INTRODUÇÃO À INTELIGÊNCIA ARTIFICIAL

PARTE 4. A LINGUAGEM PROLOG BÁSICA

4.1. Introdução

A idéia de usar lógica como um formalismo executável em computador recebeu um grande ímpeto com o advento da linguagem PROLOG ("PROgrammation en LOGic"). Desenvolvida na década de 70 em Edinburgh, a linguagem Prolog tem a sua aplicação dirigida à computação simbólica (não-numérica). Trata-se de uma evolução das linguagens de computador, pois Prolog tem características de linguagens procedimentais (o como), e de linguagens declarativas (o que).

4.1.1. Exemplo: relações familiares

Prolog é adequada para resolver problemas que envolvem objetos e relações entre objetos. Veja o exemplo da árvore familiar na figura 1.

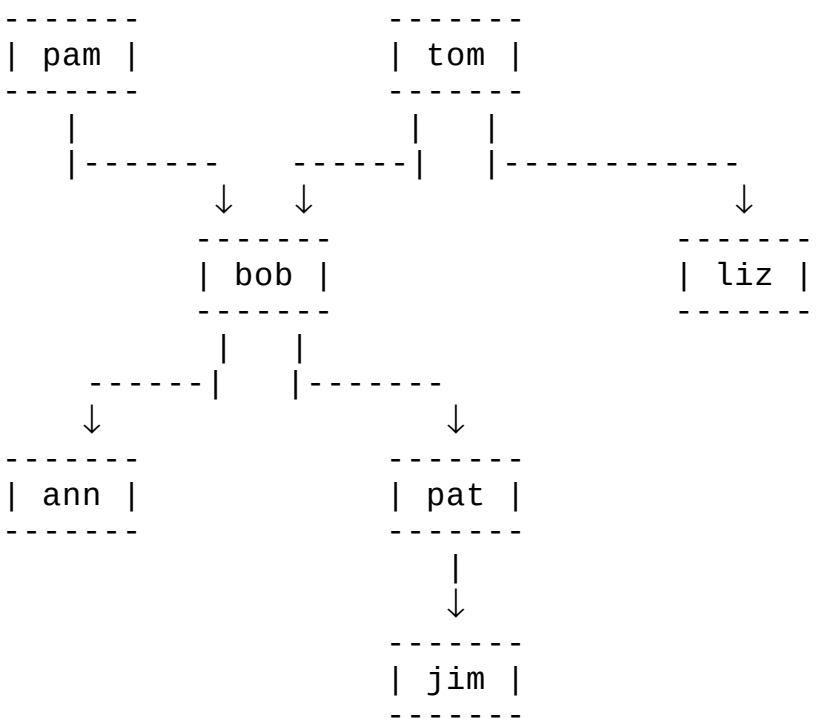

Fig. 1. Uma árvore familiar.

O fato de que Tom é "pais" (pai/mãe) de Bob se escreve:

pais(tom,bob).

pais → relação
tom, bob → argumentos

Os fatos na árvore familiar do exemplo são:

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

São 6 cláusulas declarando fatos sobre a relação "pais". Uma vez comunicado ao sistema Prolog os fatos sobre a relação "pais", pode-se colocar questões:

```
? - pais(bob,pat).
yes
? - pais(liz,pat).
no
? - pais(X,liz).
X = tom
```

Outra relação: avós:

```
? - pais(Y,jim),pais(X,Y).
X = bob
Y = pat
```

RESUMO:

- uma relação é definida pelo estabelecimento das n-uplas de objetos que satisfazem a relação.

- um programa Prolog é constituído de cláusulas. cada cláusula termina com um ponto.

- os argumentos das relações podem ser (entre outras coisas) constantes ou variáveis.

- perguntas ao sistema Prolog são constituídas de um ou mais objetivos. uma seqüência de objetivos separados por vírgulas significa a conjunção desses objetivos.

4.2. Estendendo o exemplo através de regras

Vai-se incluir uma nova informação para a base de dados: o sexo das pessoas. Pode-se usar uma relação unária:

```
feminino(pam).
masculino(tom).
```

ou usar uma relação binária:

```
sexo(pam,feminino).
sexo(tom,masculino).
```

Para a também nova relação "filhos" (filho/filha), que é o inverso da relação pais.

```
  cabeça        corpo
filhos(Y,X) :- pais(X,Y).
(conclusão)     (condição)
```

em Linguagem de Predicados de Primeira Ordem:

$\forall x \forall y(pais(x,y) \rightarrow filhos(y,x))$

A diferença principal entre fatos e regras, é que o fato é uma cláusula incondicionalmente verdadeira, e a regra é uma cláusula que pode ser verdade dependendo se alguma condição for verdade.

