

INSTITUTO FEDERAL

Minas Gerais

Campus Formiga

Introdução ao Prolog

Prof. Bruno Ferreira



SWI Prolog

Introdução

- Apesar das linguagens imperativas serem as mais utilizadas, linguagens de outros paradigmas são mais adequadas para resolver certos problemas.
- Um exemplo é o Prolog, essa linguagem pertence ao paradigma lógico.
- Prolog foi criada em 1972 por Colmerauer e Roussel, e é mais adequada para problemas onde é necessário descrever conhecimento, por exemplo:
 - em aplicações que realizem computação simbólica;
 - na compreensão de linguagem natural;
 - em sistemas especialistas.

Introdução

- Existem diversas implementações de Prolog, algumas das mais conhecidas são:
 - Win-Prolog;
 - Ciao Prolog;
 - YAP Prolog;
 - SWI Prolog + SWI-Prolog-Editor (**recomendado**);
 - www.swi-prolog.org
 - www.dsc.ufcg.edu.br/~aab/prolog
 - SICStus Prolog

Sintaxe SWI-Prolog

Os dados representados em Prolog podem ser um dos seguintes tipos:

- **variáveis** - iniciadas com maiúsculas ou underscore (_), seguidos de qualquer caractere alfanumérico. Somente underscore define uma variável anônima. Ex.: X, Y1, _Nome, ...;
- **átomos** - são constantes, devem ser iniciadas com minúsculas seguidas de qualquer caractere alfanumérico ou qualquer sequência entre ' ' (aspas simples). Ex.: joao, 'João', '16', ...;
- **inteiros** - qualquer sequência numérica que não contenha ponto (.). Caracteres ASCII entre " " (aspas duplas) são tratados como listas de inteiros. Ex.: 1, 6, -3, "a", ...;
- **floats** - números com um ponto (.) e pelo menos uma casa decimal. Ex.: 5.3 (correto), 7. (incorreto);
- **listas** - sequência ordenada de elementos entre [] e separados por vírgulas. Ex.: [a, b, c], [a | b, c].

Sintaxe SWI-Prolog – Exercício 01

Classifique os termos abaixo como átomo, variável, número, lista ou inválido:

Termo	Classificação
VINCENT	
23	
variable23	
aulas de lógica	
'Joao'	
[1, [2, 3], 4]	
Footmassage	
65.	
23.0	
–	
'aulas de lógica'	
"a"	
[]	

Sintaxe SWI-Prolog – Respostas exercício 1

Classifique os termos abaixo como átomo, variável, número, lista ou inválido:

Termo	Classificação
vINCENT	átomo
23	inteiro
variable23	átomo
aulas de lógica	erro
'Joao'	átomo
[1, [2, 3], 4]	lista
Footmassage	variável
65.	erro
23.0	float
_	variável anônima
'aulas de lógica'	átomo
"a"	lista
[]	lista

Sintaxe SWI-Prolog

Os comandos `write` e `read`, escrevem e leem sobre os arquivos padrão (monitor e teclado).

- Para escrever basta utilizar o comando `write(+termo)`:
 - `write('Teste de impressão.')`. (Correto)
 - `write(Teste de impressão.)`. (Errado)
 - `write(X)`. (Correto)
 - `write(joao)`. (Correto)
- Para ler deve-se usar o comando `read(+var)`:
 - `read(X)`. (Correto)
 - `read(x)`. (Errado)
 - `read(Joao)`. (Correto)
 - `read(joao)`. (Errado)

Sintaxe SWI-Prolog

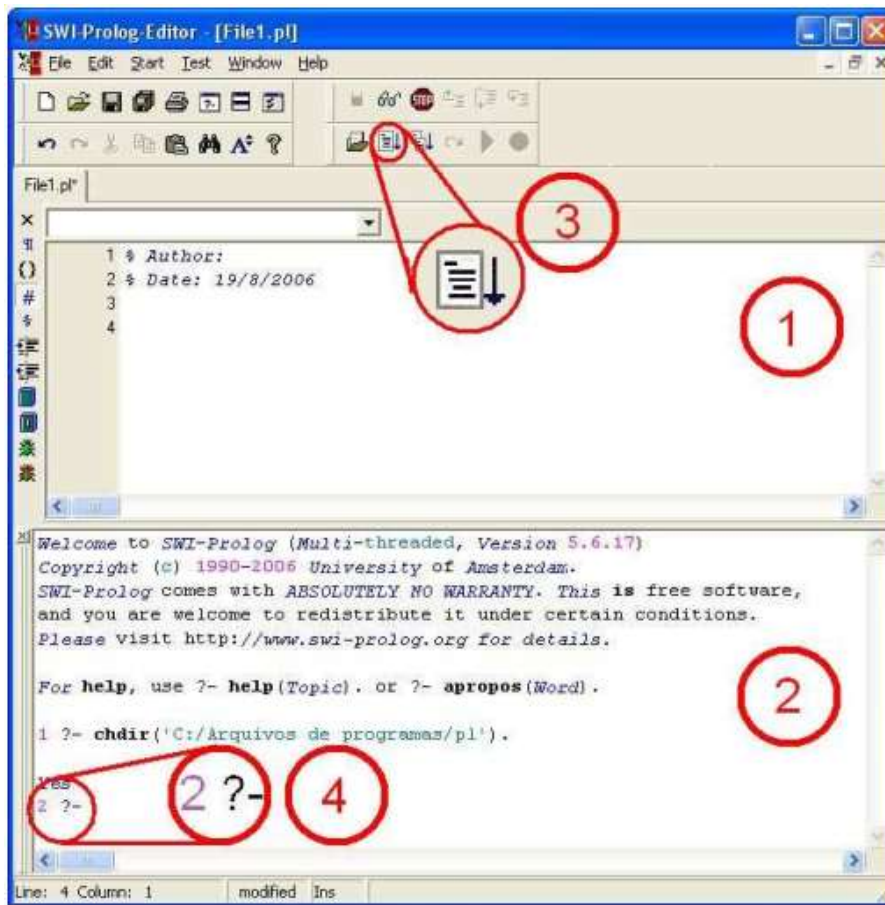
Alguns caracteres são especiais para impressão, são eles:

- `nl`, `\n`, `\l` - nova linha.
- `\r` - retorna ao início da linha;
- `\t` - tabulação;
- `\%` - imprime o símbolo %;

Existem dois tipos de comentários em Prolog, são eles:

- `%` - todo texto existente na mesma linha após o símbolo é considerado comentário;
- `/* */` - todo o texto entre os símbolos é considerado comentário.

SWI-Prolog



Para ter o primeiro contato com a ferramenta vamos checar as respostas do Exercício 1 utilizam os comandos abaixo no prompt (2):

atom(+termo) - checa se termo é um átomo;
var(+termo) - checa se termo é uma variável;
number(+termo) - checa se termo é um número (inteiro ou float);
is_list(+termo) - checa se termo é uma lista;

Em seguida vamos digitar o comando abaixo na base de conhecimento (1), carregar e executar o código (3), por fim, fazer a consulta em (4).

```
start( ) :-  
    write('Digite o valor de X: '), nl,  
    read(X), nl,  
    write(X), nl.
```

Fatos

Um programa Prolog é uma coleção de fatos e regras.

Fatos são sempre verdadeiros, mas as regras precisam ser avaliadas.

