Universidade Federal de Campina Grande - UFCG
Centro de Ciências e Tecnologia - CCT
Departamento de Sistemas e Computação - DSC

APOSTILA: INTRODUÇÃO À PROGRAMAÇÃO EM LÓGICA

Alexandre de Andrade Barbosa
aab@dsc.ufcg.edu.br

Joseluce de Farias Cunha
joseluce@dsc.ufcg.edu.br

34 páginas

2006

Av. Aprígio Veloso, 882 – Bodocongó
Caixa Postal 10.106
58.109-970 – Campina Grande – PB – Brasil
Fone: 310-1119 — Fax: 310-1273

APOSTILA: INTRODUÇÃO À PROGRAMAÇÃO EM LÓGICA

Alexandre de Andrade Barbosa
Departamento de Sistemas e Computação
E-mail: aab@dsc.ufcg.edu.br

Joseluce de Farias Cunha
Departamento de Sistemas e Computação
E-mail: joseluce@dsc.ufcg.edu.br

Av. Aprígio Veloso, 882 — Bodocongó — Caixa Postal 10.106
CEP 58109-970 — Campina Grande — PB — Brasil
Fone: 310-1119 — Fax: 310-1273 (DSC)

Resumo

Esta apostila foi criada como material de apoio à disciplina de Lógica Matemática do curso de Ciência da Computação da Universidade Federal de Campina Grande - UFCG. O conteúdo apresentado neste material é relacionado à Programação em Lógica, mais especificamente à linguagem Prolog. Nenhuma revisão sobre Lógica Matemática é apresentada, assim, é necessário que o leitor já possua conhecimento sobre Lógica Proposicional e Lógica de 1ª Ordem. Este texto possuí nível introdutório, uma vez que conhecimentos avançados sobre a linguagem não serão explorados na disciplina em questão.

Palavras-chave: Programação em lógica, Prolog, Lógica.

Sumário

1 Introdução **4**
1.1 O que é Programação em Lógica/Prolog? 4
1.2 Como Prolog funciona? 4

2 Implementações de Prolog **5**

3 Sintaxe SWI-Prolog **7**
3.1 Os comandos *write* e *read* 7
3.2 Comentários 8

4 Utilizando o SWI-Prolog **9**

5 Fatos, Regras e Consultas **10**
5.1 Regras 12
5.2 Regras recursivas 13
5.3 Como Prolog responde consultas 15

6 Listas **16**
6.1 Checagem de pertinência 17
6.2 Concatenação 17
6.3 Adicionando elementos 18
6.4 Excluindo elementos 18

7 Aritmética **19**

8 Corte de fluxo **21**

9 Exemplos **22**

10 Exercícios **27**

Lista de Figuras

1 Tela inicial do SWI-Prolog Editor . . . 9
2 Árvore genealógica . . . 10
3 Relação descendente. . . . 14
4 Árvore para a expressão $2 * a + b * c$ . . . 19
5 Jogo Torre de Hanoi . . . 24

1 Introdução

Esta apostila foi criada para apoiar a disciplina Lógica Matemática, em sua parte final relacionada à Programação em Lógica. O conteúdo apresentado aqui possui nível introdutório, uma vez que a disciplina em questão não fará uso de recursos avançados da linguagem. Estudos mais avançados sobre a Programação em Lógica deverão realizados em disciplinas posteriores.

O conteúdo deste texto busca fornecer aspectos mais práticos relacionados à linguagem Prolog, o funcionamento interno da linguagem não será detalhado neste material. Caso o leitor deseje se aprofundar no estudo da linguagem, recomenda-se a leitura de [1], [2], [3] e [4], nos quais este texto é baseado.

Espera-se que após a leitura deste material o leitor seja capaz de:

- compreender e executar programas Prolog;
- modificar programas Prolog;
- escrever programas Prolog básicos e intermediários.

1.1 O que é Programação em Lógica/Prolog?

Existem diversos paradigmas de programação, hoje, o paradigma mais utilizado é o paradigma procedural. As linguagens *Java*, *Pascal*, *C/C++* são ditas linguagens procedurais, pois, nestes programas é necessário implementar um procedimento para resolver determinado problema. Outro paradigma de programação é o paradigma imperativo, que possui a linguagem *LISP* como sua representante mais famosa. A Programação em Lógica faz parte do paradigma de programação denominado declarativo ou descritivo, neste, deve-se implementar uma descrição do problema e não um conjunto de instruções.

A linguagem Prolog é uma representante do paradigma declarativo, esta é a representante mais famosa da Programação em Lógica, a qual se baseia no cálculo de predicados. Prolog foi criada em 1972 por Colmerauer e Roussel, um programa Prolog não possui código para manipular a memória ou realizar desvios condicionais. Isso não significa que Prolog seja superior às outras linguagens, pode-se afirmar apenas que a linguagem é mais adequada para solucionar uma determinada categoria de problemas. Esta categoria diz respeito aos problemas onde é necessário representar algum tipo de conhecimento, por exemplo, em aplicações que realizem computação simbólica, na compreensão de linguagem natural ou em sistemas especialistas.

1.2 Como Prolog funciona?

Um programa Prolog constitui-se de uma coleção de fatos (base de dados) e regras (relações lógicas), esses ítens descrevem o domínio de um determinado problema. Esta descrição do problema é avaliada por um interpretador, o qual utilizando um "motor de inferência" realiza deduções em busca de conclusões válidas para consultas realizadas pelos usuários. Assim, pode-se afirmar que a computação destes programas é equivalente a prova de um teorema em lógica.

Os fatos de Prolog permitem a definição de predicados por meio da declaração de quais ítens pertencentes ao universo (ou domínio) satisfazem os predicados. Por exemplo, pode-se definir o predicado *homem(x)* e utilizar este para definir quais elementos do universo possuem tal predicado, no caso "x é homem". Vale salientar que é responsabilidade do programador manter a definição de um predicado consistente. Por exemplo, poderia ser criado o predicado *come(x,y)* para representar que "x come y" ou "y come x", o programador é que deverá especificar os fatos de maneira consistente.

As regras Prolog são descrições de predicados por meio de condicionais. Por exemplo, pode-se definir o predicado *pai(x)*, significando que " x é pai", através da regra: *pai(X) :- prole(X,_), homem(X)*. A regra significa que x é pai se x possui ao menos um filho.

O usuário interage com o programa através de consultas (*queries*). Por exemplo, sejam dados os fatos:

- homem(pedro).
- homem(joao).
- mulher(maria).
- mulher(teresa).

o usuário pode realizar a consulta *homem(X).* e receber as seguintes respostas *X=pedro; X=joao; No*, significando que pedro e joao são homens, o *No* indica que não existem mais respostas que satisfaçam a consulta.

As computações em Prolog utilizam os conceitos de cláusulas de horn, resolução e encadeamento para trás (*backtracking*), com estes é possível realizar a computação de maneira equivalente a uma dedução em Lógica de 1ª ordem.

Tanto fatos quanto regras são representados através de cláusulas de horn, ou seja, são fórmulas que contém predicados ou negação de predicados conectados por disjunções, onde ao menos um predicado não é uma negação. Quantificadores não são representados explicitamente, porém, a linguagem trata uma regra como se ela estivesse universalmente quantificada. Utilizando a regra de inferência da particularização universal repetidas vezes, é possível retirar os quantificadores e fazer com que uma variável assuma qualquer valor do domínio de representação.

Para realizar uma dedução Prolog utiliza unificação e a regra de inferência da resolução. Assim, duas cláusulas dão origem a um resolvente quando possuem predicados correspondentes, sendo um positivo (não negado) e outro negativo (negado). Quando o usuário realiza uma consulta o motor de inferência tenta resolver metas, sendo que uma meta pode conter submetas, quando não existem mais metas para serem satisfeitas em uma linha de resolução, o sistema utiliza encadeamento para trás (backtracking) em busca de outras respostas possíveis.

Ao longo desta apostila os conceitos citados serão exemplificados, no entanto, as explicações fornecidas sobre o funcionamento destes na linguagem Prolog não serão detalhadas.