Como as regras são usadas em Prolog? Para a verificação da veracidade do seguinte fato:

```
? - filhos(liz,tom).      % liz é filha de tom?
```

o procedimento do Prolog, para executar a prova de tal fato é o seguinte:

1) procura tal fato na "base de conhecimento". Não existe tal fato.

2) procura se existe uma regra sobre a relação filhos. Como existe, aplica a particular instanciação X = tom e Y = liz, ou seja,

```
filhos(liz,tom) :- pais(tom,liz).
```

3) o objetivo original filhos(liz,tom) é substituído por um novo objetivo:

```
pais(tom,liz).
```

4) o novo objetivo é encontrado como um fato na "base de conhecimento" e o Prolog responde *yes*.

RESUMO:

1. Cláusulas em Prolog são de três tipos: fatos, regras e questões.

2. Fatos declaram coisas verdadeiras. Regras declaram coisas que são verdade dependendo de uma condição. Questões: através delas o usuário pode perguntar ao programa sobre a verdade de certas coisas.

3. Cláusulas em Prolog são constituídas de cabeça e corpo. O corpo é uma lista de objetivos separados por vírgulas, esta entendida como conjunção de objetivos.

4. Fatos são cláusulas que possuem um corpo vazio. Questões são cláusulas que possuem somente corpo. Regras são cláusulas que possuem cabeça e corpo (não vazio).

5. Durante a execução, uma variável pode ser instanciada por algum objeto.

6. Variáveis são assumidas ser universalmente quantificadas e são lidas como "para todo". Leituras alternativas são possíveis para variáveis que aparecem somente no corpo.

4.1.3. Definição de regra recursivamente

Para definir uma nova relação, predecessor, vai-se fazê-lo da seguinte forma:

```
predecessor(X,Z) :- pais(X,Z).
predecessor(X,Z) :- pais(X,Y), predecessor(Y,Z).
```

Note que a definição do predicado predecessor usa recursividade. A programação recursiva é um dos princípios fundamentais da programação em Prolog. A relação predecessor é definida por duas cláusulas, o que consiste em um procedimento.

4.1.4. Como Prolog responde questões

Uma questão em Prolog é uma seqüência de objetivos (um ou mais). Prolog tenta satisfazer os objetivos, isto é, demonstrar que os objetivos seguem logicamente a partir dos fatos e regras do programa. Se as questões contêm variáveis, Prolog também tenta achar a particular instanciação que satisfaz os objetivos. Os fatos e regras são aceitos como um conjunto de axiomas (hipóteses). A

questão do usuário é aceita como uma possível tese de um teorema. Prolog tenta provar esse teorema (demonstrar que ele segue logicamente dos axiomas). Prolog encontra a seqüência de prova no sentido inverso (demonstração por absurdo): "refutação".

Exemplo: seqüência de prova da conjectura (tese):

? - predecessor(tom,pat).

1) Prolog procura um fato que combine com o objetivo:

se sim: *yes*
se não: vai para o passo 2.

2) Prolog procura uma regra cuja cabeça combine com o objetivo:

se não: *no*
se sim: vai para o passo 3.

3) Prolog "dispara" a regra para a particular instanciação

predecessor(X,Z) :- pais(X,Z).
X = tom
Z = pat

e substitui o objetivo corrente por

? - pais(tom,pat).

4) Prolog procura uma cláusula cuja cabeça combina com o objetivo (novo)

se sim e cláusula = fato: *yes*
se sim e cláusula = regra: "dispara" a regra
se não: *backtrack*

5) Prolog "dispara" a segunda regra

predecessor(X,Z) :- pais(X,Y), predecessor(Y,Z).
X = tom
Z = pat

e substitui o objetivo corrente

? - pais (tom,Y), predecessor (Y,pat).

6) Prolog procura o primeiro objetivo, que combina com o fato pais(tom,bob), instanciando Y = bob.

7) resta analisar

? - predecessor(bob,pat).

8) "dispara" a primeira regra novamente e substitui o objetivo

? - pais(bob,pat).

que finalmente é encontrado como fato no banco de conhecimento.

4.1.5. Significado declarativo e procedimental

É possível distinguir entre os significados declarativo (o que) do procedimental (o como) de um programa Prolog. A habilidade do Prolog realizar muitos detalhes procedimentais por si mesmo é considerado uma de suas vantagens.

RESUMO

- Programação em Prolog consiste em definir relações e fazer questões sobre essas relações.

- Um programa consiste de cláusulas de três tipos: fatos, regras e questões.

- Uma relação pode ser especificada por fatos simplesmente estabelecendo as n-uplas de objetos que satisfazem a relação, ou estabelecendo regras a respeito da relação.