Como criar um fato em uma base Prolog:

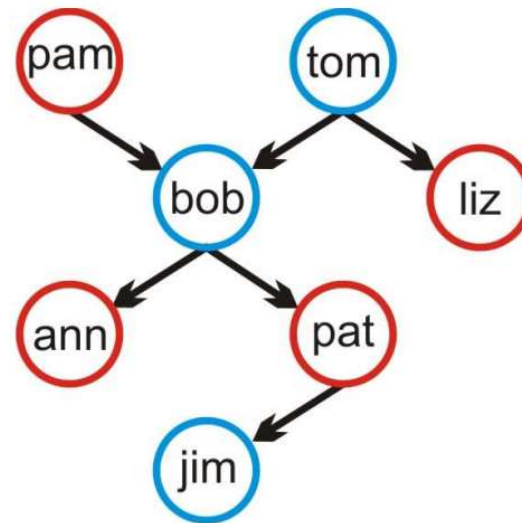
- `homem(x).` - significa que "x é um homem";
- `genitor(x, y).` - significa que "x é genitor de y" ou "y é gerado de x";

É responsabilidade do programador definir os predicados corretamente.

um **predicado** é uma [declaração](#) que deve ser verdadeira ou falsa dependendo do valor de suas variáveis.

Fatos – Exercício 2

Defina os fatos que descrevem a imagem abaixo:



Árvore genealógica

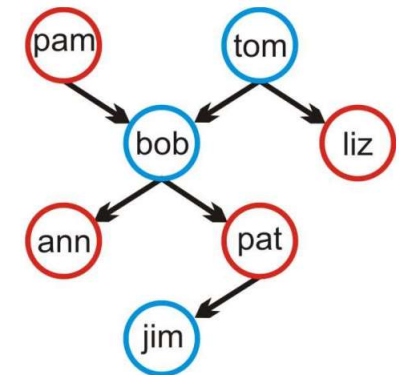
Fatos – Resposta do exercício 2

```
homem(jim). % jim é homem
homem(tom). % tom é homem
homem(bob). % bob é homem
mulher(liz). % liz é mulher
mulher(pat). % pat é mulher
mulher(ann). % ann é mulher
mulher(pam). % pam é mulher
genitor(pam, bob). % pam é genitora de bob
genitor(tom, bob). % tom é genitor de bob
genitor(bob, ann). % bob é genitor de ann
genitor(bob, pat). % bob é genitor de pat
genitor(pat, jim). % pat é genitora de jim
genitor(tom, liz). % tom é genitora de liz
```

Atenção: A ordem das regras é importante, deve-se também agrupar os predicados.

O nome do fato recebe o nome de **funtor**. Esse nome deve seguir a mesma regra de nomeação de átomo.

O termo **aridade** é usado para a quantidade de objetos que o argumento de um predicado possui. Por exemplo, genitor tem aridade 2.



Fatos – Exercício 3

Descreva o que os fatos abaixo significam

Fato	Descrição
animal(urso).	
animal(peixe).	
planta(alga).	
planta(grama).	
come(urso, peixe).	
come(coelho, leão)	
menor(formiga, tamanduá).	
menor(elefante, cavalo).	
proximo('Curitiba', 'Natal').	
proximo(Brasil, Japao).	

Fatos – Resposta do exercício 3

Descreva o que os fatos abaixo significam

Fato	Descrição
animal(urso).	urso é um animal
animal(peixe).	peixe é um animal
planta(alga).	alga é um planta
planta(grama).	grama é um planta
come(urso, peixe).	urso come peixe
come(coelho, leão)	coelho come leão???
menor(formiga, tamanduá).	formiga é menor que o tamanduá???
menor(elefante, cavalo).	elefante é menor que o cavalo???
proximo('Curitiba', 'Natal').	Curitiba é próximo de Natal
proximo(Brasil, Japao).	Não diz que Brasil está perto do Japão!

Mesmo que não faça sentido para o ser humano, o que está descrito nas regras da base de conhecimento é o que é real para o programa em questão.



Consultas

A cláusula `proximo(Brasil, Japao)`. é uma consulta Prolog, pois, “Brasil” e “Japao” são variáveis.

Para responder consultas Prolog utiliza:

- **matching** - checa se determinado padrão está presente, para saber quais fatos e regras podem ser utilizados;
- **unificação** - substitui o valor de variáveis para determinar se a consulta é satisfeita pelos fatos ou regras da base (programa);
- **resolução** - verifica se uma consulta é consequência lógica dos fatos e regras da base (programa);
- **recursão** - utiliza regras que chamam a si mesmas para realizar demonstrações;
- **backtracking** - para checar todas as possibilidades de resposta.



Consultas

Exemplo das etapas na consulta...

```
homem(tom). % fato
mulher(pam). % fato
genitor(pam, bob). % fato
genitor(tom, bob). % fato
```

```
?- genitor(tom, X). % tom é genitor de quem (X)?
X = bob ;      <- Unifica [X/bob]
No             <- genitor(tom, bob). (matching)
               <- busca outras soluções (backtrack)
               <- Nenhum outro fato satisfaz a consulta
```

```
?- genitor(X, bob). %Quem (X) é/são o(s) genitor(es) de bob?
X = tom ;      <- Unifica [X/tom]
X = pam ;      <- genitor(tom, bob). (matching)
No             <- busca outras soluções (backtrack).
               <- Unifica [X/pam]
               <- genitor(pam, bob). (matching)
               <- busca outras soluções (backtrack).
               <- Nenhum outro fato satisfaz a consulta.
```


Consultas – Exercício 4

Descreva o que querem dizer as seguintes consultas:

Consulta	Descrição
?- animal(X).	
?- animal(peixe).	
?- planta(alga).	
?- planta(x).	
?- come(urso, peixe).	
?- come(X, peixe).	
?- come(X, Y).	
?- come(raposa, _).	
?- come(_ , coelho).	

Consultas – Resposta do exercício 4

Descreva o que querem dizer as seguintes consultas:

Consulta	Descrição
?- animal(X).	Quem (X) são animais?
?- animal(peixe).	Peixe é um animal?
?- planta(alga).	Alga é uma planta?
?- planta(x).	Quem(X) é uma planta?
?- come(urso, peixe).	urso come peixe?
?- come(X, peixe).	Quem (X) come peixe?
?- come(X, Y).	Quais animais (X) comem quais?
?- come(raposa, _).	A raposa come algum animal?
?- come(_ , coelho).	Algum animal come coelho?

Consultas – Exercício 5

Dado a base de conhecimento abaixo, quais as respostas para as consultas?

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

```
?- planta(X).  
?- come(raposa, _).  
?- come(_, _).  
?- come(X, grama).
```

?

Consultas – Resposta do exercício 5

Dado a base de conhecimento abaixo, quais as respostas para as consultas?

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

?- planta(X). X = alga; X = grama; No
?- come(raposa,_). Yes
?- come(_, _). Yes
?- come(X, grama). X = coelho ; X = veado; No



Regras

Regras facilitam a execução de consultas e tornam um programa muito mais expressivo.

Uma cláusula Prolog é equivalente à uma fórmula em lógica de 1ª ordem, então, em Prolog, existem os conectivos:

- :- (se), equivalente à implicação;
- , (e), equivalente à conjunção;
- ; (ou), equivalente à disjunção.

Exemplo:

A fórmula: $A(x) \rightarrow B(x) \vee (C(x) \wedge D(x))$, seria escrita em Prolog como: $a(X) :- b(X); (c(x), d(x))$.

Prolog não utiliza quantificadores explicitamente, porém, trata todas as regras como se elas estivessem universalmente quantificadas e usa \sim EU (eliminação do universal).



Regras

Consultas são realizadas sobre regras do mesmo modo como ocorrem sobre fatos. Uma regra se divide em conclusão (ou cabeça) e condição, da seguinte forma:

CONCLUSAO(+ARG) :- CONDIÇÃO1(+ARG) CONECTIVO CONDIÇÃO2(+ARG) ...

Utilizando *matching*, Prolog encontra quais regras podem ser utilizadas para satisfazer uma consulta. Cada vez que um *matching* ocorre a satisfação da regra passa a ser a meta atual.