2 Implementações de Prolog

Diversas implementações da linguagem Prolog podem ser encontradas, o que para muitos constitui um problema, pois estas nem sempre são totalmente compatíveis. Atualmente,

existem versões livres e comerciais criadas para os principais sistemas operacionais. Algumas das implementações mais conhecidas são:

- **Win-Prolog [5]** - respeita a sintaxe de Edinburgo, é a implementação mais próxima do dialeto puro. A versão 4.6, a mais recente, está disponível apenas para Windows (98, ME, NT, 2k, XP). É um sistema comercial, mas possui uma versão para testes gratuita.

- **Open Prolog [6]** - disponível unicamente para Macintosh (Mac OS 7.5 ou superior). Respeita a sintaxe padrão ISO, baseada na sintaxe de Edinburgo. É um software gratuito, desde que seu autor seja avisado de sua utilização.

- **Ciao Prolog [7]** - distribuído gratuitamente, possui licença *Library General Public License* (*LGPL*). Respeita a sintaxe padrão ISO, pode ser utilizada em diversos sistemas operacionais, tais como, Windows (98, NT, 2k, XP), Linux, SunOS, Solaris, MacOS X, entre outros.

- **YAP Prolog [8]** - criado pela Universidade do Porto em parceria com a Universidade Federal do Rio de Janeiro. Uma das principais características desta implementação é sua velocidade. Respeita a sintaxe padrão ISO, também é compatível com outras implementações de Prolog, tais como, Quintus e SICStus. A versão 5.1.1 pode ser utilizada em diversas distribuições Linux e Windows.

- **SWI Prolog [9]** - software gratuito, sob a licença *Lesser GNU Public License*. Pode ser utilizado nas plataformas Windows, Linux e MacOS. Possui diversas ferramentas de edição gráfica, tais como, *J-Prolog Editor* e *SWI-Prolog-Editor* (recomendado). Permite a utilização da linguagem Prolog por outras linguagens, tais como, C/C++ e Java. É uma das implementações mais utilizadas atualmente.

- **SICStus Prolog [10]** - é um software comercial, mas possui versão gratuita para avaliação. Respeita a sintaxe padrão ISO e pode ser usada nas plataformas Windows (2k e XP), Linux, Solaris 7, MacOS X e algumas distribuições Unix. Assim como o *SWI*, possui *interfaces* para comunicação com outras linguagens, por exemplo, C & C++, .NET, Java, Visual Basic e Tcl/Tk.

- **Amzi! Prolog [11]** - pode ser utilizado de maneira conjunta a *IDE* Eclipse, permite também comunicação com outras linguagens. Possui distribuições gratuitas e comerciais, que podem ser utilizadas em sistemas operacionais diversos, tais como, Windows, Linux, Solaris e HP/UX.

- **Visual Prolog [12]** - bastante diferente da versão padrão de Prolog, é fortemente tipado. Também conhecido como PDC Prolog ou Turbo Prolog, possui distribuições gratuitas e comerciais para Windows e Linux. Fornece um ambiente complto para programação, porém, é um dialeto mais complexo que o original.

Outras distribuições Prolog conhecidas são: GNU Prolog [13], XSB [14] e Trinc Prolog [15]. A implementação utilizada ao longo desta apostila é a *SWI-Prolog*, todo código utilizado nos exemplos e exercícios apresentados aqui foram desenvolvidos nesta versão de Prolog.

3 Sintaxe SWI-Prolog

Assim como todas as linguagens de programação, existem algumas regras que devem ser respeitadas pelo programador. Os dados representados em Prolog podem ser um dos seguintes tipos:

- variáveis - devem ser iniciados com letras maiúsculas ou *underscore* (_), seguidos de qualquer caractere alfanumérico. Caso uma variável seja definida apenas com *underscore*, ela será considerada uma variável anônima, ou seja, não se deseja saber seu valor. Ex.: X, Y1, _Nome, ...;
- não variáveis;
  - atômicas;
    - átomos - são constantes expressas através de palavras. Devem ser iniciados com letra minúscula seguida de qualquer caractere alfanumérico. Caso seja necessário definir um átomo com letra maiúscula ou número, deve se usar aspas simples. Ex.: joao, 'João', '16', 'Mary´s', ...;
    - inteiros - qualquer número que não contenha um ponto (.) será considerado um inteiro. Caracteres *ASCII* entre aspas duplas são considerados inteiros. Ex.: 1, 6, -3, "a" (interpretado como 97), ...;
    - números em ponto flutuante - qualquer número com um ponto e pelo menos uma casa decimal. Ex.: 5.3 (correto), 7.8 (correto), 7. (incorreto);
  - não atômicas;
    - listas - é uma seqüencia de elementos ordenados. Uma lista é declarada entre colchetes e os elementos devem ser separados por vírgula. Pode-se separar a cabeça (1o. elemento) do corpo (demais elementos) de uma lista utilizando |. Ex.: [a, b, c], [a | b, c], ...

3.1 Os comandos *write* e *read*

O comando *write* exibe o valor do parâmetro no dispositivo de saída corrente. O dispositivo padrão é o monitor, assim, o comando *write('Teste de impressão.').* irá exibir a mensagem *Teste de impressão.* na tela do monitor. O mesmo comando pode ser utilizado para imprimir o valor de qualquer variável.

No entanto, não existe um comando padrão Prolog para escrita de expressões formatadas. Devido a isso, o SWI-Prolog utiliza comandos de extensão, um deste é o comando *writef* do *C-Prolog* de Edinburgo. Este comando possui a seguinte sintaxe *writef(Formato, Argumentos).* Onde as opções para determinar a formatação são:

- %w - imprime o termo;
- %d - imprime o termo ignorando seu tipo, por exemplo, \n é impresso como uma *string.*
- %s - imprime o termo como uma string;

- %Nc - imprime o termo de modo centralizado numa quantidade N de colunas;
- %Nl - imprime o termo alinhado à esquerda numa quantidade N de colunas;
- %Nr - imprime o termo alinhado à direita numa quantidade N de colunas;

Para gerar alguns caracteres deve se usar seqüencias de escape, estas são:

- \n - cria uma nova linha;
- \l - criar um separador de linha, o resultado é igual ao produzido por \n;
- \r - retorna ao início da linha;
- \t - tabulação;
- \% - imprime o símbolo %;
- \nnn - onde n é um número decimal, produz o caractere *ASCII* com o código informado.

O comando *read* lê um valor no dispositivo de entrada corrente e unifica (atribui) o valor uma variável. O dispositivo de entrada padrão é o teclado, assim, o comando *read(X).* irá ler um valor do teclado e unificar este valor com a variável X.

3.2 Comentários

Assim como em outras linguagens de programação Prolog possui caracteres, ou seqüencias de caracteres, que identificam um trecho de comentários. Os comentários não são levados em consideração pelos interpretadores, porém, são importantes para que outras programadores possam compreender mais facilmente a codificação de um programa.

Em Prolog existem dois tipos de comentários, estes são identificados pelos símbolos "%" e "/* */". O símbolo % expressa que tudo aquilo que estiver entre ele e o final da linha deve ser tratado como comentário. Os símbolos /* */ indicam que tudo que estiver entre /* e */ será tratado como comentário, pode-se observar exemplos de comentários no trecho de código abaixo:

```
/* Descrição dos precidados homem e mulher sobre todos os
   elementos do universo de representação. */
    homem(pedro).    % representa o fato de que pedro é homem.
    homem(joao).     % representa o fato de que joao é homem.
    mulher(maria).   % representa o fato de que maria é mulher
    mulher(teresa).  % representa o fato de que teresa é mulher.
```

Considerações mais detalhadas sobre a sintaxe, os tipos de dados permitidos e sobre outros comandos podem ser encontrados no manual do SWI-Prolog, disponível em [16].

