UTFPR - CAMPUS PONTA GROSSA - COCIC

Disciplina de Lógica para Computação – 2022/1

Programação em Lógica - Prolog

Professor: Gleifer Vaz Alves

Sumário

1 Introdução

Este documento elementos do paradigma de programação em lógica e em especial a linguagem Prolog. Ainda há exemplos e exercícios, bem como referências de ferramentas e leituras da área.

A maior parte do material dessa apostila é baseada no seguinte livro:

- PALAZZO, L. M. *Introdução à Programação Prolog.* EDUCAT. 1997.

2 Linguagem Prolog

A principal utilização da linguagem Prolog diz respeito ao domínio da programação simbólica, não-numérica, sendo especialmente adequada à solução de problemas, envolvendo objetos e relações entre objetos.

A linguagem Prolog auxilia a demonstrar que a lógica é um formalismo adequado para representar e processar conhecimento. Em Prolog não é necessário que o programador descreva os procedimentos necessários para a solução de um problema, permitindo que o programador expresse de forma *declarativa* apenas a estrutura lógica, por meio de **fatos**, **regras** e **consultas**.

Algumas das principais características da linguagem são destacadas:

- É uma linguagem orientada ao processamento simbólico;
- Representa uma implementação da lógica como linguagem de programação;
- Apresenta uma semântica declarativa inerente à lógica;
- Permite a obtenção de respostas alternativas (*não apenas uma única*);
- Suporta código recursivo e iterativo para a descrição de processos e problemas, dispensando os mecanismos tradicionais de controle, como: *while*, *for*, etc;
- Permite associar o processo de especificação ao processo de codificação de programas;
- Representa programas e dados através do mesmo formalismo.

Nas seções que seguem serão vistos exemplos de árvores genealógicas para assim introduzir conceitos da programação em lógica, como **fatos**, **regras** e **consultas**.

Para auxiliar neste entendimento ilustra-se na Fig. 1 a árvore genealógica da família *Simpson*.

2.1 Fatos

Considerando a árvore genealógica da família *Simpson* (vista na Fig. 1) e sabendo que numa visão futurista do desenho, a personagem *Lisa* tem filha chamada *Zia* (ver Fig. 2). Na Fig. 3 ilustra-se uma representação simbólica da respectiva árvore genealógica.

É possível definir entre os objetivos (indivíduos) representados na Fig. 3 uma relação denominada **progenitor**, a qual associa um indivíduo a seus progenitores.

Por exemplo, o fato de que *Homer* é um dos progenitores de *Bart* é representado da seguinte forma:

```
progenitor(homer, bart).
```

onde **progenitor** é o nome da relação e *homer* e *bart* são os respectivos argumentos. Devido a sintaxe da linguagem, os nomes desses argumentos devem iniciar com letra minúsucula. E a descrição do fato deve ser encerrada por um **ponto** (.).

Abaixo descreve-se por completo a relação **progenitor** (conforme ilustrada na Fig. 3). Neste programa Prolog tem-se ao todo 12 **cláusulas**, onde cada uma delas denota um **fato** acerca da relação **progenitor**.

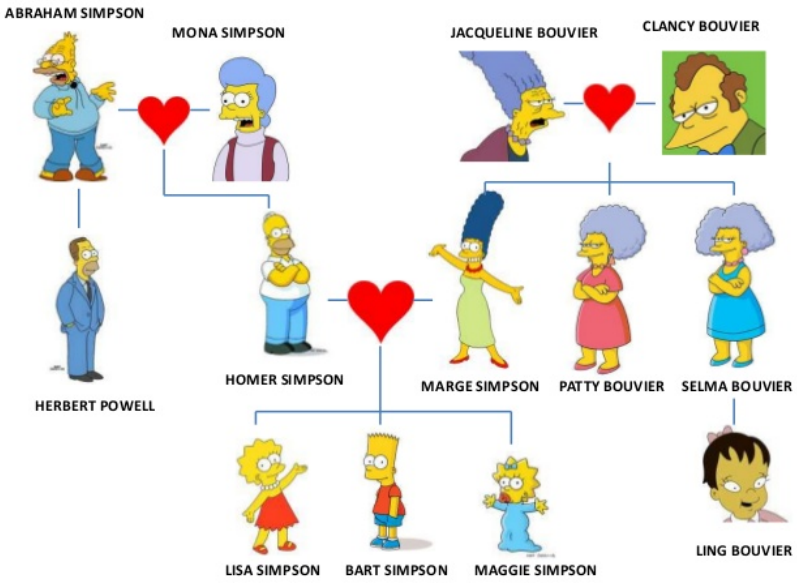

Figura 1: Árvore Genealógica - Família *Simpson*

Figura 2: Família *Simpson* - Zia (filha de Lisa)

```
progenitor(abraham, herbert).
progenitor(mona, herbert).
progenitor(abraham, homer).
progenitor(mona, homer).
progenitor(homer, lisa).
progenitor(homer, bart).
progenitor(homer, maggie).
progenitor(marge, lisa).
progenitor(marge, bart).
progenitor(marge, maggie).
progenitor(lisa, zia).
progenitor(milhouse, zia).
```

Se este programa for submetido a um sistema Prolog, será possível fazer algumas consultas a respeito das relações definidas. Por exemplo, *Homer é progenitor de Lisa?*. Para realizar consultas no Prolog é necessário utilizar os símbolos: `?-`. Esta combinação de símbolos denota que é formulada uma pergunta ao conforme a declaração do programa Prolog. Considerando a questão formulada acima, o sistema irá retornar *verdadeiro* (no caso, `true`), conforme ilustrado na Fig. 4.

Além disso, a Fig. 4 também representa duas consultas que retornam como resultado *falso* (no caso, **false**), visto que essas relações não estão de acordo com os fatos definidos anteriormente por meio da relação **progenitor**.

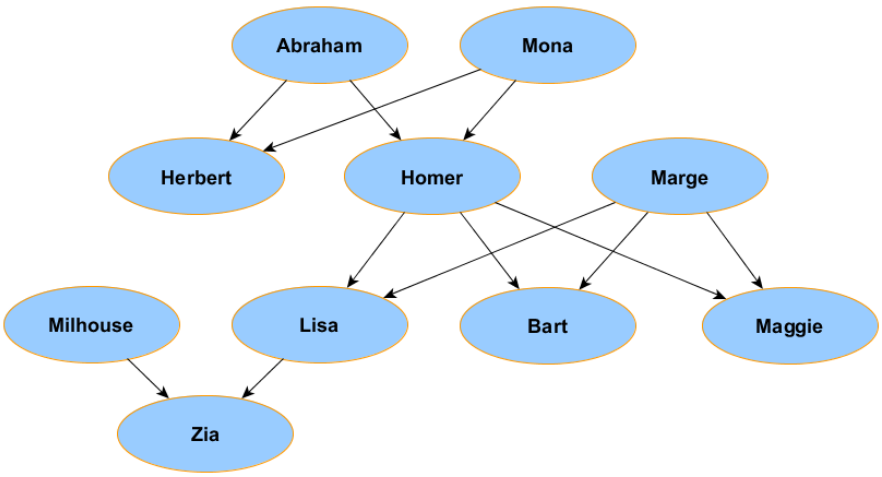

Figura 3: Árvore Genealógica - Família *Simpson* - Representação Simbólica

```
?- progenitor(homer, lisa).
true.
.
.

?- |    progenitor(homer,milhouse).
false.

?- progenitor(flanders, bart).
false.

?- |
```

Figura 4: Família *Simpson* - Exemplos de consultas

Também é possível fazer consultas utilizando variáveis, como: *Quem é o progenitor de Zia?*. Assim, o sistema não responderá apenas *true* ou *false*. De forma similar a questão: *Quem são os filhos de Marge?* pode ser formulada com a introdução de uma variável na posição do argumento correspondente aos filhos de *Marge*. Observe que neste caso, mais de uma resposta verdadeira pode ser encontrada. O sistema fornece a primeira resposta que encontrar e aguarda a manifestação do usuário. Se é digitado um ponto-e-vírgula (;), então o sistema busca outras respostas. Porém, se é digitado um ponto (.), então o sistema entende que essa resposta é suficiente. Esses tipos de consultados são ilustrados na Fig. 5.

Ainda é possível realizar uma consulta mais genérica, por exemplo, *Quem é progenitor de quem?*, ou com outra formulação:

Encontre X e Y tal que X é progenitor de Y.