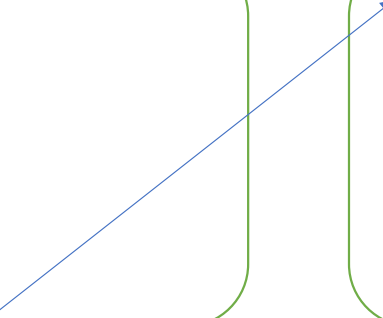
Regra: prole(X,Y) :- genitor(Y,X)

Consulta: ?- prole(pam, bob)

Regras - Exemplo

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

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



Como podemos especificar uma regra `prole(x, y).`, significando que “x é prole (filho ou filha) de y”.

Prole (filho ou filha) é a relação inversa de Genitor (pai ou mãe), então: “x é prole de y, se y é genitor de x”.

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

Regras – Exercício 6

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

```
genitor(pam, bob).  
genitor(tom, bob).  
genitor(bob, ann).  
genitor(bob, pat).  
genitor(pat, jim).  
genitor(tom, liz).  
prole(X,Y) :- genitor(Y,X).
```

a) Qual a resposta para as consultas?

?- prole(_, tom).

?- prole(tom, X).

b) Descreva regras para as seguintes relações:

mae(x,y), significando “x é mãe de y”;

avos(X,Z), significando “x é avô/avó de y”.

Regras – Resposta do Exercício 6

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

```
genitor(pam, bob).  
genitor(tom, bob).  
genitor(bob, ann).  
genitor(bob, pat).  
genitor(pat, jim).  
genitor(tom, liz).  
prole(X,Y) :- genitor(Y,X).
```

a) Qual a resposta para as consultas?

?- prole(_, tom). **Yes**

?- prole(tom, X). **X = bob ; X = liz ; No**

b) Descreva regras para as seguintes relações:

mae(X,Y) :- **genitor(X,Y), mulher(X).**

avos(X,Z) :- **genitor(X,Y), genitor(Y,Z).**

Regras

Exemplo das etapas na consulta de regras...

```

homem(tom). % fato
mulher(pam). % fato
genitor(pam, bob). % fato
genitor(tom, bob). % fato
prole(X,Y) :- genitor(Y,X).
?- prole(pam, bob)

```

Regras

Outro exemplo das etapas na consulta...

1. chama(a,b). % fato
 2. usa(b,e). % fato
 3. depende(X, Y) :- chama(X,Y). % regra
 4. depende(X,Y) :- usa(X, Y). % regra
 5. depende(X, Y) :- chama(X, Z), depende(Z, Y). % regra recursiva
- ?- depende(a,e). % 'a' depende de 'e'?

<- 'a' depende de 'e'?
<- Unifica [X/a] e [Y/e]
<- Nova meta chama(a,e)
 <- Falha, não existe o fato
<- busca outras soluções (backtrack)
<- usa regra 4
<- Unifica [X/a] e [Y/e]
<- Nova meta usa(a,e)
 <- Falha, não existe o fato
<- busca outras soluções (backtrack)
<- usa regra 5
<- Unifica [X/a] e [Y/e]
<- Nova meta chama(a,Z)
 <- Unifica [Z/b]
 <- chama(a,b) (matching)
 <- Nova meta depende(b, e) (recursão)
 <- usa regra 3
 <- Unifica [X/b] e [Y/e]
 <- Nova meta chama(b,e)
 <- Falha, não existe o fato
 <- busca outras soluções (backtrack)
 <- usa regra 4
 <- Unifica [X/b] e [Y/e]
 <- Nova meta usa(b,e)
 <- usa(b,e) (matching)
 <- OK

REGRAS

Regras – Exercício 7

Descreva uma regra para determinar quais animais são presas:

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

Regras – Resposta do Exercício 7

Descreva uma regra para determinar quais animais são presas:

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

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

Fatos, Regras – Lista de exercícios 01

1) Considere a seguinte base de fatos em Prolog:

aluno(joao, calculo).	professor(carlos, calculo).
aluno(maria, calculo).	professor(ana_paula, estrutura).
aluno(joel, programacao).	professor(pedro, programacao).
aluno(joel, estrutura).	funcionario(pedro, ufrj).
frequenta(joao, puc).	funcionario(ana_paula, puc).
frequenta(maria, puc).	funcionario(carlos, puc).
frequenta(joel, ufrj).	

Escreva as seguintes regras em Prolog:

- a) Quem são os alunos do professor X?
- b) Quem são as pessoas que estão associadas a uma universidade X? (alunos e professores)

Fatos, Regras – Lista de exercícios 01 (cont...)

2) Elabore um programa Prolog que forneça o nome da capital de qualquer estado da região sudeste.

3) 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. Depois faça consultas para saber se um tipo pode doar para outro tipo, e se um tipo pode receber de outro tipo:

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



Manipulação da base de conhecimento

A princípio, os predicados carregados pela instrução *consult* na base de conhecimento, são **estáticos**.

Porém, existem predicados pré-definidos que permitem fazer a manipulação da base de conhecimento. Ou seja, predicados que permitem acrescentar e/ou retirar factos e regras da base de conhecimento, durante a execução de um programa.

Para criar predicados dinâmicos é necessário utilizar a diretiva:

`:- dynamic NomeDoPredicado/Aridade.`

Contudo as alterações serão voláteis, ou seja, o arquivo original não é alterado.



Manipulação da base de conhecimento

Considerando a base de conhecimento da árvore genealógica apresentada anteriormente, agora devemos informar que os predicados são dinâmicos:

```
:- dynamic homem/1.  
:- dynamic mulher/1.  
:- dynamic genitor/2.
```

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

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



Manipulação da base de conhecimento

Primeiramente, pode-se consultar os predicados existentes com o seguinte comando:

```
?- listing(homem).  
homem(jim).  
homem(tom).  
homem(bob).
```

```
?- listing(mulher/1).  
mulher(liz).  
mulher(pat).  
mulher(ann).  
mulher(pam).
```

Para adicionar fatos e regras à base de conhecimento pode-se usar os predicados:

- **assert/1** – acrescenta o fato/regra como **último** item do predicado;
- **asserta/1** – acrescenta o fato/regra como **primeiro** item do predicado;
- **assertz/1** – idêntico a assert/1.

Manipulação da base de conhecimento



Exemplos (adicionando informações):

```
?- assert(homem(freddie)).
```

```
?- listing(homem).  
:- dynamic homem/1.  
homem(jim).  
homem(tom).  
homem(bob).  
homem(freddie).
```

```
?- asserta(mulher(janes)).
```

```
?- listing(mulher).  
:- dynamic mulher/1.  
mulher(janes).  
mulher(liz).  
mulher(pat).  
mulher(ann).  
mulher(pam).
```

```
?- assertz(homem(steven)).
```

```
20 ?- listing(homem).  
:- dynamic homem/1.  
  
homem(jim).  
homem(tom).  
homem(bob).  
homem(freddie).  
homem(steven).
```



Manipulação da base de conhecimento

Para remover fatos e regras da base de conhecimento pode-se usar os predicados:

- **retract/1** remove da base de conhecimento a primeira cláusula (fato ou regra) que unifica com o termo que é passado como parâmetro;
- **retractall/1** remove da base de conhecimento todos os fatos ou regras cuja cláusula (fato ou regra) unifique com o termo que é passado como parâmetro;
- **abolish/1** remove da base de conhecimento todos os fatos e regras pelo nome da regra ou fato/aridade que é passada como parâmetro (são removidos predicados estáticos também);
- **abolish/2** semelhante a abolish/1, mas passando o nome da fato/regra e a sua aridade separadamente (são removidos predicados estáticos também).