4 Utilizando o SWI-Prolog

Após realizar o *download* e a instalação do *SWI-Prolog* e do *SWI-Prolog-Editor*, inicie o editor, será apresentada a tela deste, a qual deve ser similar a exibida na Figura 1. Pode-se observar que a tela se divide em duas janelas editáveis, identificadas na imagem com os números 1 e 2. A área 1 corresponde a janela de edição do programa, a área 2 é a janela de interação do usuário com o interpretador. Para que se possa realizar qualquer interação com o programa que está sendo editado é necessário, após salvar o arquivo, clicar sobre botão de consulta, identificado com o número 3. Caso o arquivo seja alterado e o botão não seja acionado o interpretador irá trabalhar com a última versão do programa que foi consultada. O número 4 identifica o *prompt* de comandos, a partir deste é que podem ser enviados os comandos para o interpretador. Todo comando enviado para o interpretador deve obrigatoriamente ser finalizado pelo caractere ponto (.). Por exemplo, o comando ***?- write(Teste)*** não resultará em uma impressão, enquanto o comando ***?- write(Teste) .*** resultará na impressão da *string* 'Teste'. O símbolo **?-** será usado nos exemplos e exercícios apresentados para representar o *prompt* de comandos, ele não deve ser digitado.

As outras funcionalidades (e.g. salvar, abrir, debug, ...) existentes no sistema são similares ou idênticas à ações encontradas em diversos sistemas. Maiores detalhes de funcionamento do programa podem ser encontradas no manual do sistema.

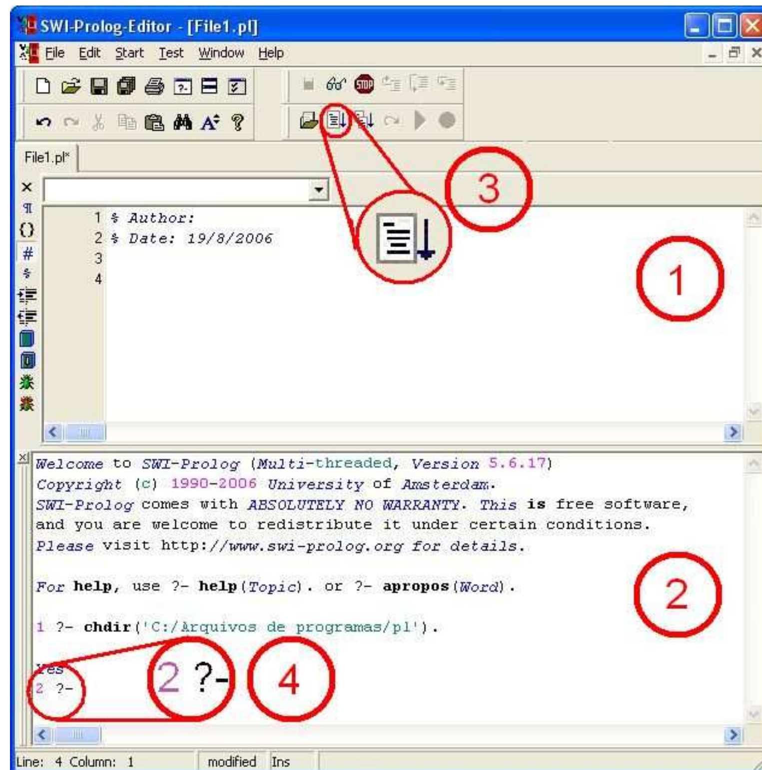

Figura 1: Tela inicial do SWI-Prolog Editor

5 Fatos, Regras e Consultas

Como já foi afirmado, um programa Prolog captura a descrição de um determinado problema. O programador deve implementar um conjunto de fatos e regras relacionados ao problema em questão. Ao longo de toda esta seção serão apresentados alguns fatos, regras e consultas Prolog que podem constituir um programa. Para facilitar a explicação destes conceitos será implementado um programa que descreva as relações de uma família. A Figura 2 exibe a árvore genealógica de uma família, nesta é possível observar que existem diversos tipos de relação, por exemplo, bob é pai de pat, pam é avó de ann, ann é irmã de pat, jim é filho de pat, entre outras.

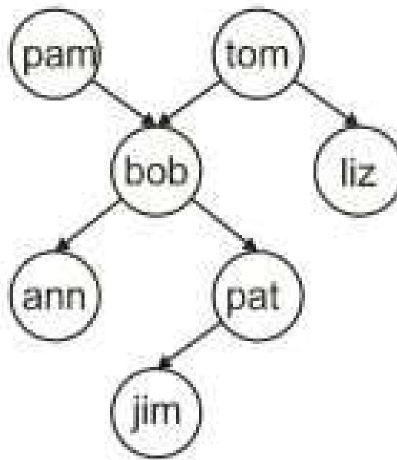

Figura 2: Árvore genealógica

Para se representar estas relações em Prolog, inicialmente pode-se criar a relação *genitor(x, y)*, significando que "x é genitor de y". Então, podemos inserir na janela de edição do programa os seguintes fatos:

```
genitor(pam, bob).
genitor(tom, bob).
genitor(tom, liz).
genitor(bob, ann).
genitor(bob, pat).
genitor(pat, jim).
```

Estas cláusulas descrevem toda a informação sobre a relação *genitor* existente na árvore apresentada (domínio). Após a definição desta relação podem ser realizadas consultas no sistema, para isso basta clicar no botão de consulta, e digitar perguntas no *prompt* de comandos, tal como exibido a seguir:

```
3 ?- genitor(pat, jim).
Yes
4 ?- genitor(jim, pat).
No
```

a consulta rotulada com o número 3, retorna a resposta *yes* (sim), isso ocorre pois o motor de inferência encontrar o fato correspondente e responder positivamente. Já para

a pergunta rotulada com o número 2, a resposta retornada é *No* (não), pois o programa não possui nenhum fato que indique que pat é genitora de jim.

As consultas apresentadas exemplificam o tipo mais simples de consulta, quando um fato da base já satisfaz a pergunta. Consultas mais interessantes são possíveis, por exemplo, pode-se indagar quem são os genitores de um determinado sujeito, ou quem são os filhos de deste na base de fatos do exemplo. Para isso, pode-se digitar no *prompt* de comandos as seguintes perguntas:

```
5 ?- genitor(X, bob).
X = pam ;
X = tom ;
No
6 ?- genitor(bob, X).
X = ann ;
X = pat ;
No
7 ?- genitor(X,Y).
X = pam
Y = bob ;
X = tom
Y = bob ;
X = tom
Y = liz ;
X = bob
Y = ann ;
X = bob
Y = pat ;
X = pat
Y = jim ;
No
```

a consulta rotulado com o número 5, indaga quem são os pais do indivíduo *bob*. Isso ocorre pois X é uma variável, assim, o motor de inferência realiza substituições sobre esta para encontrar quais fatos da base satisfazem a indagação. O mesmo ocorre na consulta rotulada com o número 6, porém, esta indaga quais são os filhos do indivíduo *bob*. A consulta rotulada com o número 7 exibe pares "genitor-filho", observe que o ordem das respostas é idêntica a ordem dos fatos inseridos na base.

Os exemplos exibidos até o momento ilustram consultas simples, é possível, por exemplo, realizar consultas sobre fatos que não estão diretamente descritos. Diga-se, por exemplo, que se deseja saber quem são os avós de determinado sujeito, apesar de não existir um fato que determine esta relação pode-se criar uma consulta que realize esta pergunta. Para isso, pode-se digitar no *prompt* de comandos a seguinte pergunta:

```
8 ?- genitor(Y, jim), genitor(X,Y).
Y = pat
X = bob ;
No
```

a consulta pode ser interpretada como "Y é genitor de jim e X é genitor de Y", ou seja, essa consulta busca quem é o pai do pai de um sujeito. Note que as cláusulas estão separadas por uma vírgula (,), para o *SWI-Prolog* a vírgula representa uma conjunção, enquanto um ponto e vírgula (;) representa uma disjunção. Então, uma seqüencia de

cláusulas separadas por vírgula só sera satisfeita se e somente se todas as cláusulas forem satisfeitas. Do mesmo modo, pode-se afirmar que uma seqüencia de cláusulas separadas por ponto e vírgula sera satisfeita se ao menos uma cláusula for satisfeita.