- Um procedimento é um conjunto de cláusulas a respeito de uma mesma relação.

- Questionar sobre relações, através de perguntas, relembra questionar um banco de dados. A resposta consiste de um conjunto de objetos que satisfazem a questão.

- A resposta é obtida através de um complexo processo que envolve inferência lógica, exploração entre alternativas, "backtracking".

- Existem dois significados nos programas Prolog: declarativo e procedimental. O declarativo é vantajoso do ponto de vista da programação. Apesar de que os detalhes procedimentais muitas vezes devem ser considerados pelo programador.

4.2. Sintaxe e significado de programas Prolog

4.2.1. Objetos de dados

A figura 2 mostra a classificação de objetos em Prolog. O sistema Prolog reconhece o tipo de um objeto no programa por sua forma sintática. Isto é possível porque a sintaxe do Prolog especifica formas diferentes para cada tipo de objetos de dados.

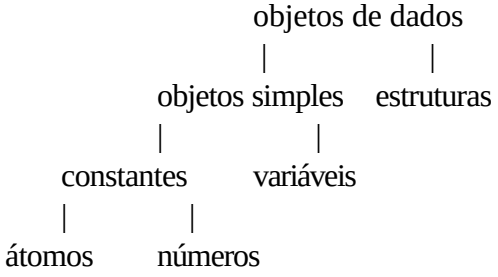

Fig. 2. Objetos de dados em Prolog.

4.2.1.1. Átomos e números

Os átomos podem ser construídos de três formas:

(1) Cadeias de letras, dígitos e o caractere "underscore" ('_'), começando com uma letra minúscula:

ana nil x25 x_25

(2) Cadeias de caracteres especiais:

```
<--->
=====>
```

Obs.: quando usar átomos desta forma, deve-se tomar cuidado, pois algumas cadeias de caracteres especiais já tem um significado pré-definido; um exemplo é ':-'.

(3) Cadeias de caracteres entre apóstrofes. Isto é útil quando se deseja, por exemplo, ter um átomo que começa com uma letra maiúscula. Colocando entre apóstrofes, torna-os distintos de variáveis:

'Tom'
'America_do_Sul'

Os números usados no Prolog, incluem números inteiros e reais:

1 1313 0 -97 3.14 -0.0035

4.2.1.2. Variáveis

As varáveis são cadeias de letras, dígitos e o caractere "underscore". Começam com uma letra maiúscula ou um "underscore":

X Resultado _23

Quando uma variável aparece apenas uma vez em uma cláusula, não há necessidade de nomeá-la. Por exemplo, na regra seguinte:

temcrianca(X) :- pais(X,Y).

Esta regra diz que para todo X, X tem uma criança se X é um pais (pai/mãe) de algum Y. Esta cláusula pode ser reescrita:

temcrianca(X) :- pais(X,_).

O escopo lexical de uma variável é uma cláusula. Isto significa que, por exemplo, se o nome X15 ocorre em duas cláusulas, então se trata de duas variáveis diferentes. Mas cada ocorrência de X15 dentro da mesma cláusula significa a mesma variável.

4.2.1.3. Estruturas

Estruturas são objetos que têm muitos componentes. Os componentes podem, por sua vez, serem estruturas. Por exemplo, a data pode ser vista como uma estrutura com três componentes: dia, mês e ano. As estruturas são tratadas como objetos simples. Para combinar os componentes em um único objeto, deve-se escolher um functor. Por exemplo, a data 1º de maio de 1993 pode ser escrita assim (veja figura 3):

data(1,maio,1993)

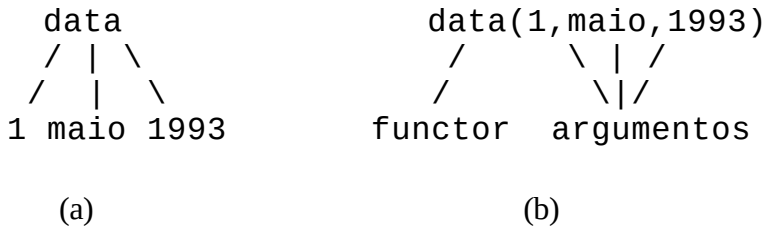

Fig. 3. Data é um exemplo de objeto estruturado: (a)representado como uma árvore; (b) como é escrito em Prolog.

Os componentes de um functor podem ser constantes, como no exemplo acima, ou variáveis, como em:

data(D,maio,1993)

Cada functor é definido por duas coisas:

(1) o nome, cuja sintaxe é a dos átomos;
(2) a aridade, ou seja, o número de argumentos.