Manipulação da base de conhecimento

Exemplos (removendo informações):

```
?- retract(homem(jim)).
```

```
23 ?- listing(homem).  
:- dynamic homem/1.
```

```
homem(tom).  
homem(bob).  
homem(freddie).  
homem(steven).
```

```
?- listing(genitor(_,bob)).  
:- dynamic genitor/2.
```

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

```
?- retractall(genitor(_, bob)).
```

```
?- listing(genitor(_,bob)).  
:- dynamic genitor/2.
```

```
abolish(genitor/2).
```

```
?- listing(genitor).  
:- dynamic genitor/0.
```

ou

```
?- abolish(genitor, 2).
```

```
54 ?- listing(genitor).  
:- dynamic genitor/0.
```



Manipulação de base – Exercício 7 B

A informação referente aos horários das salas de aula pode estar guardada na base de conhecimento em fatos da forma **sala(num,dia,inicio,fim,discipl,tipo)** :

:- dynamic sala/6.

```
sala(cp1103, seg, 10, 13, aaa, p).  
sala(cp2301, ter, 10, 11, aaa, t).  
sala(di011, sab, 12, 10, xxx, p). % com erro  
sala(cp3204, dom, 8, 10, zzz, p).  
sala(di011, sex, 14, 16, xxx, p).  
sala(cp204, sab, 15, 17, zzz, tp).  
sala(di011, qui, 14, 13, bbb, tp). % com erro  
sala(di104, qui, 9, 10, aaa, tp).  
sala(dia1, dom, 14, 16, bbb, t).  
sala(cp1220, sab, 14, 18, sss, p).
```

a) Crie uma consulta que retira da base de dados todas as salas em que, erradamente, a hora de início da aula é superior à hora de fim.

b) Insira de novo as aulas removidas, mas agora com os horários corretos, ou seja, início menor que o horário de fim.



Manipulação de base – Exercício 7 B

A informação referente aos horários das salas de aula pode estar guardada na base de conhecimento em factos da forma **sala(num,dia,inicio,fim,discipl,tipo)** :

```
:- dynamic sala/6.
```

```
sala(cp1103, seg, 10, 13, aaa, p).  
sala(cp2301, ter, 10, 11, aaa, t).  
sala(di011, sab, 12, 10, xxx, p). % com erro  
sala(cp3204, dom, 8, 10, zzz, p).  
sala(di011, sex, 14, 16, xxx, p).  
sala(cp204, sab, 15, 17, zzz, tp).  
sala(di011, qui, 14, 13, bbb, tp). % com erro  
sala(di104, qui, 9, 10, aaa, tp).  
sala(dia1, dom, 14, 16, bbb, t).  
sala(cp1220, sab, 14, 18, sss, p).
```

a) Crie uma consulta que retira da base de dados todas as sala em que, erradamente, a hora de início da aula é superior à hora de fim.

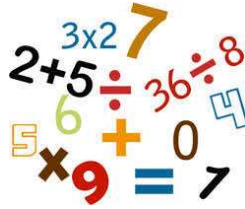
```
?- (sala(_, _, Ini, Fim, _, _), Fim < Ini),  
    retract(sala(_, _, Ini, Fim, _, _)).
```

b) Insira de novo as aulas removidas, mas agora com os horários corretos, ou seja, início menor que o horário de fim.

```
?- assert(sala(di011, sab, 10, 12, xxx, p)).
```

```
?- assert(sala(di011, qui, 14, 13, bbb, tp)).
```

Aritmética

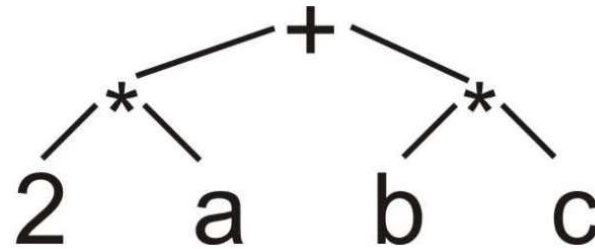


Prolog é mais indicada para resolução de problemas simbólicos, mas também oferece suporte à aritmética.

Podemos utilizar duas notações para representar expressões em Prolog:

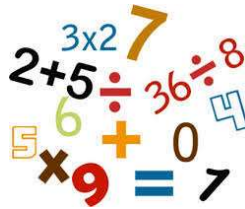
- Infixa: $2 * a + b * c$
- Prefixa: $+(*(2, a), *(b, c))$

Prolog trata as representações de forma equivalente, pois, internamente utiliza árvores para representar expressões. Assim, basta mudar a ordem de caminhamento para obter uma ou outra forma.



Árvore para a expressão $2 * a + b * c$

Aritmética



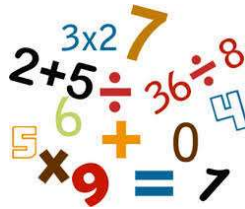
São oferecidos diversos operadores para cálculos aritméticos, alguns destes são:

Operador	Significado
+, -, *, /	Realizar soma, subtração, multiplicação e divisão, respectivamente
is	atribui uma expressão numérica à uma variável
mod	Obter o resto da divisão
^	Calcular potenciação
cos, sin, tan	Função cosseno, seno e tangente, respectivamente
exp	exponenciação
ln, log	logaritmo natural e logaritmo
sqrt	raiz

Existem predicados de conversão, tais como:

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

Aritmética



Prolog também possui predicados para comparação, os operadores são:

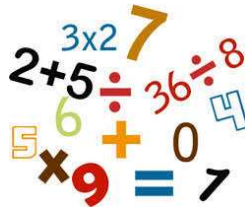
Operador	Significado
>	Maior que
<	Menor que
>=	Maior ou igual a
=<	Menor ou igual a
==	Igual
\=	diferente
\+	Negação – retorna sucesso se o predicado for falso e vice-versa.

Os operadores = e ==, realizam diferentes tipos de comparação:

- =, checa se os “objetos” são iguais, ou atribui valores para as variáveis;
- ==, avalia se os valores são iguais.

Unificação
de termo.

Aritmética



Exemplo de uso de = e :=

?- 1 + 2 = 2 + 1. **No**
?- 1 + 2 := 2 + 1. **Yes**
?- 1 + A = B + 2. A = 2 B = 1 ; **No**
?- A := 1. **ERROR**
?- a \=a. **false**
?- 1 + 2 := +(1, 2). **true**
?- 1 + 2 := +(1, 4). **False**
?- X is sqrt(9). **X = 3**

Observação sobre o operador is:

doa(a,a).
doa(a,ab).

?- doa(a,a).
true.

?- \+doa(a,a).
false.

Negação.

somar(A,B,Soma) :- Soma is A + B.

?- somar(1,2,X).

X = 3;

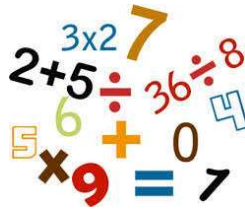
?- somar(1,2,3).

true ;

?- somar(1,X,3).

ERROR:

Todas as variáveis do lado direito do operador **is**, devem estar instanciadas, ou seja, devem ter valores associados a elas.



Aritmética – Lista de exercício 02

- 1) Crie uma regra em Prolog que peça no console um numero inteiro e imprima na tela se o número é maior que 100 ou se é menor ou igual a 100.
- 2) Suponha os seguintes fatos:

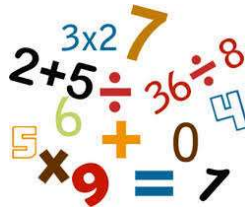
nota(joao,5.0). nota(maria,6.0). nota(joana,8.0).
nota(mariana,9.0). nota(cleuza,8.5). nota(jose,6.5).
nota(jaoquim,4.5). nota(mara,4.0). nota(mary,10.0).

Considerando que: Nota de 7.0 á 10.0 = Aprovado.

Nota de 5.0 á 6.9 = Recuperação.

Nota de 0.0 á 4.9 = Reprovado.

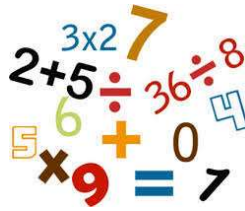
Escreva uma regra para identificar a situação de um determinado aluno.