Um outro conectivo importante é a negação. Para exemplificar a utilização deste conectivo serão definidos os predicados *homem(x)* e *mulher(x)*, significando que "x é homem" e "x é mulher" respectivamente. Assim, são definidos os fatos:

```
mulher(pam).
homem(tom).
homem(bob).
mulher(liz).
mulher(pat).
mulher(ann).
homem(jim).
```

após definidos os fatos, podemos indagar, por exemplo, "quem é a mãe de bob", ou seja, deseja-se saber quem é o genitor de bob que é mulher. Para isso, pode-se digitar no *prompt* de comandos a seguinte pergunta:

```
9 ?- genitor(X, bob), mulher(X).
X = pam ;
No
```

a mesma consulta poderia ser realizada através da seguintes pergunta:

```
10 ?- genitor(X, bob), not(homem(X)).
X = pam ;
No
```

5.1 Regras

Muitas outras consultas podem ser realizadas sobre uma base de fatos, porém, é muito mais interessante utilizar regras, pois o poder de expressão obtido é muito maior. Para exemplificar o uso de regras será definida a relação *prole(y,x)*, significando que "y é prole de x". Esta relação é a relação inversa de *genitor(x,y)*, assim, pode-se afirmar que "y é prole de x se x é genitor de y". Para isso, pode-se criar a seguinte regra na janela de edição do programa:

```
prole(Y,X) :- genitor(X,Y).
```

o símbolo **:-** pode ser lido como **se**. A parte da regra a esquerda do símbolo **:-** é denominada de conclusão (ou cabeça), já a parte a direita deste é chamada de condição (ou corpo). Assim, para responder a consulta o interpretador Prolog precisa satisfazer parte condicional da regra, uma vez que não existem fatos relacionados a prole, para então obter uma conclusão. Para isso, são utilizadas substituições, até que se satisfaça a parte condicional ou não existam mais possibilidades de substituição.

Consultas são realizadas sobre regras da mesma maneira como se estas fossem fatos, por exemplo, para indagar quem é a prole do indivíduo tom deve-se digitar no *prompt* de comandos a seguinte consulta:

```
11 ?- prole(X,tom).
```

```
X = bob ;
X = liz ;
No
```

para satisfazer a regra o interpretador Prolog substituiu a variável X até que encontrou o fato *genitor(tom, bob)*, o qual corresponde a parte condicional da regra *prole(Y,X) :- genitor(X,Y)*, após as substituições adequadas. Posteriormente foi encontrado o fato *genitor(tom, liz)*, o qual também pode satisfazer a regra, nenhuma outra substituição resultou em sucesso.

Outras relações podem ser definidas para o exemplo. Pode-se definir as relações *mae(x,y)* e *avos(x,y)*, apresentadas anteriormente como consultas. Para isso, deve-se digitar na janela de edição do programa as seguintes regras:

```
mae(X,Y) :- genitor(X,Y), mulher(X).
avos(X,Z) :- genitor(X,Y), genitor(Y,Z).
```

Um outra relação que pode ser definida é a relação *irma(x,y)*, significando "x é irmã de y". Note que deve-se ter cuidado na definição deste relacionamento, pois, pode-se definir este através da seguinte regra:

```
irma(X,Y) :- genitor(Z,X), genitor(Z,Y), mulher(X).
```

porém, esta regra permite uma pessoa seja irmã de si mesma, para comprovar isso basta realizar a seguinte consulta:

```
12 ?- irma(pat, pat).
Yes
```

o programador deve estar atento a especificações deste tipo, para descrever corretamente a relação é necessário indicar que x e y precisam ser diferentes, assim a regra correta seria:

```
irma(X,Y) :- genitor(Z,X), genitor(Z,Y), mulher(X), not(X = Y).
```

Uma diferença básica entre uma regra e um fato é que um fato é sempre uma informação verdadeira, já uma regra precisa ser avaliada para que se possa determinar se esta é verdadeira ou não. Regras podem depender diretamente de um fato, como no exemplo anterior ou de outras regras (inclusive dela mesma). Regras definidas em termos de si mesma são chamadas de regras recursivas ou recorrentes, este tipo de regra será apresentado na seção seguinte.

5.2 Regras recursivas

A recursão é um dos elementos mais importantes da linguagem Prolog, este conceito permite a resolução de problemas significativamente complexos de maneira relativamente simples. A construção de uma regra recursiva será apresentada através da definição da relação *descendente(x,y)*, significando que "y é um descendente de x", tal como ilustrado na Figura 3. É possível definir esta relação utilizando a relação genitor, assim, uma descendência direta, ou seja, quando x é genitor de y, seria representada com a seguinte regra:

```
descendente(X,Z) :- genitor(X,Z).
```

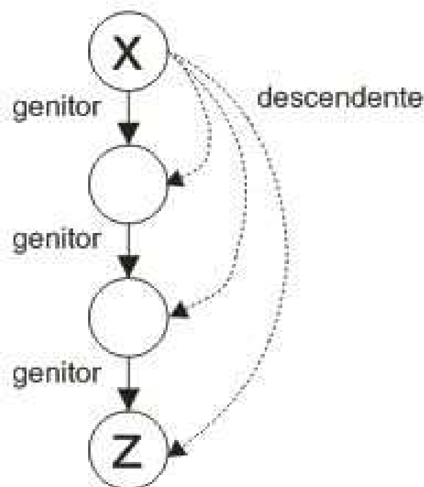

Figura 3: Relação descendente.

Para outros casos de descendência, que não uma descendência direta, poderiam ser utilizadas seguintes regras:

```
descendente(X,Z) :- genitor(X,Y), genitor(Y,Z).
descendente(X,Z) :- genitor(X,Y), genitor(Y,W), genitor(W,Z).
...
```

porém, esta solução seria limitada e trabalhosa. Usando recursão é possível obter uma solução bem mais simples e completa para a relação de descendência. Para isso, é necessário definir a seguinte afirmação "*x é um descendente de z se existe um y, tal que, x seja genitor de y e y seja um descendente de z*", a seguinte regra descreve isso:

```
descendente(X,Z) :- genitor(X,Y), descendente(Y,Z).
```

assim, utilizando as duas regras, pode-se descrever a relação de descendência de maneira correta. As duas regras são necessárias, pois o uso somente da primeira regra só seria suficiente para casos de descendência direta (equivalente à relação *genitor*), enquanto o uso exclusivo da segunda regra levaria a uma busca infinita de descendência. Então, a regra Prolog para relação de descendência incorpora as seguintes regras:

```
descendente(X,Z) :- genitor(X,Z).
descendente(X,Z) :- genitor(X,Y), descendente(Y,Z).
```

Pode-se então realizar consultas na base de exemplo sobre a relação de descendência, para isso, basta escrever a seguinte consulta:

```
13 ?- descendente(pam,X).
X = bob ;
X = ann ;
X = pat ;
X = jim ;
No
```

com isso são exibidos os nomes de todos os descendentes de pam.

O programa completo que descreve as relações familiares discutidas nesta seção pode ser observado a seguir:

```
/* Programa Prolog sobre relações familiares. */
genitor( pam, bob). % fato
genitor( tom, bob). % fato
genitor( tom, liz). % fato
genitor( bob, ann). % fato
genitor( bob, pat). % fato
genitor( pat, jim). % fato
mulher(pam). % fato
mulher(liz). % fato
mulher(pat). % fato
mulher(ann). % fato
homem(tom). % fato
homem(bob). % fato
homem(jim). % fato
prole(Y,X) :- genitor(X,Y). % regra
mae(X,Y) :- genitor(X,Y), mulher(X). % regra
avos(X,Z) :- genitor(X,Y), genitor(Y,Z). % regra
irma(X,Y) :- genitor(Z,X), genitor(Z,Y), mulher(X). % regra
descendente(X,Z) :- genitor(X,Z). % regra
descendente(X,Z) :- genitor(X,Y), descendente(Y,Z). % regra recursiva
```

Um conjunto de regras utilizados para descrever uma única relação é, em geral, chamada de procedimento (*procedure*). Assim, as regras relacionadas à descendência, ilustradas no exemplo, podem ser denominadas de procedimento.

5.3 Como Prolog responde consultas

Uma consulta Prolog é sempre um conjunto de metas. Quando uma pergunta é feita, Prolog precisa satisfazer todas as metas. Isso, como já afirmado, é equivalente a provar um teorema ou realizar uma dedução. Para isso, Prolog busca verificar se a(s) meta(s) são conseqüências lógicas dos fatos e regras contidos no programa. Para ilustrar o procedimento executado para responder uma consulta será utilizado o programa de exemplo, relacionado à relacionamentos familiares.