Todos os objetos estruturados em Prolog são árvores, representados nos programas por termos. A figura 4 mostra a estrutura de árvore que corresponde à expressão aritmética

(a + b) * (c - 5)

Isto pode ser escrito, usando os símbolos '*', '+' e '-' como functors, da seguinte forma:

*(+(a,b),-(c,5))

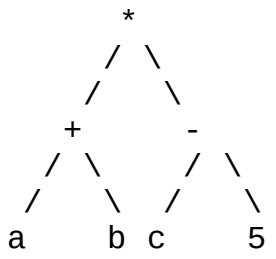

*Fig. 4. Uma estrutura de árvore que corresponde à expressão aritmética (a + b) * (c - 5).*

Na nossa base de dados da família, podemos incluir um predicado chamado data_nasc, de aridade 2, onde o primeiro argumento é o nome e o segundo, o functor *data*, que representa a data de nascimento da cada membro da família:

data_nasc(tom, data(D,M,A)).

4.2.2. "Matching" (Combinação)

Dados dois termos, diz-se que eles "combinam" se:

- são idênticos, ou
- as variáveis em ambos são instanciadas à objetos, de modo a serem idênticos.

Exemplo:

data(D,M,1993) e data(D1,maio,A1) COMBINAM, pois:
D = D1
M = maio
A1 = 1993, que são as INSTANCIAÇÕES.

A combinação ("matching") é um processo que pega como entradas dois termos e checa se eles combinam. Se os termos não combinam, diz-se que este processo falha. Se eles combinam então o processo é bem sucedido e as variáveis são instanciadas em ambos os termos para valores tais que tornem os mesmos idênticos. Considere a combinação de duas datas:

?- data(D,M,1993) = data(D1,maio,A1).

Já foi mencionado que a instanciação D = D1, M = maio e A1 = 1993 satisfaz a combinação. Existe, entretanto, outras instanciações que também satisfazem. Duas delas são:

D = 1
D1 = 1
M = maio
A1 = 1993

D = tres
D1 = tres
M = maio
A1 = 1993

Estas duas instanciações são ditas menos gerais que a primeira, pois elas atribuem constantes para as variáveis D e D1. A combinação, em Prolog, sempre resulta na instanciação mais geral. Esta instanciação mantém, na medida do possível, o maior número de variáveis, para que elas possam ser instanciadas mais tarde.

As regras gerais para decidir se dois termos, S e T, combinam são as seguintes:

(1) Se S e T são constantes então S e T combinam apenas se eles são o mesmo objeto.

(2) Se S é uma variável e T e qualquer coisa, então eles combinam, e S é instanciado a T. Se T é uma variável, então T é instanciado a S.

(3) Se S e T são estruturas então eles combinam apenas se
(a) S e T têm o mesmo functor principal e
(b) todos os seus componentes correspondentes combinam. A instanciação resultante é determinada pela combinação dos componentes. Veja o seguinte exemplo:

?- triangulo(ponto(1,1),A,ponto(2,3)) =

triangulo(X,ponto(4,Y),ponto(2,Z)).

Neste caso, tem-se duas estruturas. Para que seja possível combiná-las é necessário que elas possuam o mesmo functor principal (no caso, triangulo). Os seus componentes devem ser instanciados da seguinte forma: X = ponto(1,1); A = ponto(4,Y) e Z = 3. Ou seja, esta é a instanciação mais geral que satisfaz a combinação.

4.2.3. Significado declarativo

Já foi dito que os programas Prolog podem ser entendidos de duas formas: declarativamente e procedimentalmente. Seja a seguinte cláusula:

P :- Q, R.

onde P, Q e R têm a sintaxe dos termos. Leituras declarativas alternativas desta cláusula seriam:

P é verdadeiro se Q e R são verdadeiros.
A partir de Q e R, obter P.

Duas leituras procedimentais alternativas desta cláusula são:

Para resolver o problema P, primeiro resolva o subproblema Q e então o subproblema R.
Para satisfazer P, primeiro satisfaça Q e então R.

Portanto, a diferença entre as leituras declarativa e procedimental é que a última não apenas define as relações lógicas entre a cabeça da cláusula e as metas no corpo, mas também a ordem na qual as metas são processadas.

O significado declarativo dos programas determina se uma dada meta é verdadeira, e se for, para quais valores de variáveis ela é verdadeira. Para definir precisamente o significado declarativo é necessário introduzir o conceito de instância de uma cláusula. Uma instância de uma cláusula C é a cláusula C com cada uma de suas variáveis substituídas por algum termo. Uma variante de uma cláusula C é uma instância da cláusula C onde cada variável é substituída por outra variável. Por exemplo, considere a cláusula:

temcrianca(X) :- pais(X,Y).

Duas variantes desta cláusula são:

temcrianca(A) :- pais(A,B).
temcrianca(X1) :- pais(X1,X2).