Para esta consulta, o sistema irá responder todos os pares progenitor-filho até que estes se esgotem, ou até que se resolva encerrar a geração de respostas, digitando um ponto (.). A Fig. 6 ilustra esse tipo de consulta, onde são apresentadas as seis primeiras soluções.

Dando continuidade ao tipo de consultas que podem ser executadas, cabe destacar uma formulação um pouco mais elaborada que as anteriores: *Quem são os avós de Bart?*. Por enquanto, o programa Prolog não possui diretamente a relação avô, ou avó. Assim, essa consulta deve ser dividida em duas etapas, como pode visto na Fig. 7, onde tem-se que:

1. Quem é o progenitor de Bart? (*Por exemplo,* Y) e
2. Quem é o progenitor de Y? (*Por exemplo,* X)

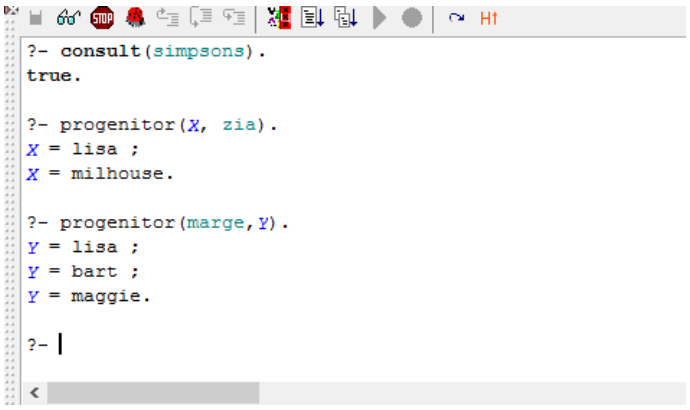

Figura 5: Família *Simpson* - Exemplos de consultas com respostas múltiplas

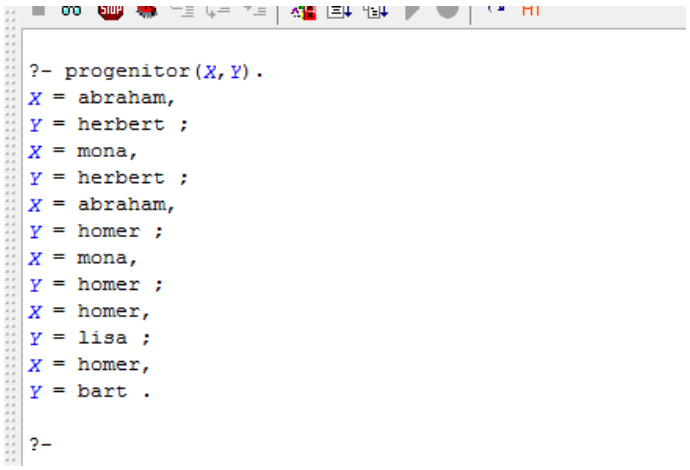

Figura 6: Família *Simpson* - Exemplos de consultas com pares progenitor-filho

Esta consulta em Prolog é escrita como uma sequência de duas consultas simples, cuja leiura é a seguinte:

Encontre X e Y tais que X é progenitor de Y e Y é progenitor de *Bart*.

O resultado para tal tipo de consulta é ilustrado na Fig. 8.

Seguindo uma linha de raciocínio similar é possível questionar: *Quem são os netos de Mona*? (ver Fig. 9).

Ainda há uma outra consulta que pode ser realizada: *Lisa e Maggie possuem algum progenitor em comum?*. Para esta consulta é necessário decompor a questão em duas etapas:

Encontre X tal que X seja simultaneamente progenitor de *Lisa* e *Maggie*.

Esta consulta é ilustrada na Fig. 9.

Conforme os exemplos ilustrados até o momento é possível resumir alguns dos principais pontos discutidos:

- Uma relação como **progenitor** pode ser facilmente definida estabelecendo as tuplas de objetos que satisfazem a relação;
- Um programa Prolog é constituído de cláusulas, cada uma delas deve ser encerrada por um ponto (.);
- Os argumentos das relações podem ser objetos concretos, como *Bart* e *Lisa*, ou então objetos genéricos, como X e Y. Objetos concretos em um programa são denomiados *átomos*, enquanto que objetos genéricos são conhecidos como *variáveis*;

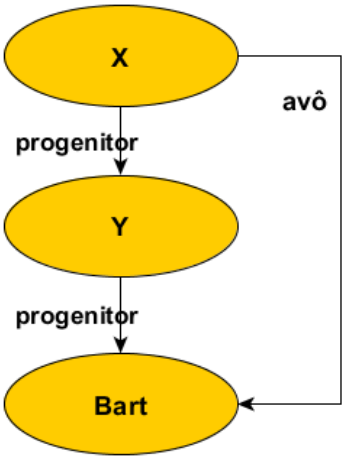

Figura 7: Relação *avô* em função de progenitor

```
?- progenitor(X,Y), progenitor(Y,bart).
X = abraham,
Y = homer ;
X = mona,
Y = homer ;
false.

?-
```

Figura 8: Família *Simpson* - Exemplo de consulta com relação progenitor-progenitor-filho

- Consultas ao sistema são constituídas por um ou mais objetivos, cuja sequência denota a sua conjunção;
- Uma resposta a uma consulta pode ser *positiva* (*true*) ou *negativa* (*false*), dependendo se o objetivo correspondente foi alcançado, ou não. No primeiro caso é dito que a consulta foi *bem-sucedida*, ao passo que no segundo caso, a consulta *falhou*;
- Se diversas respostas satisfazem uma dada consulta, então o sistema Prolog irá fornecer tantas respostas quanto for desejado pelo usuário.

2.2 Regras

O programa da árvore genealógica pode ser facilmente ampliado de diferentes maneiras. Incialmente podemos adicionar informações sobre o sexo das pessoas representadas na família *Simpson*. Isso pode ser feito por meio de novos fatos, como segue:

```
masculino(abraham).
masculino(herbert).
masculino(homer).
masculino(bart).
masculino(milhouse).
feminino(mona).
feminino(marge).
feminino(lisa).
```

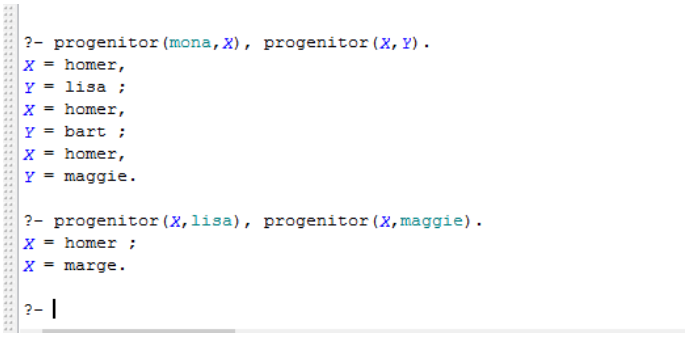

Figura 9: Família *Simpson* - Exemplos de consultas com relações: avó-progenitor-progenitor e irmãs

```
feminino(maggie).
feminino(zia).
```

As relações introduzidas no programa são *masculino* e *feminino*. Essas relações são unárias, isto é possuem um único argumento. Uma relação binária, como progenitor, é definida entre pares de objetos, enquanto que as relações unárias podem ser usadas para declarar propriedades simples desses objetos.

A primeira clásula unária da relação *masculino* pode ser lida como: *Abraham é do sexo masculino.*

Poderia ser conveniente declarar a mesma informação presente nas relações unárias definidas acima, em uma única relação binária *sexo*:

```
sexo(abraham, masculino).
sexo(herbert, masculino).
sexo(homer, masculino).
...
```

Outra extensão do programa é criar a relação binária *filho*, a qual é de fato o inverso da relação *progenitor*. A declaração da relação *filho* é feita por meio de uma lista de fatos, cada fato fazendo referência a um par de pessoas, desde que uma pessoa seja filho de outra. Por exemplo,

```
filho(bart, homer).
```

Contudo, é possível definir a relação *filho* de uma forma mais elegante, considerando que tal relação é o inverso da relação *progenitor*, definida anteriormente. Tal definição alternativa é baseada na seguinte declaração lógica:

- Para todo X e Y
  - Y é filho de X, se
  - X é progenitor de Y.