Aritmética – Lista de exercício 02

3) Crie uma base de conhecimento em Prolog declarando os fatos representados na seguinte tabela:

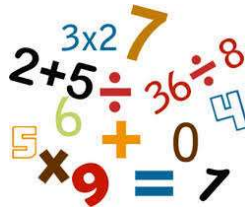
CATÁLOGO DE FILMES				
Título	Gênero	Diretor	Ano	Min.
Amnésia	Suspense	Nolan	2000	113
Babel	Drama	Inarritu	2006	142
Capote	Drama	Miller	2005	98
Casablanca	Romance	Curtiz	1942	102
Matrix	Ficção	Wachowsk	1999	136
Rebecca	Suspense	Hitchcock	1940	130
Shrek	Aventura	Adamson	2001	90
Sinais	Ficção	Shymalan	2002	106
Spartacus	Ação	Kubrik	1960	184
Superman	Aventura	Donner	1978	143
Titanic	Romance	Cameron	1997	194
Tubarão	Suspense	Spielberg	1975	124
Volver	Drama	Almodóvar	2006	121



Aritmética – Lista de exercício 02

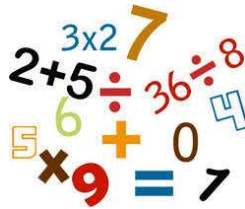
4) Escreva regras genéricas em Prolog que possam responder as seguintes perguntas:

- a) Quem dirigiu o filme Babel?
- b) Quais são os filmes de suspense?
- c) Quais os filmes dirigidos por Nolan?
- d) Em que ano foi lançado o filme Casa Blanca?
- e) Quais os filmes com duração inferior a 100min?
- f) Quais os filmes lançados entre 2000 e 2005?



Aritmética – Lista de exercício 02

- 5) Usando as regras criadas anteriormente construa o predicado “clássico”, que retorna o título dos filmes lançados antes de 1980.
- 6) Usando as regras criadas anteriormente construa o predicado “gênero”, que retorna o título dos filmes de um gênero específico.
- 7) Usando os predicados “clássico” e “gênero” faça uma consulta para recuperar os títulos de filmes clássicos de suspense.
- 8) Quais são os resultados das consultas abaixo:
- a) $\text{ponto}(A, B) = \text{ponto}(1, 2)$
 - b) $2 + 2 = 4$
 - c) $\text{ponto}(A, B) = \text{ponto}(X, Y, Z)$
 - d) $\text{mais}(2, 2) = 4$
 - e) $+(2, D) = +(E, 2)$
 - f) $t(p(-1,0), P2, P3) = t(P1, p(1, 0), p(0, Y))$



Aritmética – Lista de exercício 02

9) Crie uma regra para calcular o índice de massa corporal (IMC), a regra recebe o peso e a altura. O retorno com o resultado será escrito no console.

10) Modifique o programa acima para classificar o IMC (use corte):

imc < 20, 'IMC abaixo do normal.'

imc >= 20 e < 30, 'IMC está normal.'

imc >= 30 e < 40, 'Obesidade grau 1.'

imc >= 30 e < 40, 'Obesidade grau 2.'

imc > 40, 'Obesidade grau 3.'



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.

Um regra é recursiva se sua condição depende dela mesma, tal como:

$$a(X) : -b(X), a(X).$$

A importância do uso de recursão pode ser ilustrada na implementação da relação descendente(x, y), significando que “ x é descendente de y ”.

Um conjunto de regras com o mesmo nome é denominado **procedimento**.



Regras recursivas

Exemplo sem recursão (descendência genealógica):

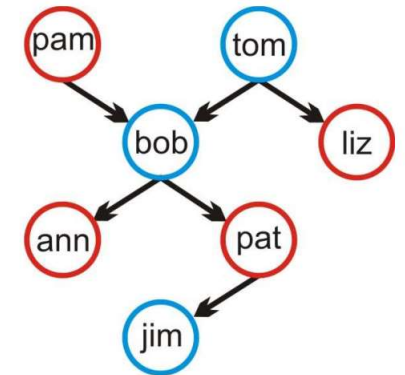
Para pais, avos, bisavos está correto, mais e para tataravôs? Essa solução é limitada e trabalhosa.

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

Exemplo com recursão (descendência genealógica):

Solução simples e trata todos os níveis da hierarquia, mas lembre-se, **nunca esqueça do caso base antes da chamada recursiva.**

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





Regras recursivas

Exemplo com recursão (cálculo do fatorial):

```
fatorial(5) = fatorial(4) * 5
            = (fatorial(3) * 4) * 5
            = ((fatorial(2) * 3) * 4) * 5
            = (((fatorial(1) * 2) * 3) * 4) * 5
            = ((((fatorial(0) * 1) * 2) * 3) * 4) * 5
            = (((((1 * 1) * 2) * 3) * 4) * 5
```

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

?- fatorial(3,R).
R = 6 ;
```

Regras recursivas – Exercício 8

1) Descreva uma regra para determinar quais animais pertencem a cadeia alimentar de outro:

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

Regras recursivas – Resposta do Exercício 8

1) Descreva uma regra para determinar quais animais pertencem a cadeia alimentar de outro:

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

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

```
?- pertence-cadeia(X,raposa).
?- pertence-cadeia(X,urso).
```

Regras recursivas – Exercício 8

2) Crie um programa em Prolog para resolver a equação de recorrência:

Regra

Caso base: $x_1 = 2$.

$$x_n = x_{n-1} - 3n^2$$

Predicado

regraRecorrencia/2

Regras recursivas – Resposta do Exercício 8

2) Crie um programa em Prolog para resolver a equação de recorrência:

Regra

Caso base: $x_1 = 2$.

$$x_n = x_{n-1} - 3n^2$$

Predicado

regraRecorrencia/2

```
rec1(N,2) :- N = 1.  
rec1(N,Result) :- N >= 2,  
                  N1 is N-1,  
                  rec1(N1,Result1),  
                  Result is Result1 - 3*(N*N).
```

```
?- rec1(2,Y).  
Y = -10 ;  
false.
```

```
28 ?- rec1(3,Y).  
Y = -37 ;  
false.
```

Corte



O retrocesso (backtracking) é um processo pelo qual todas as alternativas de solução para uma dada consulta são tentadas exaustivamente.

No Prolog, o retrocesso é automático.

Contudo, é possível controlá-lo através de um predicado especial chamado **corte**, denotado por **!**.

Visto como uma cláusula, seu valor é sempre verdadeiro. Sua função é provocar um efeito colateral que interfere no processamento padrão de uma consulta.

Corte



Pode ser usado em qualquer posição no lado direito de uma regra.

O corte é adequado às situações onde regras diferentes são aplicadas em casos mutuamente exclusivos. Faz com que o programa se torne mais rápido e ocupe menos memória.

Quando colocado no final de uma cláusula que define um predicado, evita que as cláusulas abaixo dessa, relativas ao mesmo predicado, sejam usadas no backtracking.



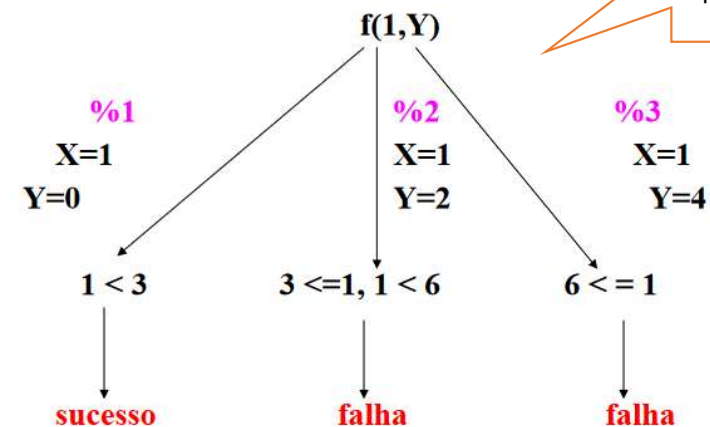
Corte (Exemplo)

Construir um programa Prolog para implementar a função:

$$f(x) = \begin{cases} 0 & \text{se } x < 3 \\ 2 & \text{se } x \geq 3 \text{ e } x < 6 \\ 4 & \text{se } x \geq 6 \end{cases}$$