Diga-se que a consulta *?- descendente(tom, pat).* seja realizada. Sabe-se que *descendente(bob, pat)* é um fato existente no programa. Este fato seria derivado a partir da primeira regra relacionada a descendência. Além disso, sabe-se que a regra *genitor(tom, bob)* é um fato. Com isso, e o fato derivado *descendente(bob, pat)* pode-se concluir *descendente(tom, pat)* utilizando a segunda regra relacionada à descendência existente no programa. Isso ilustra o que foi utilizado pra realizar a prova, será apresentado agora como esta prova foi obtida.

Prolog encontra uma prova na ordem inversa a forma ilustrada anteriormente. Prolog inicia com uma meta e, utilizando regras, substitui estas metas por novas metas, até que uma meta seja satisfeita por um fato.

Assim, feita a pergunta *descendente(tom, pat)*, Prolog procede da seguinte maneira. Para satisfazer esta meta é procurada alguma cláusula (fato ou regra) da qual a meta possa ser deduzida. Então, são encontradas as duas regras relacionadas à descendência existentes no programa, pois, a cabeça da regra corresponde à meta. Como a regra *de-*

scendente(X,Z) :- genitor(X,Z). aparece primeiro, e como a meta atual é *descendente(tom, pat)*, as variáveis são substituídas, tal como a seguir:

X = tom, Z = pat

a meta *descendente(tom, pat)* é substituída pela meta *genitor(tom, pat)*. Como não existe uma cláusula onde a cabeça seja correspondente a esta, Prolog realiza um encaminhamento para trás, ou seja, retorna a meta original para tentar encontrar uma maneira alternativa de satisfazê-la.

Então, a regra *descendente(X,Z) :- genitor(X,Y), descendente(Y,Z).* será utilizada. Mais uma vez, as variáveis *X* e *Z* serão substituídas por *tom* e *pat* respectivamente, porém *Y* ainda não foi substituída. Assim, a meta atual dá lugar as metas *genitor(tom,Y), descendente(Y, pat)*.

Prolog deve agora satisfazer a conjunção de metas *genitor(tom,Y), descendente(Y, pat)*, isso é feito na ordem em que as metas estão escritas, ou seja, primeiramente Prolog irá tentar satisfazer *genitor(tom,Y)*. Realizando uma busca por cláusulas que satisfaçam a meta, Prolog encontra o fato *genitor(tom, bob)*, assim, *Y* é substituída por *bob*. Com isso, a meta atual é *descendente(bob, pat)*.

Para satisfazer esta meta a regra *descendente(X,Z) :- genitor(X,Z)* será usada novamente. Observe que esta é uma nova seqüencia de prova, sendo assim, as substituições anteriores não possuem nenhuma relação com esta. Com isso, Prolog usa um novo conjunto de variáveis, e assim, pode-se reescrever a regra como *descendente(X1,Z1) :- genitor(X1,Z1)*. Assim, como a cabeça deve corresponder meta, as seguintes substituições são realizadas:

X = bob, Z = pat

e então, a meta atual é trocada para nova meta *genitor(bob, pat)*. Como esta é satisfeita por um fato presente no programa o procedimento acaba.

6 Listas

Listas são um dos tipos de dados mais úteis existentes na linguagem Prolog, diz-se que uma lista é uma seqüência ordenada de uma quantidade qualquer de elementos. Os elementos de uma lista podem ser de qualquer tipo, tais como, números ou átomos.

Os elementos contidos em uma lista devem ser separados por vírgulas, e precisam estar entre colchetes. Por exemplo, uma lista pode conter os nomes dos indivíduos do exemplo da seção anterior, esta lista seria definida como:

[pam, liz, pat, ann, tom, bob, jim]

Existem dois tipos de listas, as listas vazias e as não vazias. Uma lista vazia é representada por [ ]. Listas não vazias podem ser divididas em duas partes, são elas:

- cabeça - corresponde ao primeiro elemento da lista;
- cauda - corresponde aos elementos restantes da lista.

Por exemplo, para a lista:

[pam, liz, pat, ann, tom, bob, jim]

pam é a cabeça, enquanto [liz, pat, ann, tom, bob, jim] é a cauda. Observe que a cauda é uma nova lista, que por sua vez também possui cabeça e cauda. Assim, pode-se dizer que o último elemento de uma lista possui uma cauda vazia (uma lista vazia). Pode-se especificar que um elemento de uma lista é também uma lista, assim, pode-se representar listas tais como:

Hobbies1 = [tenis, musica]. Hobbies2 = [sky, comida]. Lista = [ann, Hobbies1, tom, Hobbies2].

É possível separar as partes de uma lista utilizando uma barra vertical, assim, pode-se escrever *Lista = [cabeça | cauda]*. Com isso, é possível determinar as seguintes listas:

[a | b, c] = [a, b, c]

É possível realizar uma série de operações sobre listas, as seções seguintes exibem algumas destas ações.

6.1 Checagem de pertinência

Para se checar se um determinado elemento pertence à uma lista deve-se utilizar a relação *member(x,y)*, que indica se "x pertence à y, por exemplo:

```
3 ?- member(a, [a,b,c]).
Yes
4 ?- member(a, [[a,b],c]).
No
5 ?- member([a,b], [[a,b],c]).
Yes
```

na consulta rotulada com o número 3, *a* é um elemento da lista, uma vez que corresponde à cabeça desta. Já a consulta rotulada com o número 4, indica que *a* não pertence à lista, isso ocorre por que o elemento contido na lista é uma outra lista *[a,b]* e não o átomo *a*. Isso é ilustrado na consulta rotulada com o número 5.

6.2 Concatenação

Para realizar a concatenação de listas pode-se utilizar o predicado *append(L1,L2,L3).*, este predicado concatena a lista *L1* e *L2* exibindo o resultado em *L3*. O Mesmo predicado pode ser utilizado para decompor listas. Para definir a relação de concatenação, é necessário satisfazer as seguintes restrições:

- um argumento é uma lista vazia - caso algum argumento seja vazio a concatenação resultará na repetição do argumento não vazio;

- nenhum dos argumentos é vazio - a concatenação resulta na adição de todos os elementos da segunda lista ao final da primeira lista.

Visto isso, tem-se os exemplos:

conc([a,b], [], [a,b]) = true
conc([a,b], [c,d], [a,b,c,d]) = true

Para definir esta relação é pode-se implementar as seguintes regras:

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

Definidas as regras, pode-se realizar as seguintes consultas:

```
6 ?- conc([a,b], [c], L).
L = [a, b, c] ;
No
7 ?- conc([a], [b, c], L).
L = [a, b, c] ;
No
```

As regras definidas também podem ser usadas para decompor uma lista em suas componentes. Para checar isso, basta realizar a seguinte consulta:

```
6 ?- conc(L1, L2, [a,b,c]).
L1 = [] L2 = [a, b, c] ;
L1 = [a] L2 = [b, c] ;
L1 = [a, b] L2 = [c] ;
L1 = [a, b, c] L2 = [] ;
No
```

6.3 Adicionando elementos

A adição de um elemento à uma lista pode ser definida de modo simples. Para isso, basta inserir o elemento no início da lista, esta relação é definida através da seguinte regra:

```
insere(X, L, [X |L]).
```

com isso, pode-se realizar as seguintes inserções:

```
7 ?- insere(a, [b,c], L).
L = [a, b, c] ;
No
8 ?- insere([1,2], 3, L).
L = [[1, 2]|3] ;
No
```

6.4 Excluindo elementos

A exclusão de um elemento pode ser implementada através das seguintes regras:

```
exclui(X, [X | Tail], Tail).
exclui(X, [Y | Tail], [Y | Tail1]) :- exclui(X, Tail, Tail1).
```

a primeira regra é utilizada quando o elemento que se deseja excluir corresponde à cabeça da lista. Já a segunda regra exclui um elemento que pertence a cauda da lista. Vale

salientar que esta implementação não exclui todos os elementos existentes na lista que correspondam ao elemento passado como argumento.