Essa declaração lógica pode ser representada pela seguinte clásula (Prolog):

```
filho(Y, X) <- progenitor(X, Y).
```

a qual pode ser lida da seguinte forma: *Para todo X e Y*, se X é progenitor de Y, então Y é filho de X.

Clásulas Prolog deste tipo são denominadas, **regras**. Há uma diferença significativa entre fatos e regras. Uma **fato** sempre é verdadeiro, enquanto **regras** especificam algo que pode ser verdadeiro, se algumas condições são satisfeitas. As regras possuem:

- Uma parte de **conclusão**, (o lado esquerdo da clásula), e

- Uma parte de **condição**, (o lado direito da clásula).

O símbolo `<-` significa **se** e separa a clásula em conclusão, ou *cabeça* da clásula, e condição, ou *corpo* da clásula. Note que a condição expressa pelo corpo da clásula deve ser verdadeira, para que como uma consequência lógica, seja também válida na cabeça da regra.

Observação: nos códigos Prolog será usado o símbolo $:-$ para representar `<-`.

Ainda possível novas relações ao programa. Por exemplo, a especificação da relação *mãe*, entre dois objetos do domínio pode ser escrita por meio da seguinte declaração lógica:

- Para todo X e Y
  - X é mãe de Y, se
  - X é progenitor de Y e
  - X é do sexo feminino.

Essa declaração lógica pode ser representada pela seguinte clásula (Prolog):

```
mae(X, Y) <-
progenitor(X, Y),
feminino(X).
```

Note que a vírgula entre as duas condições representa a operação de *conjunção*. Logo, para satisfazer o corpo da regra, ambas as condições devem ser satisfeitas.

A relação *avô*, apresentada anteriormente na Fig. 5, pode ser definida em Prolog:

```
avo(X, Z) <-
progenitor(X, Y),
progenitor(Y, Z).
```

Dando continuidade, agora é criada uma nova relação *irmã*, a qual segue a declaração lógica dada abaixo:

- Para todo X e Y
  - X é irmã de Y, se
  - X e Y possuem um progenitor comum e
  - X é do sexo feminino.

Essa declaração lógica pode ser representada pela seguinte clásula (Prolog):

```
irma(X, Y) <-
progenitor(Z, X),
progenitor(Z, Y),
feminino(X).
```

Em relação a esta seção, salientam-se as seguintes características da linguagem:

- Programas Prolog podem ser estendidos pelas simples adição de novas clásulas;
- As clásulas Prolog podem ser de três tipos: **fatos**, **regras** e **consultas**;
- Os fatos declaram elementos que são incondicionalmente verdadeiros;
- As regras declaram elementos que podem ser verdadeiros ou não, isso depende da satisfação das respectivas condições;

- Por meio de consultas é possível interrogar o programa acerca de quais elementos são verdadeiros;
- As clásulas Prolog são constituídas por uma *cabeça* e um *corpo*. O corpo é uma lista de objetivos separados por vírgulas que devem ser interpretadas como operações de **conjunção**;
- Fatos são clásulas que só possuem *cabeça*, enquanto que as consultas só possuem *corpo* e, por sua vez, as regras possuem *cabeça* e *corpo*;
- Ao longo de uma computação, uma variável pode ser substituída por outro objeto. Diz-se que a variável está instanciada;
- As variáveis são ditas como *universalmente quantificadas* nas regras e nos fatos. E, por sua vez são ditas como *existencialmente quantificadas* nas consultas.

2.3 Recursão

A recursão tem um papel fundamental em programas Prolog. Aqui será visto como a recursão pode ser usada no programa da árvore genealógica. Para tal será criada a relação *antepassado*, que é definida a partir da relação *progenitor*. Essa definição é expressa por meio de duas regras. A primeira determina os antepassados *diretos*, ao passo que a segunda regra estabelece os antepassados *indiretos*. A Fig. 10 ilustra esses dois tipos de antepassados.

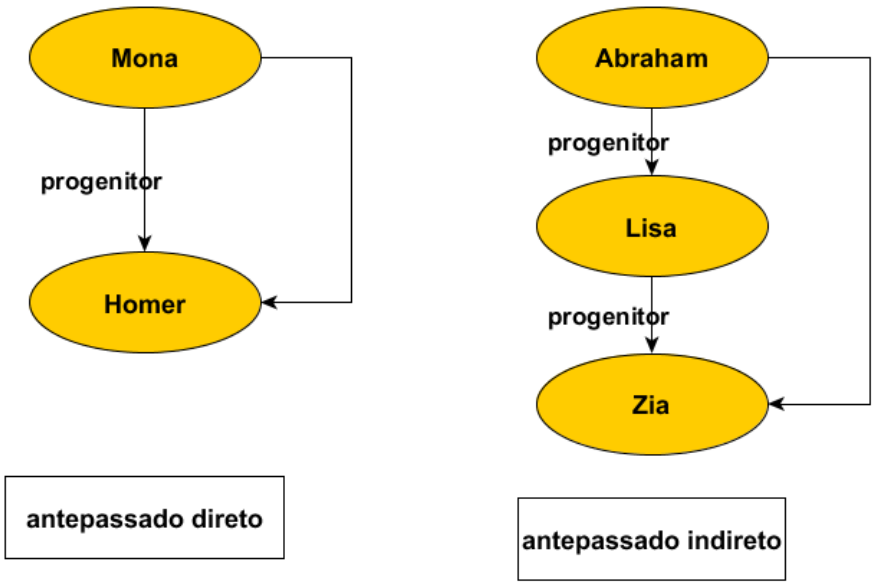

Figura 10: Exemplos da relação *antepassado*

A primeira regra, que define os antepassados diretos é formulada da seguinte maneira:

- Para todo X e Z
  - X é antepassado de Z se
  - X é progenitor de Z.

Essa declaração lógica pode ser representada pela seguinte clásula (Prolog):

```
antepassado(X, Z) <-
progenitor(X, Z).
```

A segunda regra é um pouco mais complicada, pois deve considerar que é possível ter uma cadeia de progenitores. Uma primeira tentativa seria escrever uma clásula para cada possibilidade. Isso levaria ao seguinte conjunto de clásulas:

```
antepassado(X, Z) <-
progenitor(X, Z),
progenitor(Y, Z).

antepassado(X, Z) <-
progenitor(X, Y1),
progenitor(Y1, Y2),
progenitor(Y2, Z).

antepassado(X, Z) <-
progenitor(X, Y1),
progenitor(Y1, Y2),
progenitor(Y2, Y3),
progenitor(Y3, Z).
...
```

Essa solução não é adequada é também é limitada. Porém, há uma formulação elegante e correta para a relação *antepassado*, a a qual não apresenta limitação. Essa solução é construída por meio da **recursão**, conforme a seguinte definição:

- Para todo X e Z
  - X é antepassado de Z se
    - existe um Y tal que,
    - X é progenitor de Y e
    - Y é antepassado de Z.

Essa declaração lógica pode ser representada pela seguinte clásula (Prolog):

```
antepassado(X, Z) <-
progenitor(X, Y),
antepassado(Y, Z).
```

Assim é possível construir um programa Prolog para a relação antepassado composto pelas duas regras mencionadas anteriormente. Agregando as duas regras tem-se o seguinte:

```
antepassado(X, Z) <-
progenitor(X, Z).
antepassado(X, Z) <-
progenitor(X, Y),
antepassado(Y, Z).
```

Esse tipo de definição é denominada **recursiva**, veja a ilustração na Fig. 11. O uso da recursão é, de fato, uma das principais características herdadas da lógica pela linguagem Prolog.