Instâncias desta cláusula são:

```
temcrianca(pedro) :- pais(pedro,Z).
temcrianca(paulo) :- pais(paulo,pequeno(carolina)).
```

Dado um programa e uma meta G, o significado declarativo diz:

Uma meta G é verdadeira (isto é, satisfatível, ou segue logicamente do programa) se e somente se

(1) existe uma cláusula C no programa tal que
(2) existe uma cláusula instância I de C tal que
    (a) a cabeça de I é idêntica a G, e
    (b) todas as metas no corpo de I são verdadeiras.

Em geral, uma questão ao sistema Prolog é uma lista de metas separadas por vírgulas. Uma lista de metas é verdadeira se todas as metas na lista são verdadeiras para a mesma instanciação de variáveis. Os valores das variáveis resultam da instanciação mais geral.

Uma vírgula entre metas denota a conjunção de metas: todas elas devem ser verdadeiras. Mas Prolog também aceita a disjunção de metas: basta qualquer uma das metas numa disjunção ser verdadeira. A disjunção é indicada por um ponto-e-vírgula. Por exemplo,

```
P :- Q; R.
```

que é lido como: P é verdadeiro se Q é verdadeiro ou R é verdadeiro. O significado desta cláusula é equivalente ao das seguintes cláusulas juntas:

```
P :- Q.
P :- R.
```

4.2.4. Significado procedimental

O significado procedimental especifica como Prolog responde às questões. Responder a uma questão significa tentar satisfazer uma lista de metas, isto é, encontrar uma instanciação particular das variáveis que ocorrem nas metas de tal modo que os objetivos sigam logicamente do programa.

Esquematicamente:

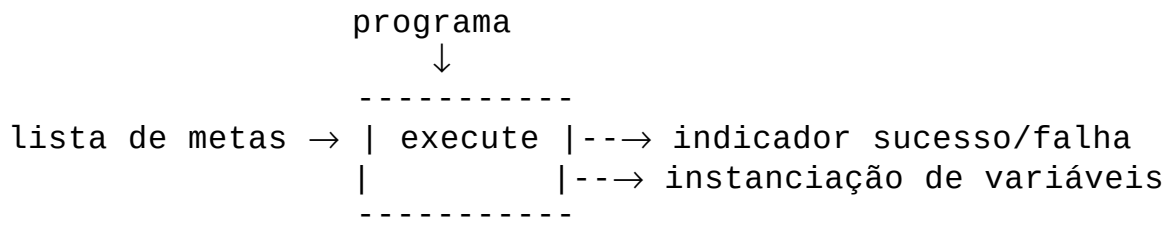

Fig. 5. Visão da entrada/saída do procedimento que executa uma lista de metas.

O significado das duas saídas na figura 5 resulta em: (1) o indicador de sucesso/falha é 'yes' se as metas são satisfatíveis e 'no' de outra forma. 'Yes' significa uma terminação bem sucedida e 'no' uma falha. (2) uma instanciação de variáveis é produzida apenas no caso de uma terminação bem sucedida; no caso de falha não há instanciação.

Veja o seguinte exemplo:

PROGRAMA

```
grande(urso).          % clausula 1
grande(elefante).      % clausula 2
pequeno(gato). % clausula 3

marrom(urso).          % clausula 4
preto(gato).           % clausula 5
cinza(elefante). % clausula 6

escuro(Z) :-           % clausula 7
        preto(Z).      % qualquer coisa preta e' escura

escuro(Z) :-           % clausula 8
        marrom(Z).     % qualquer coisa marrom e' escura
```

QUESTÃO

```
?- escuro(X), grande(X).          % quem e escuro e grande?
```

EXECUÇÃO ("TRACE")

(1) Lista de metas iniciais: *escuro(X), grande(X).*

(2) Percorrer o programa de cima para baixo, procurando uma cláusula cuja cabeça combina com *escuro(X)*, o primeiro objetivo. A cláusula 7 é encontrada:

escuro(Z) :- preto(Z).

Trocar a primeira meta pelo corpo instanciado da cláusula 7, obtendo uma nova lista de metas:

preto(X), grande(X).

(3) Percorrer o programa para achar uma cláusula cuja cabeça combine com *preto(X)*. A cláusula 5 é encontrada:

preto(gato).

Esta cláusula não tem corpo, tal que a lista meta, propriamente instanciada, se reduz para:

grande(gato).

(4) Percorrer o programa para a meta *grande(gato)*. Nenhuma cláusula encontrada. Portanto retorne ("backtrack") ao passo (3) e desfaça a instanciação *X = gato*. Agora a lista meta é novamente:

preto(X), grande(X).