```
f(X,0) :- X < 3.           %1  
f(X,2) :- 3 <= X, X < 6. %2  
f(X,4) :- 6 <= X.         %3
```

```
?- f(1,Y).  
Y = 0 ;  
false.
```



Sabemos que as regras são mutuamente exclusivas. Logo, para quer testas as outras regras?



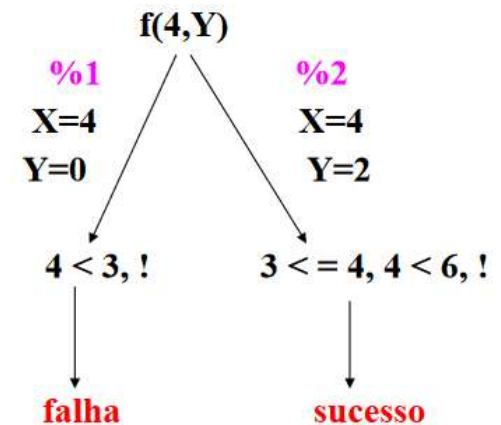
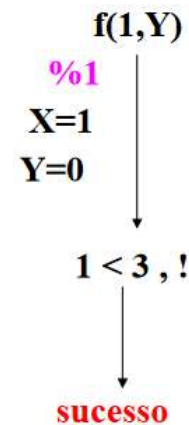
Corte (Exemplo)

Construir um programa Prolog para implementar a função:

$$f(x) = \begin{cases} 0 & \text{se } x < 3 \\ 2 & \text{se } x \geq 3 \text{ e } x < 6 \\ 4 & \text{se } x \geq 6 \end{cases}$$

```
f(X,0) :- X < 3, !.           %1
f(X,2) :- 3 =< X, X < 6, !.  %2
f(X,4) :- 6 =< X.             %3
```

```
?- f(1,Y).
Y = 0 ;
false.
```



Esse tipo de corte é chamado de **corte verde**, se for retirado, o programa executa a mesma lógica. Altera apenas a eficiência.



Corte (Exemplo)

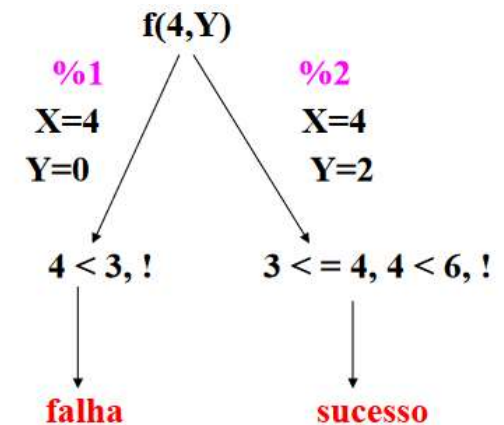
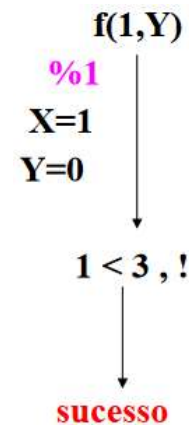
Construir um programa Prolog para implementar a função:

$$f(x) = \begin{cases} 0 & \text{se } x < 3 \\ 2 & \text{se } x \geq 3 \text{ e } x < 6 \\ 4 & \text{se } x > 6 \end{cases}$$

O corte pode deixar os programas mais compactos, mas agora temos um exemplo de **corte vermelho**, se for retirado, o programa não executará da mesma forma.

```
f(X,0) :- X < 3, ! .      %1
f(X,2) :- X < 6, ! .      %2
f(X,4) .                  %3
```

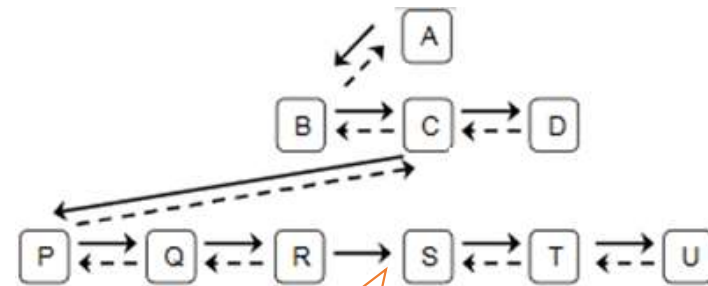
```
?- f(1,Y).
Y = 0 ;
false.
```



Corte (Exemplo)



```
b.  
d.  
p.  
q.  
r.  
s.  
t.  
u.  
v.  
c :- p, q, r, !, s, t, u.  
c :- v.  
a :- b, c, d.  
  
?- a.
```



Quando o corte é encontrado no meio de uma regra, o comportamento é um pouco diferente do apresentado no slide anterior. O Backtracking irá acontecer somente a direita do corte. Ou seja, as unificações feitas pelo Prolog à esquerda de um corte não podem ser desfeitas após a passagem da busca.

A pair of scissors with blue handles and a silver blade, positioned diagonally.

A hierarchical tree diagram showing the relationships between 11 taxa (A, B, C, D, P, Q, R, S, T, U). The root is A, which branches into B and C. C branches into D and P. P branches into Q and R. R branches into S and T. T branches into U. The diagram uses solid lines for primary branches and dashed lines for secondary branches.

Também será descartada a cláusula alternativa para a meta C :- V

A meta-pai da cláusula contendo o corte é a meta C na cláusula na cláusula A :- B, C, D.

Portanto, o corte afetará apenas a execução da meta C; mas será 'invisível' de dentro da meta A: backtracking automático continuará a existir entre a lista de metas B, C, D apesar do corte dentro da cláusula usada para satisfazer C



Corte (Exemplo)

```
a(1).  
b(2).  
c(1).  
p(X) :- c(X).  
p(X) :- a(X) , ! , b(X).
```

```
?- p(1).  
  Call: (7) p(1) ? creep  
  Unify: (7) p(1)  
  Call: (8) c(1) ? creep  
  Unify: (8) c(1)  
  Exit: (8) c(1) ? creep  
  Exit: (7) p(1) ? Creep  
true
```

```
a(1).  
b(2).  
c(1).  
p(X) :- c(X).  
p(X) :- a(X) , ! , b(X).
```

```
?- p(2).  
  Call: (7) p(2) ? creep  
  Unify: (7) p(2)  
  Call: (8) c(2) ? creep  
  Fail: (8) c(2) ? creep  
  Redo: (7) p(2) ? creep  
  Unify: (7) p(2)  
  Call: (8) a(2) ? creep  
  Fail: (8) a(2) ? creep  
  Fail: (7) p(2) ? creep  
false.
```



Corte (Exemplo)

```
a(1).  
b(2).  
c(1).  
p(X) :- a(X) , ! , b(X).  
p(X) :- c(X).
```

```
?- p(1).  
  Call: (7) p(1) ? creep  
  Unify: (7) p(1)  
  Call: (8) a(1) ? creep  
  Unify: (8) a(1)  
  Exit: (8) a(1) ? creep  
  Call: (8) b(1) ? creep  
  Fail: (8) b(1) ? creep  
  Fail: (7) p(1) ? creep  
false
```

```
a(1).  
b(2).  
c(1).  
p(X) :- a(X) , ! , b(X).  
p(X) :- c(X).
```

```
?- p(2).  
  Call: (7) p(2) ? creep  
  Unify: (7) p(2)  
  Call: (8) a(2) ? creep  
  Fail: (8) a(2) ? creep  
  Redo: (7) p(2) ? creep  
  Unify: (7) p(2)  
  Call: (8) c(2) ? creep  
  Fail: (8) c(2) ? creep  
  Fail: (7) p(2) ? creep  
false.
```




Corte – Exercício 9

1) Considere o programa:

`m(1) .`

`m(2) :- !.`

`m(3) .`

`m1(X, Y) :- m(X), m(Y) .`

`m2(X, Y) :- m(X), !, m(Y) .`

Diga quais são todas as respostas do Prolog aos seguintes objectivos:

1. `?-m(X) .`

2. `?-m1(X, Y) .`

3. `?-m2(X, Y) .`

4. `?-m(3) .`

Depois que tentou responder as consultas, use o SWI-Prolog com o comando **trace** ativado e acompanhem passo a passo a solução da consulta.



Corte – Respostas do Exercício 9

1) .