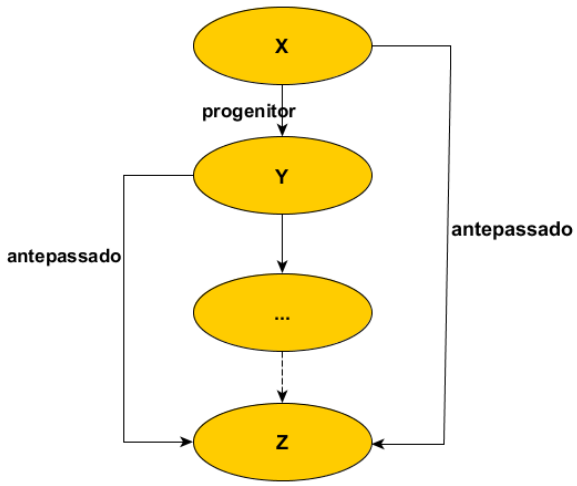

Figura 11: Formulação recursiva da relação *antepassado*

A forma final do programa da árvore genealógica é apresentada no Código 1

```
/* Arvore genealogica da familia */

progenitor(abraham, herbert).      % Abraham é progenitor de Herbet.
progenitor(mona, herbert).
progenitor(abraham, homer).
progenitor(mona, homer).
progenitor(homer, lisa).
progenitor(homer, bart).
progenitor(homer, maggie).
progenitor(marge, lisa).
progenitor(marge, bart).
progenitor(marge, maggie).
progenitor(lisa, zia).
progenitor(milhouse, zia).

masculino(abraham).                % Abraham é do sexo masculino.
masculino(herbert).
masculino(homer).
masculino(bart).
masculino(milhouse).

feminino(mona).                    % Mona é do sexo feminino.
feminino(marge).
feminino(lisa).
feminino(maggie).
feminino(zia).

filho(Y, X) :-                     % Y é filho de X.
          progenitor(X, Y).

mae(X, Y) :-                       % X é mãe de Y.
        progenitor(X, Y),
        feminino(X).

avo(X, Z) :-                       % X é avô de Z.
        progenitor(X, Y),
        progenitor(Y, Z),
        masculino(X).

irma(X, Y) :-                      % X é irmã de Y.
            progenitor(Z, X),
            progenitor(Z, Y),
            feminino(X).
```

```
45 antepassado(X, Z) :-              % predicado 1: caso base.
           progenitor(X, Z).
47 antepassado(X, Z) :-              % predicado 2: caso recursivo.
           progenitor(X, Y),
49         antepassado(Y, Z).
```

Listing 1: Programa Prolog 1

Este programa define diversas relações: *progenitor*, *masculino*, *feminino*, *antepassado*, entre outras. Observe também a utilização de comentários. Em Prolog existem dois tipos de comentários:

```
% comentário 1
/* comentário 2 */
```

2.4 Consultas

Uma consulta em Prolog é sempre uma sequência composta por um ou mais subobjetivos. Para obter a resposta, o sistema Prolog tenta satisfazer todos os objetivos que compõem a consulta, interpretando-os como uma conjunção.

Satisfazer um objetivo significa demonstrar que esse objetivo é verdadeiro, assumindo que as relações que o implicam são verdadeiras no contexto do programa. Se a questão também contém variáveis, o sistema Prolog deverá encontrar ainda os objetos particulares que, atribuídos às variáveis, satisfazem a todos os sub-objetivos propostos na consulta. A particular instanciação das variáveis com os objetos que tornam o objetivo verdadeiro é então apresentada ao usuário. Se não for possível encontrar, no contexto do programa, nenhuma instanciação comum de suas variáveis que permita derivar alguns dos sub-objetivos propostos então a resposta será "não".

Uma visão apropriada da interpretação de um programa Prolog em termos matemáticos é a seguinte: o sistema Prolog aceita fatos e regras como um conjunto de axiomas e a consulta do usuário como um teorema a ser provado. A tarefa do sistema é demonstrar que o teorema pode ser provado com base nos axiomas representados pelo conjunto das claúsulas que constituem o programa. Essa visão é ilustrada por meio de um exemplo clássico da lógica de *Aristóteles*.

Tendo os axiomas:

Todos os homens são falíveis.
Sócrates é um homem.

Um teorema que deriva logicamente desses dois axiomas é:

Sócrates é falível.

O primeiro axioma pode ser reescrito, como: *Para todo X, se X é um homem, então X é falível.* Nessa mesma linha de raciocínio o exemplo pode ser escrito em Prolog da seguinte forma:

```
falivel(X) <--
homem(X).

homem(Socrates).
```

E conseguimos fazer a consulta para descobrir que é falível.

```
?- falivel(X).
X = sócrates.
```

2.5 Significado de programas Prolog

Um programa Prolog possui três interpretações semânticas básicas:

- Interpretação Declarativa;
- Interpretação Procedimental;
- Interpretação Operacional.

Na interpretação declarativa entende-se que as clásulas que definem o programa descrevem uma teoria de *primeira ordem*. Na interpretação *procedimental*, as clásulas são vistas como entrada para um método de prova. Enquanto, que na interpretação *operacional* as clásulas são vistas como comandos para um procedimento particular de prova por refutação.

Essas três interpretaçãos da linguagem evidenciam o poder de expressão do Prolog. Além disso, o programador é encorajado a considerar a semântica declarativa dos programas de forma independente dos significados procedimental e operacional. Uma vez que os resultados do programa são considerados, em princípio, pelo seu significado declarativo.

3 Sintaxe e Semântica

A linguagem Prolog possui um alfabeto específico para definição de átomos, números e variáveis, como é apresentado na próxima seção.

3.1 Objetos

3.1.1 Alfabeto Prolog

- Símbolos de Pontuação: . ′ ( ) ”
- Símbolos lógicos: , (conjunção)
  ; (disjunção)
  $<-$ (implicação)
- Letras: $a, b, c, \ldots z$, $A, B, C, \ldots Z$
- Dígitos: $0, 1, 2, \ldots 9$
- Símbolos especiais: $+, - * / \langle \rangle =: \_$

3.1.2 Átomos e Números

Os **átomos** podem ser construídos de três diferentes maneiras:

1. Como cadeias de letras e/ou dígitos, podendo conter o símbolo _ (sublinhado). Necessariamente deve iniciar com uma letra **minúscula**.

   Exemplos:

   ```
   socrates
   nil
   x78
   x_y
   buscarObjetivo
   a_b_1_2
   ```

2. Como cadeiras de símbolos especiais. *Exemplos*:

   ```
   <----------->
   ::=
   =/=
   ==========>
   ++++++
   ```

3. Como cadeias de caracteres quaisquer, podendo inclusive incluir espaços em branco, desde que devidamente delimitados por apóstrofos (′). *Exemplos*:

   ```
   'Gentzen'
   'representação de conhecimento'
   'máquina de inferência lógica'
   '11 de outubro de 1908'
   'olá, boa tarde!'
   ```

Os **números** utilizados em Prolog compreendem aos números inteiros e reais.
Exemplos:

```
1
1812
0
-274
3.14159
0.0000000028
-273.16
```

Números reais são pouco usados em Prolog. Isso ocorre por que Prolog é uma linguagem orientada ao processamo simbólico, não-numérico. Na computação simbólica, números inteiros são usados com frequência para contar os itens de uma lista, por exemplo. Porém, a necessidade de números reais é bem limitada.

3.1.3 Variáveis

Uma variável em Prolog é definida por cadeias de letras, dígitos e do símbolo sublinhada (_). Note que necessariamente uma variável deve iniciar com letra **maiúscula**.

Exemplos de variáveis:

```
X
Resultado
Objeto2
Lista_de_associado
_var35
_194
```

3.1.4 Estrutura

Objetos estruturados, ou apenas **estruturas**, são objetos que possuem vários componentes. Os próprios componentes podem ser também outras estruturas.

Por exemplo, uma data pode ser definida como uma estrutura de três componentes: *dia*, *mês* e *ano*. Mesmo que sejam formadas por diversos componentes as estruturas são tratadas no programa como objetos simples. Para combinar os componentes em uma estrutura é necessário utilizar um **functor**.

Um **functor** é um símbolo funcional (um nome de função) que permite agrupar diversos objetos em um único objeto estruturado. Um functor referente ao exemplo menciondo anteriormente é *data*, e pode ser representado em Prolog, como:

data(11, outubro, 1908)

Todos componentes da estrutura são constantes, dois valores inteiros e um átomo. Porém, é possível usar variáveis ou até mesmo outras estruturas. Por exemplo, um dia qualquer do mês de setembro de 2017.

data(Dia, outubro, 1908)

Observe que *Dia* é uma variável e pode ser instanciada para qualquer objeto em algum ponto da execução.

Sintaticamente todos os objetos em Prolog são denominados *termos*. O conjunto de *termos* em Prolog, ou simplesmente *termos*, é o menor conjunto que satisfaz às seguintes condições:

- Toda constante é um termo;
- Toda variável é um termo;
- Se $t_1, t_2, \ldots, t_n$ são termos e f é um átomo, então $f(t_1, t_2, \ldots, t_n)$ também é um termo, onde o átomo f desempenha o papel de um símbolo funcional n-ário. Diz-se que a expressão $f(t_1, t_2, \ldots, t_n)$ é um termo funcional Prolog.