Definidas as regras podem ser realizadas as seguintes consultas:

```
9 ?- exclui(a, [a,b,c], L).
L = [b, c] ;
No
10 ?- exclui(b, [a,b,c], L).
L = [a, c] ;
No
11 ?- exclui(c, [a,c,b,c], L).
L = [a, b, c] ;
L = [a, c, b] ;
No
```

Existem diversas outras operações nativas de Prolog, um exercício interessante seria realizar a implementação de algumas operações. Pode-se, por exemplo, criar uma operação para checar se uma lista está contida em outra ou uma operação para realizar permutação dos elementos.

7 Aritmética

Geralmente, quando se escreve uma expressão matemática a notação infixa é utilizada, por exemplo $2*a+b*c$, onde $2, a, b$ e c são argumentos e $+$ e $*$ são operadores. Em Prolog uma expressão é representada internamente como uma árvore, assim a expressão anterior seria representada pela árvore da Figura 4. Uma maneira de representar em Prolog a expressão em questão utiliza notação prefixa, a expressão seria representada como $+(*(2, a), *(b, c))$. Porém, por ser mais usual a representação infixa também é compreendida pela linguagem.

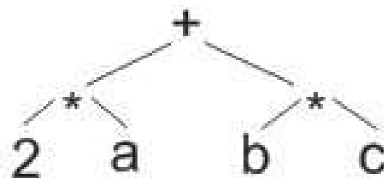

Figura 4: Árvore para a expressão $2 * a + b * c$

Prolog possui definidos operadores para as quatro operações: $+, -, *, /$, para realizar soma, subtração, multiplicação e divisão, respectivamente. Para se obter o resultado de uma operação é necessário utilizar o operador **is**, tal como ilustrado nas consultas abaixo:

```
3 ?- X is 2 + 3.
X = 5 ;
No
4 ?- X is 4 - 1.
X = 3 ;
No
5 ?- X is 2 * 5.
```

```
X = 10 ;
No
6 ?- X is 9 / 2.
X = 4.5 ;
No
```

Para o *SWI-Prolog* a operação / representa uma divisão real, para se obter uma divisão inteira deve-se usar o operador //. A precedência de operações aritméticas em Prolog é a mesma precedência adotada na matemática, assim, quando necessário devem ser utilizados parênteses para descrever uma expressão corretamente. Alguns dos operadores reconhecidos são:

- mod - para obter o resto da divisão;
- ^- para potenciação;
- cos - função cosseno;
- sin - função seno;
- tan - função tangente;
- exp - exponencial;
- ln - logaritmo natural;
- log - logaritmo;
- sqrt - raiz quadrada.

Existem também operações de conversão, algumas são automáticas outras precisam ser explicitamente solicitadas. Um exemplo de conversão automática ocorre quando um número inteiro é relacionado em uma expressão com números de ponto flutuante, automaticamente os inteiros são convertidos para números números de ponto flutuante. Algumas conversões explícitas nativas são:

- integer(X) - converte X para inteiro;
- float(X) - converte X para ponto flutuante.

Prolog também possui operações para comparação, os operadores são:

- > - maior que;
- < - menor que;
- >= - maior ou igual;
- =< - menor ou igual;
- =:= - igual;
- = / = - diferente.

É importante explicitar a diferença entre os operadores = e =:=, o primeiro operador verifica se dois objetos são iguais, enquanto o segundo verifica se o resultado da operação é igual. Isso fica mais claro através das consultas:

```
18 ?- 1 + 2 = 2 + 1.
No
19 ?- 1 + 2 =:= 2 + 1.
Yes
20 ?- 1 + A = B + 2.
A = 2
B = 1 ;
No
```

8 Corte de fluxo

A ordem das cláusulas em um programa e a ordem de definição de uma regra podem determinar o fluxo de execução de um programa. Esta seção apresenta um elemento para controle do fluxo denominado de corte, representado por "!".

A principal função do corte é melhorar a eficiência de um programa. Como já foi apresentado, Prolog utiliza encadeamento para trás sempre que necessário para satisfazer uma meta. Porém, muitas vezes, a utilização de encadeamento para trás causa uma busca desnecessária, levando a ineficiência. Para estes casos o uso do mecanismo de corte é extremamente útil.

Para apresentar o corte será utilizado o seguinte exemplo: *Sejam dados dois números X e Y, é desejado saber qual é o valor máximo destes.*

As seguintes regras descrevem a relação *maximo(x,y)*, significando que "x é o máximo valor se x é maior ou igual a y, y é o maior valor se y é maior que x":

```
maximo(X,Y,X) :- X >= Y.
maximo(X,Y,Y) :- X < Y.
```

estas regras computam a relação de maneira correta, porém, elas são exclusivas, ou seja, quando a primeira obtém sucesso a segunda irá falhar. Porém, Prolog sempre executa as duas regras, utilizando encadeamento para trás, o que para esta relação resulta apenas em ineficiência.

A mesma relação pode ser obtida, porém, sem gerar processamento ineficiente utilizando corte. Para isso, as regras seriam escritas como:

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

com isso, caso a primeira regra obtenha sucesso a segunda regra não será executada. A execução do programa com corte não altera o resultado deste, o corte apenas evita que sejam realizadas buscas desnecessárias.

No entanto, o uso do corte exige muito mais atenção do programador. Isso ocorre pois um programa sem cortes pode ter a ordem de suas cláusulas e regras modificadas sem alterar o significado do mesmo. Por sua vez, um programa que possua cortes pode ter seu significado alterado caso suas cláusulas sejam reordenadas. O seguinte exemplo ilustra estas afirmações. Sejam dadas as regras:

```
p :- a, b.
p :- c.
```

o significado lógico das regras pode ser interpretado pela fórmula: $p \leftrightarrow (a \wedge b) \vee c$. Com esta fórmula podemos modificar a ordem das cláusulas e seu significado não será alterado. Porém, caso seja utilizado corte, tal como nas seguintes regras:

```
p :- a, !, b.
p :- c.
```

o significado lógico das regras pode ser interpretado pela fórmula: $p \leftrightarrow (a \wedge b) \vee (\ a \wedge c)$. Com esta, caso a ordem das regras seja alterada para:

```
p :- c.
p :- a, !, b.
```

o significado lógico das regras pode será pela fórmula: $p \leftrightarrow c \vee (a \wedge b)$.

Assim, pode-se afirmar que o corte é um mecanismo útil, porém, deve ser utilizado com cuidado.

9 Exemplos

Uma das melhores formas de se aprender uma nova linguagem de programação é observando o código de programas. Esta seção apresenta alguns programas escritos em Prolog, o nível de complexidade destes programas será crescente, no que diz respeito aos conceitos utilizados. Uma boa atividade para o leitor seria realizar algumas modificações ou extensões sobre os programas apresentados.

Exemplo 9.1.

Descrição: O exemplo clássico para determinar que se todo homem é mortal e se Sócrates é um homem, então Sócrates é mortal. Essas afirmações podem ser representadas através das fórmulas:

$$\forall x(homem(x) \rightarrow mortal(x))$$
$$homem(socrates)$$

a partir destas pode-se concluir:

$$mortal(socrates)$$

O programa Prolog que descreve estas relações pode ser representado como a seguir.

Código:

```
mortal(X) :-                  % Todos os homens são mortais
  homem(X),
  writef('%w%w%w', ['Sim, ', X, ' é mortal']).
homem(socrates).              % Sócrates é um homem
```

Consulta:

```
?- mortal(socrates).
Sim, socrates é mortal
Yes
```

Exemplo 9.2.

Descrição: Dado um conjunto de animais determinar a cadeia alimentar de um animal qualquer.

O programa Prolog que descreve estas relações pode ser representado como a seguir.

