidas assimétricas. A palavra *condicional* tem relação com uma implicação, com a noção de causalidade; o conceito de causalidade numa rede Bayesiana é direcional, assimétrico.

Por fim, temos também a entropia condicional **restrita a um valor** `H(X|Y=v)` que é a entropia da coluna `X` restrita as linhas em que `Y` tem o valor `v`.

Seguem abaixo os testes para estes predicados: `entrCondV`, `entrCond`, `iMutua` e `iMutuaRel`. Alguns testes destacam a assimetria da `iMutuaRel` e da `entrCond`.

```
?- mkdata(0).
?- entrCondV([1,2,3,4,5,6,7,8],2,(1,hist),P),Po is P.
   P = - (2/2)* (log(2/2)/log(2))+0
   Po = 0.0
?- entrCondV([1,2,3,4,5,6,7,8],2,(1,math),P), Po is P.
   P = - (2/4)* (log(2/4)/log(2))+ (- (2/4)* (log(2/4)/log(2))+0)
   Po = 1.0
?- entrCond([1,2,3,4,5,6,7,8],1,2,E). E = 1.0
?- entrXY0(1,2,X). X = 2.0
?- entrXY0(2,1,X). X = 2.0
?- entrX0(1,X).  X = 1.5
?- entrX0(2,X).  X = 1.0
?- entrCond(2,1,X).  X = 0.5
?- entrCond(1,2,X).  X = 1.0
?- iMutuaRel(2,1,M), Mo is M.  Mo  = 0.5
```

21.4 Escolha de atributos

Seguem abaixo três tabelas usadas nos testes do predicados codificados neste capítulo. Temos também algum testes que geram os valores de entropia condicional e informação mútua para todas as combinações de atributos de uma tabela.

```
1 pre_test_data(0,  [[xx,yy],
2    [math, yes],
3    [hist, no],
4    [comp, yes],
```

```
    [math, no],
    [math, no],
    [comp, yes],
    [hist, no],
    [math, yes] ] ).
%%
pre_test_data(10,[ [outlook, temp, humidity, wind, playTennis],
[sunny, hot,   high, weak,  no],
[sunny, hot,   high, strong,no],
[overcast,hot, high, weak,  yes],
[rain, mild,   high, weak,  yes],
[rain, cool,   nml,  weak,  yes],
[rain, cool,   nml,  strong, no],
[overcast,cool,nml,  strong, yes],
[sunny, mild,  high, weak,   no],
[sunny, cool,  nml,  weak,   yes],
[rain,  mild,  nml,  weak,   yes],
[sunny, mild,  nml,  strong, yes],
[overcast,mild,high, strong, yes],
[overcast,hot,  nml, weak,   yes],
[rain,   mild,  high, strong, no]]).
%%
pre_test_data(13,[ [cdcli, classe, sal, id, depto],
[a1,    high   ,10   ,a   ,no],
[a2,    high   ,10   ,b   ,no],
[a3,    high   ,10   ,c   ,no],
[a4,    med   ,20    ,d   ,yes],
[a5,    low    ,30   ,d   ,yes],
[a6,    low    ,30   ,f   ,yes],
[a7,    low    ,30   ,g   ,yes]]).
%%
all_e(DOM,(D1,D2,Eo)):-member(D1,DOM),member(D1,DOM), member(D2,DOM),
                       entrCond(D1,D2,E),Eo is E.
all_g(DOM,(D1,D2,Eo)):-member(D1,DOM),member(D1,DOM), member(D2,DOM),
                     D1=<D2,iMutua(D1,D2,E),Eo is E.
all_r(DOM,(D1,D2,Eo)):-member(D1,DOM),member(D1,DOM), member(D2,DOM),
                     iMutuaRel(D1,D2,E),Eo is E.
le:- writeln('EntrCond:'),dom(_,_,D),bagof(L,all_e(D,L),Ls),wTrip(Ls),nl,nl.
lm:- writeln('infMutua:'),dom(_,_,D),bagof(L,all_g(D,L),Ls),wTrip(Ls),nl,nl.
lr:- writeln('iMutuaRel:'),dom(_,_,D),bagof(L,all_r(D,L),Ls),wTrip(Ls),nl,nl.
wTrip([(X,Y,Z)|Xs]):-wTrip(X,[(X,Y,Z)|Xs]).
wTrip(X,[(X,Y,Z)|Xs]):-!,w3(X,Y,Z), wTrip(X,Xs).
wTrip(X,[(X1,Y,Z)|Xs]):-!,nl,wTrip(X1,[(X1,Y,Z)|Xs]).
wTrip(_,[]).
w3(X,Y,Z):-sformat(S,'~w ~w ~2f  ',[X,Y,Z]),writef(S).
```