Continuar percorrendo o programa para baixo da cláusula 5. Nenhuma cláusula encontrada. Portanto, retornar ("backtrack") para o passo (2), restituindo a lista de metas original *escuro(X), grande(X)* e continuar percorrendo abaixo da cláusula 7. A cláusula 8 é encontrada:

escuro(Z) :- marrom(Z).

Substituir a primeira meta na lista de metas por *marrom(X)*, dando:

marrom(X), grande(X).

(5) Percorrer o programa para combinar *marrom(X)*, achando *marrom(urso)*. Não há corpo, então a lista de metas se reduz a *grande(urso)*. Como esta cláusula é encontrada como fato (cláusula 1), então a lista de metas se reduz ao vazio. Isto indica a terminação bem sucedida e a instanciação da variável correspondente é:

X = urso

4.3. Listas

4.3.1. Representação de listas

A *lista* é uma estrutura de dados simples muito usada em programação não-numérica. Uma lista é uma seqüência de qualquer número de itens, tais como *ana, tenis, tom, esqui*. Tal lista pode ser escrita em Prolog como:

[ana, tenis, tom, esqui]

Esta é entretanto, apenas a aparência externa das listas. Como já foi visto, todos os objetos estruturados em Prolog são árvores.

Como se pode representar uma lista como um objeto Prolog padrão? Deve-se considerar dois casos: ou a lista está vazia ou não vazia. No primeiro caso, a lista é simplesmente escrita como o átomo do Prolog, []. No segundo caso, a lista pode ser vista como consistindo de duas coisas:

(1) o primeiro item, chamado a *cabeça* da lista;
(2) a parte remanescente da lista, chamada de *cauda*.

Para o exemplo acima, a cabeça é *ana* e a cauda é a lista

[tenis, tom, esqui]

Em geral, a cabeça pode ser qualquer coisa (qualquer objeto Prolog, por exemplo, uma árvore ou uma variável); a cauda deve ser uma lista. A cabeça e a cauda podem ser combinadas em uma estrutura por um functor especial. A escolha deste functor depende da implementação Prolog; assume-se aqui que é o ponto:

.(Cabeca, Cauda)

Como *Cauda* é por sua vez uma lista, que pode estar vazia ou ter sua própria cabeça e cauda. Portanto, para representar listas de qualquer tamanho é só seguir a estrutura estabelecida, como no exemplo abaixo:

.(ana, .(tenis, .(tom, .(esqui, []))))

A figura 6 mostra a estrutura de árvore correspondente. Note que a lista vazia aparece no termo acima. Isto ocorre, porque a última cauda é uma lista de um único item:

[esqui]

Esta lista tem a lista vazia como sua cauda:

[esqui] = .(esqui, [])

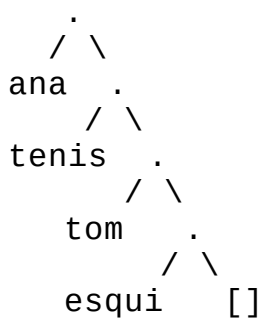

Fig. 6. Representação em árvore para a lista [ana, tenis, tom, esqui].

4.3.2. Exemplos de aplicações de listas

1. Concatenação de duas listas L = L1 + L2:

```
conc ([], L, L). % se L1 é vazia, L = L2
conc ({X|L1], L2, [X|L3]) :-        % senão, X (cabeça de L1) será a cabeça de
        conc (L1, L2, L3).          % L e a cauda de L1 concatenada com L2,
                                    % resultará na cauda de L
```

2. X é membro de uma lista:

```
membro (X, [X|T].                   % X é membro se for cabeça
membro (X, [H|T]) :-                % senão, X é membro da cauda
        membro (X, T).
```

3. adicionar um elemento na cabeça da lista:

```
ad (X, L, [X|L]).                   % X vira cabeça da nova lista
```

4. apagar um elemento X de uma lista:

```
apag (X, [X|T], T).                 % se X for a cabeça, retira-a
apag (X, [Y|T], [Y|T1]) :-          % senão apaga X da cauda
        apag (X, T, T1).
```

Prolog é a linguagem para *programação lógica* e pode fazer muitas coisas. Mas tem quatro fraquezas lógicas fundamentais (Rowe, 1988):

◊ Prolog não permite fatos ou conclusões disjuntivos, ou seja, sentenças onde uma ou mais coisas são verdadeiras, mas não se sabe quais.
◊ Prolog não permite fatos ou conclusões negativos, isto é, sentenças diretas onde alguma coisa é falsa.
◊ Prolog não permite que muitos fatos, conclusões ou regras tenham quantificação existencial, isto é, sentenças onde exista algum valor de variável, ainda que não se saiba qual, tal que o predicado que a contém seja verdadeiro.
◊ Prolog não permite diretamente *lógica de segunda ordem*, nomes de predicados como variáveis, isto é, sentenças sobre P onde P significa um nome de predicado.