Código:

```
animal(urso).
animal(peixe).
animal(peixinho).
animal(guaxinim).
animal(raposa).
animal(coelho).
animal(veado).
animal(lince).
planta(alga).
planta(grama).

come(urso, peixe).
come(peixe, peixinho).
come(peixinho, alga).
come(guaxinim, peixe).
come(urso, guaxinim).
come(urso, raposa).
come(raposa, coelho).
come(coelho, grama).
come(urso, veado).
come(veado, grama).
come(lince, veado).

cadeia-alimentar(X, Z) :-
  come(X, Z).

cadeia-alimentar(X, Z) :-
  come(X, Y),
  cadeia-alimentar(Y, Z).
```

Consulta:

```
?- cadeia-alimentar(urso, Y).
Y = peixe ;
Y = guaxinim ;
Y = raposa ;
Y = veado ;
Y = peixinho ;
Y = alga ;
Y = peixe ;
Y = peixinho ;
```

```
Y = alga ;
Y = coelho ;
Y = grama ;
Y = grama ;
No
```

A repetição de elementos ocorre, pois, um urso come um animal z *diretamente e come algum animal* y *que come* z. *Então, a cada vez que um animal aparece na cadeia alimentar de um urso ele será exibido.*

Exemplo 9.3.

Descrição: Implementar um código para solucionar o jogo Torre de Hanoi, com n *peças. O jogo é formado por uma base contendo três pinos, em um destes pinos estão dispostos sete discos uns sobre os outros, em ordem crescente de diâmetro, de cima para baixo, tal como ilustrado na Figura 5(a). O problema consiste em passar todos os discos de um pino para outro qualquer, usando um dos pinos como auxiliar, de maneira que um disco maior nunca fique em cima de outro menor em nenhuma situação. O número de discos pode variar sendo que o mais simples contém apenas três. A Figura 5(a) ilustra um possível estado inicial e a Figura 5(b) ilustra um possível estado final para o estado inicial exibido.*

(a) Estado inicial para o jogo torre de hanoi. (b) Estado final para o jogo torre de hanoi.

Figura 5: Jogo Torre de Hanoi

Um programa Prolog que fornece soluções para o jogo descrito pode ser implementado como a seguir.

Código:

```
hanoi(N) :-
  move(N, esquerdo, central, direito).
move(0, _, _, _) :- !.
move(N, A, B, C) :-
  M is N-1,
  move(M, A, C, B),
  inform(A, B),
  move(M, C, B, A).
inform(X, Y) :-
  writef('%w%w%w%w', ['Mova o disco do pino ', X, ' para o pino ', Y]),
  nl.
```

Consulta:

```
?- hanoi(3).
Mova o disco do pino esquerdo para o pino central
```

```
Mova o disco do pino esquerdo para o pino direito
Mova o disco do pino central para o pino direito
Mova o disco do pino esquerdo para o pino central
Mova o disco do pino direito para o pino esquerdo
Mova o disco do pino direito para o pino central
Mova o disco do pino esquerdo para o pino central
Yes
```

Para visualizar a solução procure uma versão do jogo na internet, porém, o código implementado considera que o estado inicial do jogo contém todos os discos no pino da esquerda, enquanto alguns versões disponíveis na internet aceitam outras configurações.

Exemplo 9.4.

Descrição: O bubble sort*, é o mais simples algoritmo de ordenação. O algoritmo consiste em percorrer um vetor várias vezes, em cada iteração o menor ou o maior elemento é colocado em sua posição correta no vetor.*

O programa Prolog que implementa este algoritmo é o exibido a seguir.

Código:

```
bubblesort(Lista_In, Lista_Out) :-
  append(L_Front, [A, B|Rest], Lista_In), % Lista_In = [A + B|Rest]
  B < A,!,                                % checa se B é menor que A
  append(L_Front, [B, A|Rest], L_Rest),   % L_Rest = [B + A|Rest]
  bubblesort(L_Rest, Lista_Out).          % utiliza e regra recursivamente
bubblesort(Lista, Lista).                 % Lista já está ordenada
```

Consulta:

```
?- bubblesort([2,3,1,7,5,4], L).
L = [1, 2, 3, 4, 5, 7] ;
No
```

Exemplo 9.5.

Descrição: Dada uma lista de elementos, determine se ela é um palíndromo ou não. Um palíndromo é uma palavra ou número cuja leitura é a mesma, quer se faça da esquerda para a direita, quer se faça da direita para a esquerda;

Código:

```
palindrome([]).
palindrome([_]).
palindrome([F|R]) :-
  append(S,[F],R),
  palindrome(S).
```

Consulta:

```
?- palindrome([m,a,m]).
Yes
?- palindrome([m,a,a]).
```

```
No
?- palindrome([s,o,c,o,r,r,a,m,m,e,e,m,m,a,r,r,o,c,o,s]).
Yes
```

Exemplo 9.6.

Descrição: Máximo Divisor Comum (M.D.C.). Dados dois inteiros positivos A *e* B*, o eu máximo divisor comum,* C*, é o maior número que divide* A *e* B *sem deixar resto. Para encontrar o m.d.c de dois números é necessário trabalhar com três casos, são eles:*

- *se A=B, então, C=A ou B;*
- *se A < B, então, C é igual ao maior divisor comum de A e B - A;*
- *se A > B, então, C é igual ao maior divisor comum de B e A - B.*

Um programa Prolog que que implementa m.d.c pode ser representado como a seguir.
Código:

```
mdc(A, A, A). mdc(A, B, C) :-
   A < B,
   Temp is B - A,
   mdc(A, Temp, C).
mdc(A, B, C) :-
   A > B,
   Temp is A - B,
   mdc(Temp, B, C).
```

Consulta:

```
?- mdc(12, 18, C).
C = 6 ;
No
?- mdc(4,4,C).
C = 4 ;
No
```

Exemplo 9.7.

Descrição: Cálculo de fatorial. Implementar um programa Prolog que realize o cálculo do fatorial de um número.
Código:

```
fatorial(0,1).
fatorial(N,F) :-
    N>0,
    N1 is N-1,
    factorial(N1,F1),
    F is N * F1.
```

Consulta:

```
?- fatorial(3,W).
W = 6 ;
No
?- fatorial(4, X).
X = 24 ;
No
```

Exemplo 9.8.

Descrição: Implementar um programa que determina se um determinado dia faz parte de um dia da semana ou final de semana. Note que a determinação de uma categoria já exclui a possibilidade de que o elemento pertença a outra categoria, ou seja, não existe um dia que seja semana e final de semana.

Código:

```
semana(segunda).
semana(terca).
semana(quarta).
semana(quinta).
semana(sexta).
fimdesemana(sabado).
fimdesemana(domingo).

categoria(X) :-
  fimdesemana(X),
  writef('%w%w', [X, ' é um final de semana.']),
  !.
categoria(X) :-
  semana(X),
  writef('%w%w', [X, ' é um dia de semana.']),
  !.
```

Consulta:

```
?- categoria(segunda).
segunda é um dia de semana.
Yes.
?- categoria(domingo).
domingo é um final de semana.
Yes.
```

10 Exercícios

As questões 10.1 até 10.4 são relacionadas ao programa Prolog apresentado a seguir:

```
genitor( pam, bob).
genitor( tom, bob).
genitor( tom, liz).
```

```
genitor( bob, ann).
genitor( bob, pat).
genitor( pat, jim).
mulher(pam).
mulher(liz).
mulher(pat).
mulher(ann).
homem(tom).
homem(bob).
homem(jim).
prole(Y,X) :- genitor(X,Y).
mae(X,Y) :- genitor(X,Y), mulher(X).
avos(X,Z) :- genitor(X,Y), genitor(Y,Z).
irma(X,Y) :- genitor(Z,X), genitor(Z,Y), mulher(X), not(X = Y).
descendente(X,Z) :- genitor(X,Z).
descendente(X,Z) :- genitor(X,Y), descendente(Y,Z).
```

Exercício 10.1. *Quais as respostas para as seguintes consultas?*

1. *?- genitor(X, jim).*
2. *?- genitor(jim, X).*
3. *?- genitor(pam, X), genitor(X, pat).*
4. *?- genitor(pam, X), genitor(X, Y), genitor(Y, jim).*

Exercício 10.2. *Formule consultas para descobrir:*

1. *Quem são os pais de Pat?*
2. *Liz possui filhos?*
3. *Quem é o avô de Pat?*
4. *Quem é a mãe de Jim?*
5. *Quem é o irmão de Bob?*
6. *Quem é a irmã de Pat?*