Com estes predicados podemos comparar os atributos de uma tabela, todos contra todos. Estas medidas podem ser utilizadas na seleção dos melhores atributos para resolver um determinado problema de aprendizagem ou de mineração de dados. Elas permitem fazer escolhas de atributos para resolver duas situações: escolha dos melhores atributos para classificação e eliminação de atributos altamente correlacionados.

Escolha dos melhores atributos: por exemplo, numa tabela temos 30 atributos e queremos escolher os 7 melhores atributos para predizer um outro atributo (a classe). Um critério de escolha pode ser a informação mútua. Escolher os 7 atributos que tem a maior informação mutua com o atributo classe.

Um atributo é correlacionado ao outro se o valor de correlação é 1 ou (-1 inversamente proporcional). Para problemas de classificação, e.g., Naive Bayes, é bom saber se um atributo prediz totalmente o outro, ou se a correlação é 1. Neste caso, o classificador fica inflacionado: é como se o valor do atributo fosse elevado ao quadrado. A correlação é usada para dados contínuos. Já, para dados discretos podemos usar a informação mútua relativa ou a entropia condicional: i) se `IR(A|B)=1` então `B` prediz `A`; ii) se `H(A|B)=0` então `B` prediz `A` (tem toda a informação de).

Com estes critérios podemos afirmar que na tabela abaixo a coluna 1 prediz qualquer uma das outras 2,3,4,5. Por outro lado, a coluna 4 tem mais informação para predizer a coluna 1. As colunas 2 e 3 são equivalentes, uma prediz a outra.

```
?- mkdata(13).
data(1, [a1, high, 10, a, no]).
data(2, [a2, high, 10, b, no]).
data(3, [a3, high, 10, c, no]).
data(4, [a4, med, 20, d, yes]).
data(5, [a5, low, 30, d, yes]).
data(6, [a6, low, 30, f, yes]).
data(7, [a7, low, 30, g, yes]).
?- lm,le,lr.
infMutua:
1 1 2.81  1 2 1.45  1 3 1.45  1 4 2.52  1 5 0.99
2 2 1.45  2 3 1.45  2 4 1.16  2 5 0.99
3 3 1.45  3 4 1.16  3 5 0.99
4 4 2.52  4 5 0.99
5 5 0.99
EntrCond:
1 1 0.00  1 2 1.36  1 3 1.36  1 4 0.29  1 5 1.82
2 1 0.00  2 2 0.00  2 3 0.00  2 4 0.29  2 5 0.46
3 1 0.00  3 2 0.00  3 3 0.00  3 4 0.29  3 5 0.46
4 1 0.00  4 2 1.36  4 3 1.36  4 4 0.00  4 5 1.54
5 1 0.00  5 2 0.00  5 3 0.00  5 4 0.00  5 5 0.00
iMutuaRel:
1 1 1.00  1 2 0.52  1 3 0.52  1 4 0.90  1 5 0.35
2 1 1.00  2 2 1.00  2 3 1.00  2 4 0.80  2 5 0.68
3 1 1.00  3 2 1.00  3 3 1.00  3 4 0.80  3 5 0.68
4 1 1.00  4 2 0.46  4 3 0.46  4 4 1.00  4 5 0.39
5 1 1.00  5 2 1.00  5 3 1.00  5 4 1.00  5 5 1.00
Yes
```