3.2 Unificação

A operação mais importante entre dois termos Prolog é denominada **unificação**. A unificação pode produzir resultados relevantes em programas. Tendo dois termos é dito que eles são unificados, caso:

1. Os termos são idênticos,
2. ou as variáveis de ambos os termos podem ser instanciadas com objetos de maneira que, após a substituição de variáveis por esses objetos, os termos se tornam idênticos.

Por exemplo, os termos: `data(D, M, 2017)` e `data(X, Y, 17)` não unificam, bem como não unificam os termos `data(X, Y, Z)` e `ponto(X, Y, Z)`.

A unificação é um processo que toma dois termos como entrada e verifica se eles podem ser unificados. Se os termos não unificam, então é dito que o processo *falha*. Se os termos unificam, então o processo é bem sucedido e as variáveis e os termos que participam do processo são instanciados com os valores encontrados para os objetos, de modo que os dois termos da unificação se tornam idênticos. Novamente considerando a unificação de duas datas. O requisito para que essa operação ocorra é informado ao sistema Prolog pela seguinte consulta, usando o operador =.

```
?- data(D, M, 2017) = data(X, maio, A)
```

Para obter a unificação é necessário a seguinte instanciação: `D = X, M = maio, A = 2017`.

Contudo, ainda existem outras instanciações que igualmente tornam os termos idênticos. Outros dois exemplos, são:

```
D = 1, X = 1, M = maio, A = 2017
D = terceiro, X = terceiro, M = maio, A = 2017
```

As regras gerais que determinam se dois termos S e T unificam são as seguintes:

- Se S e T são constantes, então S e T unificam somente se ambos representam o mesmo objetivo;
- Se S é uma variável e T é *qualquer coisa*, então S e T unificam com S instanciada com T. Inversamente, se T é uma variável, então T o instanciada com S;
- Se S e T são estruturas, unificam somente se: (**i**) S e T têm o mesmo functor principal; e (**ii**) todos os seus componentes correspondentes também unificam. A instanciação resultante é determinada pela unificação dos componentes.

Essa última regra pode ser exemplificada pelo processo de unificação dos termos

```
triangulo(ponto(1,1), A, ponto(2,3))
triangulo(X, ponto(4,Y), ponto(2,Z))
```

O processo de unificação inicia pela raiz, ou o functor principal. Como ambos os functores (*triangulo*) unificam, o processo continua com a unificação dos argumentos, onde a unificação dos pares de argumento correspondente ocorre. Assim, o processo completo pode ser visto como a seguinte sequência de operações de unificação simples:

```
triangulo = triangulo
ponto(1,1) = X
A = ponto(4,Y)
ponto(2,3) = ponto(2,Z)
```

O processo de unificação completo é bem sucedido porque todas as unificações na sequência anterior também o são. A instanciação resultante é:

```
X = ponto(1,1)
A = ponto(4,Y)
Z = 3
```

4 Ambiente para execução de programas Prolog

Na Seção 9 encontram-se links para ferramentas do Prolog. No material aqui desenvolvido será utilizada a ferramenta **SWI-Prolog**, especificamente o compilador em conjunto com o editor.

Dessa forma tem-se um Ambiente Prolog para edição e execução de programas.

- Para reiniciar o ambiente e deixá-lo preparado para executar consultas é utilizada a função **CTRL + F9**.
- Para compilar um programa é utilizado **F9**.

A linguagem Prolog é do paradigma declarativo, onde não se utilizam variáveis com atribuição explícita de valores. Logo, para fazer operações simples de soma, por exemplo, faz-se necessário utilizar mecanismos um pouco diferentes. Considerando os exemplos a seguir, onde são criadas duas regras: **soma1**, **soma2**:

```
soma1(X,Y) :-
          K is X + Y,
          print(K).

soma2(X,Y,K) :-
          K is X + Y.
```

Observe na regra **soma1** utiliza-se o comando com a palavra-chave **is**, a qual possibilita a execução da operação de soma. Sendo que o resultado da soma é associado a variável K.

Mas, perceba que na regra **soma2** não é necessário imprimir explicitamente o valor de K. A variável K é dos argumento de **soma2**. Portanto, essa regra **soma2** seria uma recomendação adequada para utilizar a operação de soma quando é necessário usar o resultado para uma outra regra Prolog.

5 Utilização de Listas em Prolog

Na linguagem Prolog a utilização de listas é fundamental. As listas em Prolog podem ser representadas como árvores. A analogia entre listas aninhadas e árvores é importante para o entendimento de algumas operações em listas.

A sintaxe de listas em Prolog é uma variante da sintaxe da linguaguem *Lisp*.

Nesta seção será vista a representação de listas, codificação em Prolog e diferentes operações em listas.

5.1 Representação de Listas

Listas são estruturas simples de dados. Uma lista é uma sequência de qualquer número de itens. Por exemplo, uma lista com algumas cidades da região dos *Campos Gerais*: *Ponta Grossa*, *Castro*, *Carambeí* e *Irati*. Essa lista pode ser escrita em Prolog:

```
[pontaGrossa, castro, carambei, irati]
```

Essa é apenas a aparência externa das listas. Já que os objetos estruturados em Prolog são na realidade árvores. Para representar listas é necessário considerar dois casos: a lista vazia e a lista não-vazia. No primeiro caso, a lista é representada simplesmente como um átomo, `[]`. No segundo caso, a lista é constituída de dois componentes: uma *cabeça* e um *corpo*. Por exemplo, na lista dos *Campos Gerais*, *pontaGrossa* é a cabeça da lista, enquanto o corpo da lista é: `[castro, carambei, irati]`.

Para simplificar a representação de listas utilizam-se vírgulas para separar cada item de uma lista, sendo que uma lista deve iniciar e fechar com colchetes. Seguindo essa notação, um termo da forma `[H | T]` é identificado como uma lista de cabeça H e corpo T. Salienta-se que o corpo de uma lista é sempre outra lista, mesmo que uma lista vazia. Aqui seguem alguns exemplos para esclarecer tais conceitos e representação.

`[X | Y]` ou `[X | [Y | Z]]` unificam com `[a, b, c, d]`

`[X, Y, Z]` não unifica com `[a, b, c, d]`

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

As consultas ilustradas na Fig. 12 ajudam a compreender como identificar os elementos numa lista, bem como a unificação de listas.

```
For built-in help, use ?- help(Topic). or ?- apro

?- [X | Y] = [a, b, c].
X = a,
Y = [b, c].

?- [X, Y, Z] = [a, b, c, d].
false.

?  [X | [Y | Y]] = [a, b, c, d].
false.

?- [X | [Y | Z]] = [a, b, c, d].
X = a,
Y = b,
Z = [c, d].

?-
```

Figura 12: Exemplos de unificação de listas

5.2 Operações em Listas

Estruturas em lista podem ser definidas e transformadas de diversas maneiras. Aqui serão representadas algumas operações em listas. Para maior clareza é utilizada a notação:

símbolo_predicativo/aridade

para a identificação de predicados. Por exemplo, **cidades/3** denota a relação denominada **cidades** com três argumentos. Nome e aridade são os elementos necessários para a identificação de um predicado.

5.2.1 Construção de Listas

A primeira operação necessária para a manipulação de listas é naturalmente a construção de listas a partir de seus elementos básicos: a cabeça e o corpo. Essa relação pode ser escrita em um único fato:

```
cons(X, Y, [X | Y]).
```

Por exemplo:

```
?- cons(a, b, Z).
Z = [a | b]
```

Durante a unificação a variável X é instanciada com a, Y com b e Z com `[X | Y]`, que por sua vez é instanciada com `[a | b]`, devido aos valores de X e Y. Se X é um elemento e Y é uma lista, então `[X | Y]` é uma nova lista com X como primeiro elemento. Por exemplo:

```
?- cons(a, [b, c], Z).
Z = [a, b, c]

?- cons(a, [], Z).
Z = [a]
```

A generalidade da unificação permite a definição de um resultado implícito:

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

onde identifica-se em X os elementos do corpo da lista.

Agora, considerando um outro exemplo, onde o primeiro argumento é uma lista com três elementos e o segundo elemento é uma lista com dois itens. Note que o resultado não será uma lista com cinco elementos, mas uma lista onde a cabeça da lista é uma lista com três itens.

```
?- cons([a, b, c], [d, e], Z).
Z = [[a, b, c], d, e]
```

Portanto, o predicado **cons/3** não é usado para concatenar duas listas em uma terceira lista. Adiante será vista a operação de concatenação de listas.