RESUMO:

- Uma lista é uma estrutura de dados que ou está vazia ou consiste de duas partes: uma *cabeça* e uma *cauda*. A cauda por sua vez também é uma lista.

- As listas são manipuladas pelo Prolog com um caso especial de *árvores binárias*. Para melhorar a legibilidade, Prolog provê uma notação especial para listas:

```
        [Item1, Item2, ...]
ou
        [Cabeca | Cauda]
ou
        [Item1, Item2, ... | Outros]
```

4.4. Exemplo de um programa completo Prolog

O seguinte exemplo traz um programa Prolog para Processamento de Linguagem Natural, com as diretivas (*entry-points*) para compilação no Arity Prolog:

```
% ANALISADOR SINTÁTICO
% Autor: João Luís Garcia Rosa
%          Instituto de Informática - PUCCAMP
%           e-mail: joaol@zeus.puccamp.br
%
% definindo os "entry-points" para o compilador

:- module anasin.
:- public main/0.

main :- repeat, anasin, fail.
```

```
anasin :- nl, write('Entre com a frase: '),
        read(X),
        frase(X,A),
        nl, write('Frase CORRETA sintaticamente'), nl, nl.

% as letras são transformadas em minúsculas

letra(C,D) :-
        C >= "A", C =< "Z",
        D is C + 32.
letra(C,C) :-
        C >= "a", C =< "z".

identificador(C,D) :- letra(C,D).

% o predicado append coloca uma lista no
% final de outra lista

append([],L,L).
append([H|T],L,[H|T1]) :- append(T,L,T1).

% o predicado pega_resto_pal pega o restante da
% palavra e vai criando uma nova lista, ate encontrar
% o espaço ou o ponto

pega_resto_pal([H|T],Lista,Pal,X) :-
        identificador(H,Id), !,
        append(Lista,[Id],Nlista),
        pega_resto_pal(T,Nlista,Pal,X).
pega_resto_pal([32|T],Lista,Pal,T) :-
        name(Pal,Lista), !.
pega_resto_pal(["."|T],Lista,Pal,T) :-
        name(Pal,Lista), !.
pega_resto_pal([],Lista,Pal,[]) :-
        !,name(Pal,Lista).

% tokenizar cria a partir de uma seqüência de
% caracteres ASCII teclados, uma lista com todas as
% palavras separadas por virgulas

tokenizar([H|T],Lista,L) :-
        letra(H,Letra), !,
        pega_resto_pal(T,[Letra],Pal,Res),
        append(Lista,[Pal],NovaLista),
        tokenizar(Res,NovaLista,L).

tokenizar([32|T],Lista,L) :-
        !, tokenizar(T,Lista,L).

tokenizar([],Lista,Lista).


% gramática GCD do F. Pereira e D. Warren, com
% gênero e numero
```

```
sen([NP,VP],S0,S) :-
        frsu(_,N,NP,S0,S1),frve(N,VP,S1,S2).

frsu(G,N,[SUBST,SUBST2],S0,S) :-
        det1(G,N,S0,S1),
        determ(G,N,S1,S2),
        expsubst(G,N,SUBST,S2,S3),
        expcom(G1,N1,SUBST2,S3,S),!.
frsu(G,N,[SUBST],S0,S) :-
        expsubst(G,N,SUBST,S0,S),!.
frsu(_,sing,[SUBST],S0,S) :-
        nome(SUBST,S0,S).

expsubst(G,N,[SUBST],S0,S) :-
        subst(G,N,SUBST,S0,S1),
        expadj(G,N,S1,S).
expsubst(G,N,[SUBST],S0,S) :-
        expadj(G,N,S0,S1),
        expsubst(G,N,SUBST,S1,S).
expsubst(G,N,[SUBST],S0,S) :-
        subst(G,N,SUBST,S0,S).

expadj(G,N,S0,S) :-
        adj(G,N,S0,S).
expadj(G,N,S0,S) :-
        adj(G,N,S0,S1),
        expadj(G,N,S1,S).

expcom(G,N,[SUBST],S0,S) :-
        pcom(S0,S1),
        frsu(G,N,SUBST,S1,S).

expcom(G,N,x,S,S).

frve(N,[VTD,NP],S0,S) :-
        vtd(N,VTD,S0,S1),frsu(_,N1,NP,S1,S),!.
frve(N,[VTI,PP],S0,S) :-
        vti(N,VTI,S0,S1),fprep(G,N1,PP,S1,S),!.
frve(N,[VR],S0,S) :-
        part(S0,S1),vr(N,VR,S1,S).

fprep(G,N,[NP],S0,S) :-
        prep(G,N,S0,S1),frsu(G,N,NP,S1,S).


det1(masc,pl,S0,S) :- conecta(S0,todos,S).
det1(fem,pl,S0,S) :- conecta(S0,todas,S).
det1(_,_,S,S).

determ(masc,sing,S0,S) :- conecta(S0,o,S).
determ(fem,sing,S0,S) :- conecta(S0,a,S).
determ(masc,pl,S0,S) :- conecta(S0,os,S).
determ(fem,pl,S0,S) :- conecta(S0,as,S).
```

```
determ(_,_,S,S).

subst(masc,sing,alimento,S0,S) :- conecta(S0,alimento,S).
subst(masc,pl,alimento,S0,S) :-  conecta(S0,alimentos,S).

adj(_,sing,S0,S) :- conecta(S0,elegante,S).
adj(_,pl,S0,S) :- conecta(S0,elegantes,S).
adj(masc,sing,S0,S) :- conecta(S0,belo,S).
adj(fem,sing,S0,S) :- conecta(S0,bela,S).

nome(homem,S0,S) :- conecta(S0,joao,S).
nome(mulher,S0,S) :- conecta(S0,maria,S).

vtd(sing,comer,S0,S) :- conecta(S0,comeu,S).
vtd(sing,quebrar,S0,S) :- conecta(S0,quebrou,S).
vtd(sing,mover,S0,S) :- conecta(S0,moveu,S).
vr(sing,mover,S0,S) :- conecta(S0,moveu,S).
vti(sing,bater,S0,S) :- conecta(S0,bateu,S).

part(S0,S) :- conecta(S0,se,S).

prep(_,_,S0,S) :- conecta(S0,em,S).
prep(masc,sing,S0,S) :- conecta(S0,no,S).
prep(fem,sing,S0,S) :- conecta(S0,na,S).

pcom(S0,S) :- conecta(S0,com,S).

conecta([W|S],W,S).

frase(X,A) :- tokenizar(X,Y,Z),!,sen(A,Z,[]),write(A),
                    create(H1,frase),write(H1,A),close(H1).
```