Exercício 21.1 *Relacionado ao tema de normalização relacional em Banco de Dados, a informação mútua relativa pode ser utilizada para inferir as dependências funcionais. Lá em BD, é comum se fazer uma análise de todas as dependência funcionais considerando também as chaves compostas (colunas compostas). Crie um predicado que calcula a* `IR` *entre conjuntos de colunas, e.g.,* `iMutuaRelL([1,3],[2,5],X)`*. Utilizando este predicado liste todas as possíveis dependências funcionais de uma tabela dada. Não mostre as dependências funcionais triviais, entre sub-conjuntos de colunas.*

Exercício 21.2 *Utilizando uma técnica chamada binarização transforme os valores discretos das tabelas utilizadas nos testes em valores contínuos. Depois rode a correlação de todos com todos. Compare as medidas (informação mútua, informação mutua relativa e entropia condicional) contra a correlação. Qual destas medidas é mais equivalente a correlação?*

Dica: na binarização respeite a ordem crescente ou decrescente se existir entre os valores discretos, e.g., reg<bom<exc; além disso, não utilize todos os valores binários seqüencialmente, use valores assimétricos como segue:

```
math = 0b0001  % 1
hist = 0b0010  % 2
comp = 0b0100  % 4
lang = 0b1000  % 8
```

21.5 Indução de árvore para classificação: Id3

Com o uso dos predicados acima é fácil fazer uma versão simples do algoritmo de indução de regras via árvore de decisão chamado ID3.

A entrada do algoritmo é uma tabela e uma coluna chamada de classe. Por exemplo, na tabela abaixo consideramos a classe a coluna 5, playTennis. Supondo que queremos induzir um modelo que classificará um exemplo retornando playTennis=yes ou playTennis=no.

A saída do ID3 é uma árvore de decisão como aparece no final da execução abaixo. Todos os dados das 14 instâncias estão condensados naquela árvore de decisão que está representada nas 5 regras condicionais que seguem.

```
     if outlook=overcast             then playTennis=yes [4 inst]
else if outlook=rain and wind=strong then playTennis=no  [2 inst]
else if outlook=rain and wind=weak   then playTennis=yes [3 inst]
else if outlook=sunny and humidity=high then playTennis=no [3 inst]
else if outlook=sunny and humidity=nml  then playTennis=yes [2 inst]
```

O ID3 escolheu como primeiro atributo para agrupar as instâncias o atributo outlook, o ganho de informação em relação à classe é 0.24; ele é a raiz da árvore. São criadas três partições de dados: overcast, rain e sunny. Na primeira, todas as instâncias são playTennis=yes; na segunda, calcula-se novamente o ganho de informação de todos as colunas ainda não utilizadas; a melhor escolha é wind; criam-se as duas partições... No segundo nível novamente cada partição é entrada para o ID3. Este processo segue assim recursivamente. O processo termina quando todos as instâncias pertencem à mesma classe (entropia é 0) ou quando todos os atributos já foram utilizados; nesse caso existe uma taxa de erro nos dados de treinamento; tipicamente escolhe-se a moda da classe das instâncias que sobraram.

```
?- mkdata(10),  main_id3(5,T),wtree(1,T).
dom(10,[(1,outlook),(2,temp),(3,humidity),(4,wind),(5,playTennis)],[1,2,3,4,5]).
data(1, [sunny, hot, high, weak, no]).
data(2, [sunny, hot, high, strong, no]).
data(3, [overcast, hot, high, weak, yes]).
data(4, [rain, mild, high, weak, yes]).
data(5, [rain, cool, nml, weak, yes]).
data(6, [rain, cool, nml, strong, no]).
```

```
data(7, [overcast, cool, nml, strong, yes]).
data(8, [sunny, mild, high, weak, no]).
data(9, [sunny, cool, nml, weak, yes]).
data(10, [rain, mild, nml, weak, yes]).
data(11, [sunny, mild, nml, strong, yes]).
data(12, [overcast, mild, high, strong, yes]).
data(13, [overcast, hot, nml, weak, yes]).
data(14, [rain, mild, high, strong, no]).

@@@Part(melhorlast):[part(0.026,2),part(0.048,4),part(0.15,3),part(0.24,1)]
@@@Part(melhorlast):[part(0.019,2),part(0.019,3),part(0.97,4)]
@@@Part(melhorlast):[part(0.019,4),part(0.57,2),part(0.97,3)]
inf(1.57741/0.24675)
 #outlook =
      overcast #playTennis =todos(yes, ids([13, 12, 7, 3]))
      rain inf(0.970951/0.970951)
      #wind =
           strong #playTennis =todos(no, ids([14, 6]))
           weak #playTennis =todos(yes, ids([10, 5, 4]))
      sunny inf(0.970951/0.970951)
      #humidity =
           high #playTennis =todos(no, ids([8, 2, 1]))
           nml  #playTennis =todos(yes, ids([11, 9]))
T = [inf(1.57741/0.24675), attr:1, overcast:[attr:5, ...
```