5.2.2 Ocorrência de elementos

Aqui será uma implementada uma relação de pertinência que estabelece se determinado objeto é membro de uma lista.

```
membro(X, L)
```

onde X é um objeto e L uma lista. O objetivo `membro(X, L)` é verdadeiro se X ocorre em L. Por exemplo, as seguintes declarações são verdadeiras:

```
membro(b, [a, b, c])
membro([b, c], [a, [b, c], d])
```

Porém, a declaração abaixo é falsa:

```
membro(b, [a, [b, c]])
```

O programa que define a relação **membro/2** tem como fundamento a seguinte afirmação:

- X é membro de L, se:

  1. X é cabeça de L, ou
  2. X é membro do corpo de L.

Em Prolog é possível representar essa relação por meio de duas clásulas. A primeira é um fato que estabelece a primeira condição: X é membro de L, se X é a cabeça de L. A segunda é definida por meio de uma chamada recursiva, que diz que X ainda pode ser membro de L, desde que seja membro do corpo de L.

```
membro(X, [X | C]).

membro(X, [Y | C]) :-
membro(X, C).
```

Observa-se que o corpo da lista na primeira clásula é sempre um resultado sem interesse, da mesma forma que cabeça da lista na segunda clásula. Portanto, é possível usar variáveis anônimas e escrever o predicado de outra forma:

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

membro(X, [_ | C]) :-
membro(X, C).
```

5.2.3 Concatenação

Para a concatenação de duas listas quaisquer, resultando em uma terceira lista, faz-se necessário definir a relação **conc/3**.

```
conc(L1, L2, L3)
```

onde L_1 e L_2 são duas listas e L_3 é a concatenação resultante. Por exemplo:

```
conc([a, b], [c, d], [a, b, c, d])
```

Dois casos são considerados para a definição de **conc/3**, os quais dependem do primeiro argumento L_1:

1. Se o primeiro argumento é uma lista vazia, então o segundo e o terceiro argumento devem ser a mesma lista. Denominando tal lista de L, tem-se a seguinte representação:

   ```
   conc([], L, L).
   ```

2. Se o primeiro argumento de **conc/3** é uma lista não vazia, então é porque ela possui uma cabeça e um corpo e pode ser denotada por `[X | L1]`.

   A concatenação de `[X | L1]` com uma segunda lista L_2, produzirá uma terceira lista com a mesma cabeça X da primeira e um corpo L_3 que é a concatenação do corpo de L_1 com toda lista L_2.

O programa completo para concatenação de listas descrevendo o predicado **conc/3** é definido como segue:

```
conc([], L, L).

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

Exemplos simples de utilização de **conc/3** são apresentados abaixo:

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

?- conc([a, [b, c], d], [a, [], b], L).
L = [a, [b, c], d, a, [], b]

?- conc([a, b], [c | R], L).
L = [a, b, c | R]
```

O programa **conc/3** pode ser usado em inúmeras aplicações. Por exemplo, no sentido inverso ao que foi projetado, pode ser empregado para decompor uma lista em duas partes:

```
?- 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 = [];
```

Em outro exemplo identifica-se a possibilidade de encontrar um determinada padrão em uma lista. No caso, é possível encontrar os meses antes e depois de um determinado mês, no caso o mês de maio.

```
?- M = [jan, fev, mar, abr, mai, jun, jul, ago, set, out, nov, dez],
conc(Antes, [mai | Depois], M).
Antes = [jan, fev, mar, abr]
Depois = [jun, jul, ago, set, out, nov, dez]
```

Um outro exemplo similar serve para apagar de uma lista todos elementos que seguem a um determinado padrão. No exemplo a seguir, retira-se da lista dos dias da semana, a sexta e os demais dias que a seguem.

```
?- conc(Trab, [sex | _], [seg, ter, qua, qui, sex, sab, dom]).
Trab = [seg, ter, qua, qui]
```

O próprio predicado visto anteriormente, **membro/2** pode ser reescrito por meio do uso de **conc/3**.

```
novoMembro(X, L) :-
conc(_, [X | _], L).
```

Essa cláusula diz que X é membro de L, se L pode ser decomposta em duas outras listas, onde a cabeça da segunda lista é o elemento X.

Ainda é possível testar se existem diferentes ocorrências de um mesmo elemento numa lista L, por exemplo fazendo a consulta:

```
?- novoMembro(b, [a, b, c, d, b]).
```

E para acompanhar melhor a quantidade de ocorrência de um elemento é possível usar a função **print**, como segue.

```
novoMembro(X, L) :-
conc(_, [X | _], L),
print(X).
```

E refazendo a consulta anterior, tem-se:

```
?- novoMembro(b, [a, b, c, d, b]).
b
true.
b
true.
```

5.2.4 Remoção e Inserção

A remoção de um elemento X de uma lista L pode ser feita por meio da relação **remover/3**.

```
remover(X, L, L1)
```

onde L_1 é (quase) a mesma lista L, porém com o elemento X removido. A relação **remover/3** é definida por meio de dois casos:

1. Se X é a cabeça da lista L, então L_1 é o corpo;
2. Se X está no corpo de L, então L_1 é obtida removendo X desse corpo.

O programa Prolog correspondente segue abaixo:

```
remover(X, [X | C], C).

remover(X, [Y | C], [Y | D]) :-
remover(X, C, D).
```

A relação **remover/3** é não-determinística. Logo, se há diversas ocorrências de X em L, é possível retirar cada ocorrência por meio do mecanismo de *backtracking* do Prolog. Em cada execução de **remover/3** é excluída uma ocorrência repetida de X, mantendo as demais ocorrências na lista resultante.

```
?- remover(a, [a, b, a, a], L).
L=[b, a, a];
L=[a, b, a];
L=[a, b, a];
```

A relação **remover/3** irá falhar, caso a lista L não tiver nenhuma ocorrência do elemento X. Essa relação pode ainda ser usada no sentindo inverso, no caso para para inserir um novo item em qualquer lugar da lista. Por exemplo, é possível formular a questão: *Qual é a lista L da qual retirando-se* **a**, *obtem-se a lista* `[b, c, d]`*?*

```
?- remover(a, L, [b, c, d]).
L=[a, b, c, d];
L=[b, a, c, d];
L=[b, c, a, d];
L=[b, c, d, a];
```

De modo geral é possível inserir um elemento X em algum lugar de uma lista L, resultando em uma nova lista L_1, tendo o elemento X inserido na posição desejada. Para tal é definida a seguinte cláusula:

```
inserir(X, L, L1) :-
remover(X, L1, L).
```

Observe o resultado da seguinte consulta:

```
?- inserir(x, [y, z, w], NovaLista).
NovaLista = [x, y, z, w];
NovaLista = [y, x, z, w];
NovaLista = [y, z, x, w];
NovaLista = [y, z, w, x];
```

onde o elemento novo x é inserido de forma *não-derterminística*, ou seja o resultado da operação de inserção considera quatro diferentes posições para inserção de x.

5.2.5 Inversão

A relação **inverter/3** é responsável por organizar seus elementos na ordem inversa. Assim, como apresentado nos seguintes exemplos:

```
inverter([a, b, c], [c, b, a]).
inverter([], []).
inverter([a, [b, c], d], [d, [b, c], a]).
```

Uma abordagem simples de fazer a operação de inversão é a seguinte:

1. Tomar o primeiro elemento da lista;
2. Inverter o restante;
3. Concatenar a lista formada pelo primeiro elemento ao inverso do restante.

Em Prolog, tem-se o seguinte predicado **inverter/2**:

```
inverter([], []).

inverter([X | Y], Z) :-
inverter(Y, Y1),
conc(Y1, [X], Z).
```

Outra maneira de fazer a inversão é utilizar um predicado auxiliar, **aux/3**. Essa versão alternativa é mais eficiente que a anterior e denominada **inverterAux/2**.