4.5. *Exercícios*

1. Traduza as seguintes declarações em regras do Prolog:
   (a) "Todo mundo que tem criança é feliz".
   (b) "Para todo X, se X tem uma criança que tem uma irmã então X tem duas crianças".

2. Defina a relação *netos* usando a relação *pais* dada em aula.

3. Defina um predicado Prolog para concatenar duas listas.

4. Defina um predicado Prolog para achar o mínimo entre quatro números fornecidos.

5. Quais dos seguintes objetos é sintaticamente correto em Prolog? Que tipo de objetos são eles?
   (a) Diana
   (b) diana
   (c) 'Diana'

(d) 'diana'
(e) _diana
(f) ir(diana,sul)
(g) 45
(h) 'Diana ir sul'
(i) 5(X,Y)
(j) +(north,west)
(k) tres(Preto(Gatos))

6. As seguintes operações de "matching" terão sucesso ou falharão. Se tiverem sucesso, quais são as instanciações das variáveis?

(a) ponto(A,B) = ponto(1,2)
(b) ponto(A,B) = ponto(X,Y,Z)
(c) mais(2,2) = 4
(d) +(2,D) = +(E,2)
(e) triangulo(ponto(-1,0),P2,P3) = triangulo(P1,ponto(1,0),ponto(0,Y))

Obs: Para o item (e), a instanciação resultante define uma família de triângulos. Como você descreveria esta família?

7. Defina a relação *gemeos(Crianca1,Crianca2)*, para achar gêmeos na base de dados da família.

8. Escreva um programa Prolog para reconhecimento de animais, baseado na seguinte base de conhecimento:

(a) animal que tem pena não é mamífero.
(b) animal que tem pelo é mamífero e não é ave.
(c) animal que não é mamífero é ave.
(d) animal que não tem pena tem pelo.
(e) animal que não é mamífero e voa é sabiá.
(f) animal que é ave e não voa é pingüim.
(g) animal que é mamífero e não voa é vaca.
(h) animal que voa e não é ave é morcego.

Consulte o seu programa para identificar qual é o animal que voa e não tem pena. Qual é a seqüência de prova usada?

BIBLIOGRAFIA

- Bratko, I. (1986). Prolog Programming for Artificial Intelligence - Addison-Wesley Pub. Co.

- Rowe, N. C. (1988). Artificial Intelligence Through Prolog - Prentice Hall.

- Soares Filho, S. & Tavares, H.. (1991). Notas de aula da disciplina *Engenharia do Conhecimento I*. DCA - FEE - UNICAMP, 2º semestre.