Abaixo segue o programa `main_id3`. O primeiro argumento é o número da coluna classe e o segundo é árvore de decisão que é retornada.

O `main_id3` assume todas as instâncias numa partição inicial; ele remove a classe dos valores do domínio a serem utilizados na classificação; e por fim chama o `id3/4`, que recebe a lista de ids das instâncias, o domínio e a classe; retornando a árvore.

O `id3` testa a entropia da classe for zero termina o processo; senão ele testa se a lista de atributos é vazia então ele escolhe a moda; senão continua o processo recursivo: ele escolhe o melhor atributo para criar as novas partições e chama o `id3` para cada partição.

No final do código temos o predicado `wtree` que exibe a árvore.

```
1  main_id3(CLASS,Tree):-                          %% assume  data/2 e dom/2
2         setof(ID,Row^(data(ID,Row)),Lids),       %% tudo na 1ra partição
3         dom(_,_,Dom), select(CLASS,Dom,Dom1), %% retira CLASS de Dom
4         id3(Lids,Dom1,CLASS,Tree).               %% CLASS = coluna objetivo
5  id3([],_,_,[]).
6  %%
7  id3(L,Dom,CLASS,Tree):-
8       entrX0(L,CLASS,I),
9       I=0.0,!,Tree=[attr:CLASS,todos(Val,ids(L))],      %% fim 1
10         member(ID,L),data(ID,Row),nth1(CLASS,Row,Val);
11      Dom=[],!,freqAttr(L,CLASS,FV),getC(CLASS,M),
12                  Tree=[escolheModa(M,ids(L),fv(FV))];  %% fim 2
13      criaParts(L,Dom,CLASS,Part),                      %% senão particiona
14 writeln('@@@Part(melhor last)':Part),
15      last(Part,Mel),Mel=part(_,Mel1), select(Mel1,Dom,Dom1),!,
```

```
16      chamaPart(L, Dom1, Mel1,CLASS,Tree).
17  %%
18  chamaPart(L,Dom, Attr,CLASS,Tree):-
19      freqAttr(L,Attr,LF), entrX0(L,Attr,Ip), iMutua(L,Attr,CLASS,M),
20      lids(Attr,LF,L/Lo),
21      subPart(Dom, Lo,CLASS, Ts), Tree=[inf(Ip/M),attr:Attr|Ts].
22  subPart(Dom, [(Val,IDs)|VIDs], CLASS, Ts):-
23      id3(IDs,Dom, CLASS,Ts0),
24      subPart(Dom,VIDs,CLASS,Tss), Ts = [Val:Ts0|Tss].
25  subPart(_, [],_,[]).
26  criaParts(L,Dom,CLASS,Ps):- setof(part(I,A),(member(A,Dom),iMutua(L,A,CLASS,I)),Ps).
27  %%
28  getC(X,C):-dom(_,D,_),nth1(X,D,(V,C)).
29  wtree(N,[L|Ls]):-is_list(L),!,N1 is N+5,nl,wtree(N1,L),wtree(N,Ls).
30  wtree(N,[X:L|Ls]):-is_list(L),!,N1 is N+5,wtree(N1,X,L),wtree(N,Ls).
31  wtree(N,X,[L|Ls]):-nl,tab(N), wr(X),wnode(N,L),wtree(N,Ls).
32  wtree(N,  [L|Ls]):-wnode(N,L),wtree(N,Ls).
33  wtree(_,[]).
34  wnode(N,attr:X):-getC(X,C),!,write('#'),wr(C),write('=').
35  wnode(N,inf(X)):-wr(inf(X)),nl,tab(N).
36  wnode(N,X):-wr(X).
37  wr(X):- sformat(S, '~w~t~4|~w', [X, ' ']), write(S).
```