```
inverterAux(X, Y) :-
aux([], X, Y).

aux(L, [], L).
aux(L, [X | Y], Z) :-
aux([X | L], Y, Z).
```

5.2.6 Sublistas

Aqui é apresentada uma nova relação **sublista/2** que possui como argumentos uma lista S e uma lista L, tais que S ocorre em L como sublista. A seguinte afirmação é verdadeira:

```
sublista([c, d, e], [a, b, c, d, e, f])
```

Porém, é falso declarar que:

```
sublista([c, e], [a, b, c, d, e, f])
```

A relação **sublista/2** pode se basear na mesma ideia da definição do predicado **membro/2**, exceto que a relação para sublista é mais genérica, podendo ser formulada da seguinte maneira:

- S é sublista de L se:

    1. L pode ser decomposta em duas listas, L_1 e L_2, e
    2. L_2 pode ser decomposta em S e L_3.

A relação **conc/3** pode ser usada para a decomposição de listas. Assim, a formulação para sublistas pode ser escrita da seguinte forma:

```
sublista(S, L) :-
conc(L1, L2, L),
conc(S, L3, L2).
```

Um exemplo de aplicação dessa operação é para descobrir todas sublistas de uma dada lista.

```
?- sublista(S, [a, b, c]).

S = [];
S = [a];
S = [a, b];
S = [a, b, c];
S = [b];
S = [b, c];
S = [c];
```

5.2.7 Permutação

Ainda outra operação que pode ser definida para listas. A operação de permutação, a qual é criada pela relação **permutar/2**, cujos argumentos são duas listas tais que cada uma é permutação da outra. A intenção é permitir a geração de todas as permutações possíveis de uma dada lista utilizando o mecanismo de *backtracking*. Como, por exemplo:

```
?- permutar([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];
```

A relação **permutar/2** é estabelecida em dois casos, dependendo da lista a ser permutada:

1. Se a primeira lista é vazia, então a segunda também é;

2. Se a primeira lista é não vazia, então possui a forma `[X | L]` e uma permutação de tal lista pode ser construída primeiro permutando L para obter L_1 e depois inserindo X em qualquer posição de L_1.

A relação **permutar/2** em Prolog é definida da seguinte forma:

```
permutar([], []).

permutar([X | L], P) :-
permutar(L, L1),
inserir(X, L1, P).
```

5.3 Outras operações

Aqui ainda são apresentadas outras operações complementares para listas.

Tamanho de uma lista

A relação **tamanho/2**, representada por `tamanho(L, T)` será verdadeira T for o número de elementos existentes em L:

```
tamanho([], 0).

tamanho([_ | R], N) :-
                    tamanho(R, N1),
                    N is N1 + 1.
```

Com isso é possível fazer consultas, como:

```
?- tamanho([a, b, c, d, e], X).
X=5
```

Operações de soma e produto

Ainda é possível criar predicados específicos para cálculo de soma e produto por meio das relações **soma/2** e **produto/2**. No que o cálculo de produto ainda utiliza um predicado auxiliar **prod/2**.

```
soma([], 0).

soma([X | Y], S) :-
                 soma(Y, R),
                 S is R + X.

produto([], 0).
produto([X], X).
produto(L, P) :-
              prod(L, P).

prod([], 1).
prod([X | Y], P) :-
                 prod(Y, Q),
                 P is Q * X.
```

Abaixo estão alguns exemplos de consultas para as operações de soma e produto.

```
?-soma([1,2,3,4], X).
x=10

?-soma([1,12,16], S).
S=29

?-produto([], X).
x=0

?-produto([1,5,6,4], R).
x=120
```

Intersecção

O predicado **intersec/3** calcula a intersecção entre duas listas, onde o resultado é dado em uma terceira lista.

```
intersec(_, _, []).

intersec([X | Y], L, [X | Z]) :-
                              membro(X, L),
                              intersec(Y, L, Z).

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

Um exemplo de utilização deste predicado é dado a seguir.

```
?-intersec([a, b, c, d], [aa, b, d], L).
L=[b, d]
```

Seleção de elementos

Aqui são apresentados dois predicados que serão utilizados para identificação e seleção de elementos específicos em listas.

O predicado **enehsimo/3** é determinado para selecionar exatamente o enésimo elemento de uma lista.

```
enehsimo(1,X,[X | _]).

enehsimo(N,X,[_ | Y]) :-
                      enehsimo(M, X, Y),
                      N is M + 1.
```

Alguns exemplos de utilização desse predicado são apresentados a seguir:

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

?-enehsimo(N,b,[a,b,c,d]).
N=2
```

Agora é possível utilizar o predicado **enehsimo/3** (definido anteriormente) para construir outro predicado capaz de filtrar elementos de uma lista, a partir de uma lista de seleção de elementos. Isso é realizado pelo predicado **seleciona/3**.

```
seleciona([], _, []).

seleciona([M | N], L, [X | Y]) :-
                                enehsimo(M, X, L),
                                seleciona(N, L, Y).
```

Abaixo seguem possíveis consultas para este predicado de seleção de elementos.

```
?-seleciona([2,4], [a, b, c, d, e], L).
L=[b, d]
```

6 Outros comandos e operador Cut

Nesta seção serão visto alguns outros comandos Prolog que podem auxiliar na escrita de programas, no caso são comandos tipicamente empregados em linguagens procedurais para leitura, escrita e repetição. Além disso é apresentado um operador específico da linguagem, denominado CUT.

6.1 Comandos básicos

Aqui são apresentados alguns comandos básicos da linguagem que podem auxiliar na elaboração de exemplos e exercícios.

Comando `write`

No predicado **proc/1** observe a utilização do comando `write` para escrever na saída o valor da variável Y.

```
proc(X) :-
       Y is X+3,
       write(Y).
```

Fazendo a seguinte consulta:

```
proc(5).
8
```

Comandos `read` e `nl`

Ademais, outro comando que pode ser usado, neste caso para leitura de informações é o comando `read`. No predicado abaixo (**proc2/1**), além do comando `read`, utiliza-se o comando `nl`, o qual é usado para inserção de uma nova linha (do inglês, *new line*).

```
proc2(A):-
          write('Informe o segundo valor: '), nl,
          read(B),
          R is A+B,
          write('o resulta eh: '),
          write(R).
```

Fazendo a seguinte consulta:

```
?- proc2(15).
Informe o segundo valor:
13
o resultado eh: 28
```

6.2 Operador CUT

O sistema Prolog tem na execução de seus programas a características de percorrer a árvore de busca procurando por alguma solução de forma exaustiva. A forma de percorrer a árvore por soluções alternativas é dita *backtracking*.

Contudo, em algumas vezes esse mecanismo de busca exaustiva por soluções pode gerar uma quantidade indesejável de respostas. Quando isso ocorrer, faz-se necessário empregar o operador CUT, o qual permite que algumas ramificação na árvore de busca não sejam inspecionadas pelo mecanismo de *backtracking*.

O operador CUT tem algumas vantagens:

- O programa irá executar mais rapidamente, porque não irá desperdiçar tempo tentando satisfazer objetivos que não vão contribuir para a solução desejada;
- Igualmente, a memória será economizada, uma vez que determinados pontos de *backtracking* não necessitam ser armazenados para exame posterior.

E, igualmente, desvantagens:

- O operador quebra a pureza semântica da lógica e deve ser utilizado com parcimônia.

Este operador pode ser usada de diferentes formas, por exemplo:

1. Unificação de padrões, de forma que quando um padrão é encontrado, os outros padrões possíveis são descartados;
2. Para eliminar da árvore de pesquisa soluções alternativas, considerando aquelas casos onde apenas uma solução já é suficiente;
3. Para encerrar a pesquisa quando a continuação poderia conduzir a uma pesquisa infinita.

A sintaxe do operador CUT é representada por um *ponto de exclamação* !.

Exemplo de utilização

Para constatar o uso do operador CUT (!), reescreve-se o predicado **membro/2**, já visto anteriormente na Seção 5. Veja a seguir o código, note a pequena diferença no caso base da recursão, o qual agora é uma regra Prolog, ao invés de um fato.

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

Observe as diferentes consultas feitas no Prolog. Inicialmente por meio do predicado **membro/2**.

```
membroCut(X, [a,b,c]).
X = a
```

Depois usando o predicado **membroCut/2** (naturalmente, usando o operador).

```
membro(X, [a,b,c]).
X = a;
X = b;
X = c;
```

6.3 Comando Condicional

Na linguagem é possível escrever predicados com uma condição explícita, conforme apresentado no predicado **max/2**, o qual é usado para retornar o elmento máximo em uma lista de valores inteiros.