Exercício 10.3. *Formule regras para as seguintes relações:*

1. *tio(a);*
2. *irmão;*
3. *avós paternos;*
4. *avós maternos;*
5. *ascendente (o inverso de descendente);*
6. *primo(a), insira novos fatos para realizar consultas sobre esta relação.*

Exercício 10.4. *Em quais das seguintes consultas Prolog utiliza encadeamento para trás?*

1. *?- genitor(pam, bob).*

2. *?- mae(pam, bob).*

3. *?- avos(pam, ann).*

4. *?- avos(bob, jim).*

Exercício 10.5. *Considere as seguintes premissas:*

Todos os animais têm pele. Peixe é um tipo de animal, pássaros são outro tipo e mamíferos são um terceiro tipo. Normalmente, os peixes têm nadadeiras e podem nadar, enquanto os pássaros têm asas e podem voar. Se por um lado os pássaros e os peixes põem ovos, os mamíferos não põem. Embora tubarões sejam peixes, eles não põem ovos, seus filhotes nascem já formados. Salmão é um outro tipo de peixe, e é considerado uma delícia. O canário é um pássaro amarelo. Uma avestruz é um tipo de pássaro grande que não voa, apenas anda. Os mamíferos normalmente andam para se mover, como por exemplo uma vaca. As vacas dão leite, mas também servem elas mesmas de comida (carne). Contudo, nem todos os mamíferos andam para se mover. Por exemplo, o morcego voa.

Considere ainda que existem os seguinte animais:

1. *Piupiu, que é um canário.*

2. *Nemo, que é um peixe.*

3. *Tutu, que é um tubarão.*

4. *Mimosa, que é uma vaca.*

5. *Vamp, que é um morcego.*

6. *Xica, que é uma avestruz.*

7. *Alfred, que é um salmão.*

Defina fatos e regras Prolog que representam as premissas acima, e formule consultas Prolog para responder às seguintes perguntas:

1. *O piupiu voa?*

2. *Qual a cor do piupiu?*

3. *A Xica voa?*

4. *A Xica tem asas?*

5. *O Vamp voa? Tem asas? Poem ovos?*

6. *Quais os nomes dos animais que põem ovos?*

7. *Quais os nomes dos animais que são comestíveis?*

8. *Quais os nomes dos animais que se movem andando?*
9. *Quais os nomes dos animais que se movem nadando mas não põem ovos?*

As questões 10.6 até 10.8 são relacionadas ao programa Prolog apresentado a seguir:

```
animal(urso).
animal(peixe).
animal(peixinho).
animal(guaxinim).
animal(raposa).
animal(coelho).
animal(veado).
animal(lince).
planta(alga).
planta(grama).

come(urso, peixe).
come(peixe, peixinho).
come(peixinho, alga).
come(guaxinim, peixe).
come(urso, guaxinim).
come(urso, raposa).
come(raposa, coelho).
come(coelho, grama).
come(urso, veado).
come(veado, grama).
come(lince, veado).

presa(X) :- come(_, X), animal(X).
```

Exercício 10.6. *Quais as respostas para as seguintes consultas?*

1. *?- come(urso, peixinho).*
2. *?- come(raposa, coelho).*
3. *?- come(guaxinim, X).*
4. *?- come(X, grama).*
5. *?- come(urso, X), come(X, coelho).*
6. *?- presa(X), not(come(raposa, X)).*

Exercício 10.7. *Formule regras para as seguintes relações:*

1. *herbívoro;*
2. *carnívoro.*

Exercício 10.8. *Utilizando a regra da questão anterior, elabore consultas para as seguintes perguntas:*

1. *Quais animais são herbívoros?*

2. *Quais animais são carnívoros?*

3. *Quais animais herbívoros são presas de uma raposa?*

Exercício 10.9. *Elabore um programa Prolog que forneça o nome da capital de qualquer estado brasileiro.*

Exercício 10.10. *Implemente um programa para determinar quais tipos sanguíneos podem doar/receber sangue de quais tipos. A tabela seguinte fornece a informação necessária para a implementação.*

Tabela 1: Tipos sanguíneos.

	A	B	AB	O
A	Doa/Recebe	-	Doa	Recebe
B	-	Doa/Recebe	Doa	Recebe
AB	Recebe	Recebe	Doa/Recebe	Recebe
O	Doa	Doa	Doa	Doa/Recebe

Exercício 10.11. *Defina:*

1. *um predicado que forneça a intersecção de duas listas;*

2. *um predicado que identifique se um conjunto de elementos está contido em uma lista;*

3. *dois predicados que identifiquem se uma lista possui tamanho par ou ímpar;*

4. *um predicado que escreva uma lista em ordem inversa. Dica: utilize concatenação;*

5. *um predicado que retorne o maior valor contido em uma lista numérica;*

6. *um predicado para obter a soma dos* N *primeiros números naturais.*

Exercício 10.12. *Implemente um programa que retorne a N-ésima potência de um número.*

Exercício 10.13. *As regras abaixo identificam se um número é positivo, negativo ou zero. Defina regras que realizem o mesmo procedimento, porém, de maneira mais eficiente. Dica: utilize cortes.*

```
checagem(N, positivo) :- N > 0.
checagem(0, zero).
checagem(N, negativo) :- N < 0.
```

Exercício 10.14. *Implemente um programa que classifique se uma pessoa é considerada: criança (idade <= 12), adolescente (12 < idade <= 18), adulto (18 < idade <= 65) ou idoso (65 < idade).*

Obs.: Utilize cortes para melhorar a eficiência do programa.

Exercício 10.15. *Implemente um programa que forneça o signo de uma pessoa, mediante a uma consulta do tipo* dataNascimento(DD,MM, Signo).

Obs.: Utilize cortes para melhorar a eficiência do programa.

Exercício 10.16. *Um estudante acostumado a usar linguagens procedimentais está desenvolvendo um compilador em Prolog. Uma das tarefas consiste em traduzir um código de erro para uma pseudo-descrição em português. O código por ele usado é:*

```
traduza(Codigo, Significado):- Codigo=1, Significado=integer_overflow.
traduza(Codigo, Significado):- Codigo=2, Significado=divisao_por_zero.
traduza(Codigo, Significado):- Codigo=3, Significado=id_desconhecido.
```

O código funciona, porém, pode ser implementado de outra maneira (mais descritiva e menos procedimental). Melhore o código.

Referências

[1] Ivan Bratko. *Prolog Programming for Artificial Intelligence.* Addison-Wesley, 1986.

[2] L. Sterling e E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[3] Marco Antônio Casanova, Fernando Giorno, and Antônio L. Furtado. *Programação em lógica e a linguagem Prolog.* Edgar Blucher, 1987.

[4] Judith L. Gersting. *Fundamentos matemáticos para à Ciência da Computação.* LTC, 2001.

[5] Logic Programming Associates. Logic Programming Associates. www.lpa.co.uk/, 2006.

[6] Mike Brady. Open Prolog Home Page. www.cs.tcd.ie/open-prolog/, 2006.

[7] The Computational logic, Languages, Implementation, and Parallelism Laboratory. The Ciao Prolog Development System WWW Site. www.clip.dia.fi.upm.es/Software/Ciao, 2006.

[8] LIACC-Universidade do Porto and COPPE Sistemas-UFRJ. Yet Another Prolog. www.ncc.up.pt/ vsc/Yap, 2006.

[9] Jan Wielemaker. What is SWI-Prolog? www.swi-prolog.org/, 2006.

[10] Swedish Institute of Computer Science. SICStus Prolog. www.sics.se/sicstus/, 2006.

[11] Amzi! Inc. AMZI! www.amzi.com/, 2006.

[12] Prolog Development Center. Visual Prolog. www.visual-prolog.com/, 2006.

[13] Daniel Diaz. The GNU Prolog web site. gnu-prolog.inria.fr, 2003.

[14] Stony Brook University. Welcome to the home of XSB! xsb.sourceforge.net/, 2006.

[15] TRINC. Trinc Prolog R3a. www.trinc-prolog.com, 2006.

[16] University of Amsterdam. SWI-Prolog 5.6.17 Reference Manual. gollem.science.uva.nl/SWI-Prolog/Manual/, 2006.