Exercício 21.3 *Faça as seguintes modificações no ID3: I) quando acontece escolha da moda pode-se contabilizar o percentual de erro; calcule-o mostrando na saída; II) implemente uma abordagem de testes; após induzir o modelo, utilize as regras contra os dados de testes e mostre numa matriz de confusão o percentual de erros e acertos.*

Referências Bibliográficas

[1] A. Aho, R. Seti. e J. Ulmman. Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading, MA, 1986. (ver versão traduzida)

[2] H. Ait-Kaci, *Warren's Abstract Machine: A Tutorial Reconstruction*, MIT Press, Cambridge, 1991, (also in the Web).

[3] I. Brakto, *Prolog Programming for Artificial Intelligence*, Second Edition, Addison-Wesley Publishing Company. 1990.

[4] M. A. Casanova, F. A. C. Giorno e A. L. Furtado, *Programação em Lógica e a Linguagem Prolog* Edgar Blücher Ltda, Rio de Janeiro, 1987.

[5] W. F. Clocksin e C. S. Mellish, *Programming in Prolog* Springer-Verlag, 4th edition, 1994.

[6] A. Comerauer, H. Hanoui, P. Roussel e R. Pasero. *"Un systeme de communication Homme-Machine en Français"*, Groupe d'Intelligence Artificielle, Université d'Aix-Marseille, France, 1973.

[7] M. A. Covington, D. Nute e A. Velino, *Prolog Programming in Depth*, Prentice Hall, New Jersey, 1997.

[8] M. A. Covington, *Natural Language Processing for Prolog Programmers*, Prentice Hall, New Jersey, 1994.

[9] C.J. Date, *Introdução as Sistemas de Bancos de Dados*, Campus, Rio de Janeiro, 1990.

[10] P. Deransart, A. Ed-Dbali e L. Cervoni, *Prolog: The Standard – Reference Manual* Springer, Berlin, 1996.

[11] R. Duncan, *Programação eficaz com Microsoft macro Assembler*, Rio de Janeiro : Campus, 1993

[12] R. A. Kowalski, The predicate calculus as a programming language, *International symposium and summer school on Mathematical Foundation of Computer Science* Jablona, Poland, 1972.

[13] C. J. Hooger, *Essentials of Logic Programming*, Oxford University Press, Oxford, 1990.

[14] S. J. Russell e P. Norvig, *Artificial Intelligence: A modern approach*, Prentice Hall, New Jersey, 1995.

[15] P. Van Roy, *1983-1993: The Wonder Years of Sequential Prolog Implementation*, Journal of Logic Programming, 1993.

[16] L. Sterling e E. Shapiro, *The Art of Prolog*, The MIT Press, Cambridge, 1986.

[17] D.H.D Warren, *WARPLAN: a system for generate plans*, Memo 76, Department of Artificial Intelligence, University of Edinburgh, Scotland, 1974.

[18] Ian H. Witten and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*, Second Edition, Morgan Kaufmann, 2005.

Índice Remissivo