```
max([X], X).
max([X | Y], M) :-
                 max(Y, N),
                 (X > N -> M=X; M=N).
```

Consulta:

```
?- max([1,3,5,12,7,2,6,0], Res).
Res = 12;
```

6.4 Comando de Repetição

Além da recursão é possível utilizar de forma similar a linguagem imperativa, um comando de repetição. Veja o predicado definido a seguir **loop**, o qual executa leitura de listas de números (e respectivo cálculo da soma dos valores da lista) até que seja escrita a palavra **fim**.

```
%-- predicado de repetição
loop :- repeat,
        write('Entre com uma lista numérica ou <fim> para encerrar '), nl,
        read(X),
        (X=fim, ! ; soma(X,Y), write(Y), nl, fail).

soma([],0).
soma([X|Y],S):-
      soma(Y,R),
      S is R+X.
```

Consulta:

```
loop.
Entre com uma lista numérica ou <fim> para encerrar
|: [1,2].
3
Entre com uma lista numérica ou <fim> para encerrar
|: [4,5,6].
15
Entre com uma lista numérica ou <fim> para encerrar
|: fim.
```

Abaixo é apresentado um outro programa com uso de estrutura de repetição, denominado **loopInc**, o qual incremente valores menores ou igual a 10.

Note neste programa o uso de `fail`, que é um predicado específico da linguagem, o qual sempre irá falhar.

```
%-- predicado de repetição
loopInc :- repeat,
      write('Informe um valor '), nl,
      read(X),
      (X =< 10 -> (proc(X), fail) ; fim(X)).


%-- predicados auxiliares
proc(X):-
      Y is X+3,
      write(Y),
      nl.

fim(X):-
      write(X), write(' é maior do que 10').
```

Consulta:

```
?- loopInc.
Informe um valor
|: 5.
```

```
8
Informe um valor
|: 9.
12
Informe um valor
|: 11.
11 é maior do que 10
true.
.
```

7 Exemplos

Aqui são descritos alguns exemplos complementares escritos em Prolog.

No Código 2 é apresentado um exemplo que contém uma base de conhecimento com informações de robôs e o conhecimento de cada um. A partir dessas informações são criadas regras que definem quais os robôs mais aptos para a realização de diferentes missões, *Diplomática*, *Espacial* ou de *Exploração*.

```
% ----------------------------------
% Exemplo
% Agentes(robôs) & Conhecimento
% ----------------------------------

robot(r2d2).
robot(c3po).
robot(bb8).

knowledge(c3po, tradutor).
knowledge(r2d2, mensageiro).
knowledge(bb8, navegador).

diplomaticMission(X,Y) :-
                        robot(X),
                        robot(Y),
                        knowledge(X, tradutor),
                        knowledge(Y, mensageiro).

spaceMission(X,Y) :-
                    robot(X),
                    robot(Y),
                    (knowledge(X, navegador), knowledge(Y, mensageiro));
                    (knowledge(X, navegador), knowledge(Y, tradutor)).

exploreMission(X,Y) :-
                      robot(X),
                      robot(Y),
                      (knowledge(X, tradutor), knowledge(Y, mensageiro));
                      (knowledge(X, tradutor), knowledge(Y, navegador)).

allMissionSettings(X, Y) :-
                      diplomaticMission(X,Y);
                      spaceMission(X,Y);
                      exploreMission(X,Y).
```

Listing 2: Exemplo com base de conhecimento e regras

8 Exercícios

1. Escreva um programa Prolog para representar o seguinte:

   - Paco nasceu em Granada e Saramago nasceu em Azinhaga.
   - Granada fica na Espanha.
   - Azinhaga fica em Portugal.
   - Só é espanhol quem nasceu na Espanha.

2. Considerando que os arcos de um grafo expressam custos, como na Fig. 13, onde os custos são representados por números.

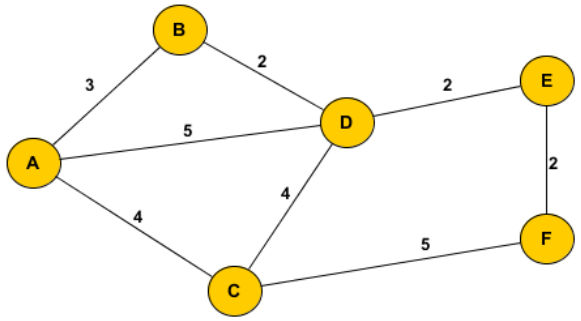

Figura 13: Representação de caminhos em um grafo

   Para descrever os arcos são usadas assertivas da forma:

$$\textbf{arco}(R, S, T)$$

   a qual significa que há um arco de custo T entre os nós R e S.

   Por exemplo, **arco**$(A, B, 3)$ representa um arco de custo 3 entre os nós A e B.

   Considera-se também que a relação **mais**(X, Y, Z) é válida quando $X + Y = Z$, o que indica uma soma de custos no grafo.

   - Tendo essa descrição, defina a relação **custo**(U, V, L) de forma a expressar que existe um caminho de custo L, entre os nós U e V.
   - Após implementar os fatos e as regras. Faça consultas ao sistema para identificar os diferentes custos e caminhos entre os nós **a** e **f**, por exemplo.
   - Além disso, identifique (por meio de consultas) quais caminhos no grafo possuem custos iguais a: **8**, **10** e **12**.

3. Escrever um programa que recebe duas listas e retorne **true**, caso ambas as listas tiverem a mesma quantidade de elementos, sendo que a primeira lista deve ter símbolos **a**, ao passo que a segunda lista apenas símbolos **b**. Caso contrário o retorno para a consulta é **falso**. O nome do predicado principal é **aNbN/2**.

   Por exemplo, para a seguinte consulta o retorno é **verdadeiro**:

```
aNbN([a,a,a,a],[b,b,b,b]).
```

   Enquanto para ambas as consultas abaixo, o retorno é **falso**.

```
aNbN([a,a,a,a],[b,b,b]).
```

```
aNbN([a,c,a,a],[b,b,5,4]).
```

4. Escreva um programa Prolog que execute a combinação de duas listas, resultando em uma terceira lista.

   Para tal, o predicado **agregarListas/3** deve funcionar como apresenta o seguinte exemplo:

   ```
   ?- agregarListas([a,b,c],[1,2,3],X).

   X = [a,1,b,2,c,3]
   ```

5. Construir soluções em Prolog para as seguintes operações aritméticas:

   (a) Subtração
   (b) Divisão

   Onde em ambas, as operações, deve ser considerado dois argumentos: uma lista com números inteiros a serem operadores, e um segundo elemento contendo o resultado da respectiva operação.

6. Construir soluções em Prolog para as seguintes operações em conjuntos:

   (a) União
   (b) Diferença

   Onde em ambas, as operações, deve ser considerado três argumentos: duas lista de entrada, contendo os dois conjuntos a serem operados, e uma terceira lista contendo o resultado da respectiva operação de conjuntos.

7. Construir um programa Prolog para calcular o elemento **mínimo** de uma lista numérica.

   Lembre-se de utilizar estrutura condicional.

8. Construir um programa que faça uma leitura de N listas numéricas, até que seja escrito *exit*. Para cada lista deve ser calculado e impresso o respectivo **tamanho** da lista.

   Lembre-se de utilizar estrutura de repetição e comandos para leitura e escrita.

9 Referências & Ferramentas

Ferramentas

- Compilador **SWI-Prolog**

  `http://www.swi-prolog.org/`

  Link para download:

  `http://www.swi-prolog.org/download/stable`

- Versão simplificada do SWI-Prolog, executada diretamente no *browser*.

  SWISH-Prolog

  `http://swish.swi-prolog.org`

- Editor **SWI-Prolog**

  `http://arbeitsplattform.bildung.hessen.de/fach/informatik/swiprolog/indexe.html`

  Link para download:

  `http://arbeitsplattform.bildung.hessen.de/fach/informatik/swiprolog/SWIPrologEditorSetup.exe`

- Ferramentas de desenvolvimento Prolog

  `http://www.swi-prolog.org/IDE.html`

Livros e Materias de Referência

- PALAZZO, L. M. *Introdução à Programação Prolog*. EDUCAT. 1997.
- BLACKBURN, P.; BOS, J.; STRIEGNITZ, K. **Learn Prolog Now**. (*livremente disponível on line*)

  `http://www.learnprolognow.org/index.php`

- Manual de Prolog

  `http://www.swi-prolog.org/pldoc/index.html`