```
m(1).  
m(2):-!.  
m(3).  
m1(X,Y) :- m(X),m(Y).  
m2(X,Y) :- m(X),!,m(Y).
```

```
?- m(1).  
    Call: (7) m(1) ? creep  
    Unify: (7) m(1)  
    Exit: (7) m(1) ? Creep
```

true

```
?- m1(X,Y)  
    Call: (7) ms(_G9768, _G9769) ? creep  
        Call: (8) m(_G9768) ? creep  
        Exit: (8) m(1) ? creep  
        Call: (8) m(_G9769) ? creep  
        Exit: (8) m(1) ? creep  
    Exit: (7) ms(1, 1) ? creep  
X = Y, Y = 1 ;  
    Redo: (8) m(_G9769) ? creep  
    Exit: (8) m(2) ? creep  
    Exit: (7) ms(1, 2) ? creep  
X = 1,  
Y = 2 ;  
    Redo: (8) m(_G9768) ? creep  
    Exit: (8) m(2) ? creep  
    Call: (8) m(_G9769) ? creep  
    Exit: (8) m(1) ? creep  
    Exit: (7) ms(2, 1) ? creep  
X = 2,  
Y = 1 ;  
    Redo: (8) m(_G9769) ? creep  
    Exit: (8) m(2) ? creep  
    Exit: (7) ms(2, 2) ? creep  
X = Y, Y = 2 ; false.
```

```
?- ms(X,Y).  
    Call: (7) ms(_G12488, _G12489) ? creep  
        Call: (8) m(_G12488) ? creep  
        Exit: (8) m(1) ? creep  
        Call: (8) m(_G12489) ? creep  
        Exit: (8) m(1) ? creep  
    Exit: (7) ms(1, 1) ? creep  
X = Y, Y = 1 ;  
    Redo: (8) m(_G12489) ? creep  
    Exit: (8) m(2) ? creep  
    Exit: (7) ms(1, 2) ? creep  
X = 1,  
Y = 2 ;  
    Redo: (8) m(_G12488) ? creep  
    Exit: (8) m(2) ?
```

Fail



O predicado **fail** (chamado de falha em português), sempre retorna uma falha na unificação. Com isso, força o Prolog a fazer o backtracking, repetindo os predicados anteriores a esse predicado.

1) Como não tem variável de entrada na regra, o prolog tenta satisfazer a regra somente uma vez.

2) Aqui também não tem variável de entrada na regra, mas a palavra fail força o backtracking da cabeça do predicado.

```
aluno(marcelo).  
aluno(andre).  
aluno(roberto).  
escreveSemFail :- aluno(X), write(X).  
escreveComFail :- aluno(X), write(X), nl, fail.
```

```
?- escreveSemFail.  
marcelo  
true.
```

```
?- escreveComFail.  
marcelo  
andre  
roberto  
false.
```

3) O que acontece se colocarmos o predicado "!, " antes de fail?

Fail



Imagina que queres escrever a palavra Olá repetidas vezes. O *fail* não funciona:

1) Por mais que forcamos o backtracking, os predicados nl e write não tem soluções alternativas, então o sistema para.

```
?- write(ola), nl, fail.  
ola  
false.
```

2) Se alterarmos a consulta incluindo um predicado chamado repeat, criamos um laço infinito.

```
?- repeat, write(ola), nl, fail.  
ola  
ola  
ola  
ola  
...
```



Repeat

O predicado **repeat** é usado para forçar uma repetição que termina só quando a cláusula como um todo é verdadeira ou quando o corte(!) não permite mais o backtracking. Para evitar o *loop* infinito pode-se criar regras com o seguinte formato:

repeat, lerDados, processarDados, condicao.

1) Criamos uma regra para iniciar um jogo para o usuário adivinhar o número gerado pelo programa

```
guess_number :-  
    N is random(5) + 1,  
    repeat,  
        ler_dados(G),  
        process_guess(G, N).
```

2) Leitura e processamento da entrada.

```
ler_dados(G):-  
    write('Input your guess (from 1 to 5): '),  
    read(G).
```

4) Força o backtracking, para que o usuário tente adivinhar novamente.

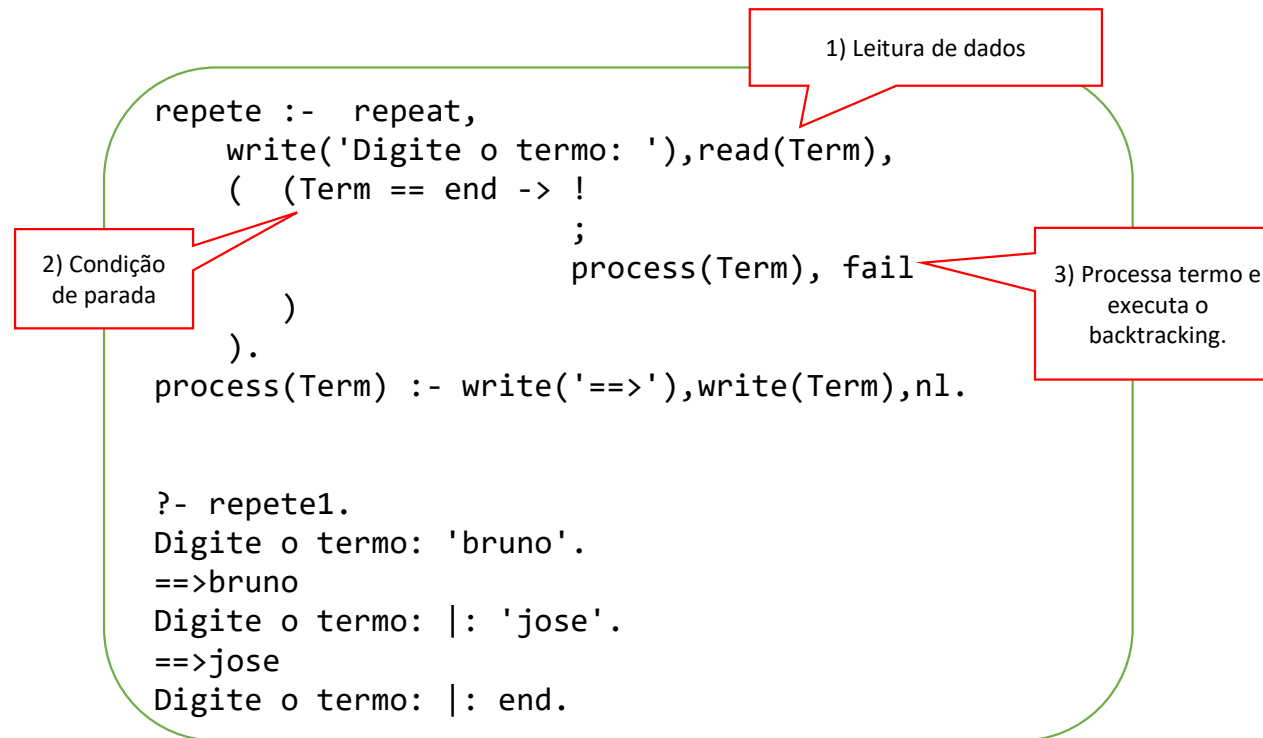
```
process_guess(N, N):-  
    write('You win!'), nl.  
process_guess(I, N):-  
    I \= N,  
    write('Não, não!'), nl,  
    fail.
```

3) Condição de parada, toda a cláusula é verdadeira.



Repeat

Outro exemplo com repeat:





Recursão, corte e Repeat – Exercício 10

- 1) Crie um programa em Prolog que leia um número e calcule e imprima o quadrado desse número, o programa deve continuar a execução até que o usuário digite a palavra 'stop'.
- 2) Crie uma regra recursiva com um parâmetro que é um número inteiro, a regra deve imprimir os valores de zero até o numero informado. Ex:

```
?- imprimeAte(3).
```

```
0 1 2 3
```

- 3) Usando um acumulador, e somente as operações(+)(-)(*), crie uma regra que calcule X elevado a Y. Assuma X e Y inteiros.



Recursão, corte e Repeat– Exercício 10

4) Crie uma regra para calcular o enésimo termo da série de Fibonacci.

$$\text{fib}(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ \text{fib}(n-2) + \text{fib}(n-1), & \text{if } n > 1 \end{cases}$$

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...

5) Escreva uma regra que informe se o número é primo ou não. Um número primo é aquele divisível somente por um e por ele mesmo. Uma das formas de se responder a essa pergunta seria, pegar esse numero e dividi-lo por ele mesmo e depois por todos os números antecessores a ele até chegar a um. Assuma que os valores informados serão maiores que zero.

6) Faça um menu para usuário executar as questões um ou dois, o menu deve-se encerrar quando o usuário digitar a palavra stop.



Listas

Uma lista é uma sequencia ordenada de elementos de qualquer tipo de dados de Prolog.

Os elementos contidos em uma lista devem ser separados por vírgulas, e precisam estar entre colchetes. Existem notações alternativas, porém, esta é a mais simples.

```
[pam, liz, pat, ann, tom, bob, jim]  
[1, 2, 3, 4, 5]  
[a, [b, c], d, e] % onde [b,c] é o segundo elemento da lista
```

Listas podem ser de dois tipos:

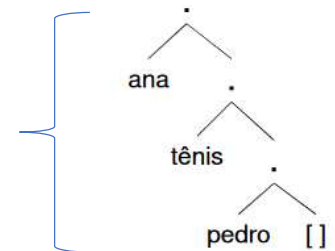
- ❑ vazias - quando não contém nenhum elemento, representadas por [];
- ❑ não-vazias - quando contém ao menos um elemento.

Listas



Uma lista vazia é representada por [];

Toda lista tem uma lista vazia dentro dela, o qual é o último elemento;



(ana, .(tênis, .(pedro, [])))

Listas não vazias possuem duas partes, são elas:

- ❑ cabeça (head) - corresponde ao primeiro elemento da lista;
- ❑ cauda (tail) - corresponde aos elementos restantes da lista.

```
[pam, liz, pat, ann, tom, bob, jim]
%pam é a cabeça, [liz, pat, ann, tom, bob, jim] é a cauda.
[1, 2, 3, 4, 5]
%1 é a cabeça, [2, 3, 4, 5] é a cauda
[a, [b, c], d, e]
%a é a cabeça, [[b, c], d, e] é a cauda
```



Listas

Operador **Pipe** (`|`) – Uma lista não-vazia pode ser representada de forma a apresentar explicitamente a sua cabeça e a sua cauda, usando a sintaxe `[Cabeça|Cauda]`;

```
% lista [b]
?- L = [b|[]]

% lista [a, b]
?- L = [a|[b|[]]]
```

Essa sintaxe é útil em consultas quando queremos decompor uma lista em cabeça e cauda:

```
?- [Head|Tail] = [mia, vincent, jules, yolanda].
Head = mia,
Tail = [vincent, jules, yolanda]

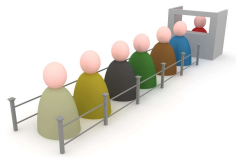
?- [X,Y | W] = [[], dead(zed), [2, [b, chopper]], [], Z].
X = [],
Y = dead(zed),
W = [[2, [b, chopper]], [], Z]
```



Listas

Unificação de listas

Lista 1	Lista 2	Unificação
[mesa]	[X Y]	X/mesa Y/[]
[a,b,c,d]	[X,Y Z]	X/a Y/b Z/[c,d]
[[ana,Y] Z]	[[X,foi],[ao,cinema]]	X/ana Y/foi Z/[[ao,cinema]]
[ano,bissextto]	[X,Y Z]	X/ano Y/bissextto Z/[]
[ano,bissextto]	[X,Y,Z]	não unifica
[data(7,Z,W),hoje]	[X Y]	X/data(7,Z,W) Y/hoje
[data(7,W,1993),hoje]	[data(7,X,Y),Z]	X/W Y/1993 Z/hoje



Listas

Para se trabalhar com uma lista deve-se utilizar unificações, a notação [Cabeca | Cauda] para buscar um elemento ou propriedade através de recursão.

Exemplo 01: Descobrir se um determinado elemento (X) pertence a lista (L):

```
pertence(X, [X | _]). % caso base
pertence(X, [_ | L]) :- pertence(X, L). (regra recursiva).

?- pertence(3,[1,2,3,4,56]).
true.
```

Exemplo 02: Descobrir se um determinado elemento (X) é o último da lista (L):

```
eh_ultimo(X,[X]).
eh_ultimo(X,[_ | Tail]) :- eh_ultimo(X,Tail).

?- eh_ultimo(2,[1,6,7,8,9,2]).
true.
```



Listas

Para se trabalhar com uma lista deve-se utilizar unificações, a notação [Cabeca | Cauda] para buscar um elemento ou propriedade através de recursão.

Exemplo 01: Descobrir se um determinado elemento (X) pertence a lista (L):

```
pertence(X, [X | _]). % caso base
pertence(X, [_ | L]) :- pertence(X, L). (regra recursiva).

?- pertence(3,[1,2,3,4,56]).
true.
```

Exemplo 02: Descobrir se um determinado elemento (X) é o último da lista (L):

```
eh_ultimo(X,[X]).
eh_ultimo(X,[_ | Tail]) :- eh_ultimo(X,Tail).

?- eh_ultimo(2,[1,6,7,8,9,2]).
true.
```



Listas

Exemplo 03: Diga se dois elementos são consecutivos em uma lista.

Dica: Eles são consecutivos se forem o primeiro e o segundo elemento, ou se forem consecutivos na lista:

```
sao_consecutivos(X,Y,[X,Y|_]).  
sao_consecutivos(X,Y,[_|Tail]) :- sao_consecutivos(X,Y,Tail).  
  
?- sao_consecutivos(2,3,[2,3,4,5,2,3,7]).  
true.
```

Exemplo 04: Crie uma regra para descobrir o tamanho de uma lista (L):

```
tamanho([],0).  
tamanho(_|Tail,Tam) :- tamanho(Tail,Tam1), Tam is 1 + Tam1.  
  
?- tamanho([1,2,3,4,5,6,7,8,9], R).  
R = 9
```



Listas – Exercício 11

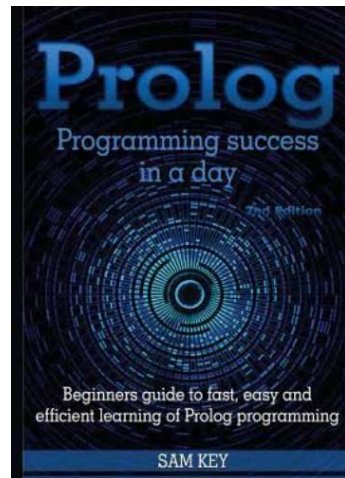
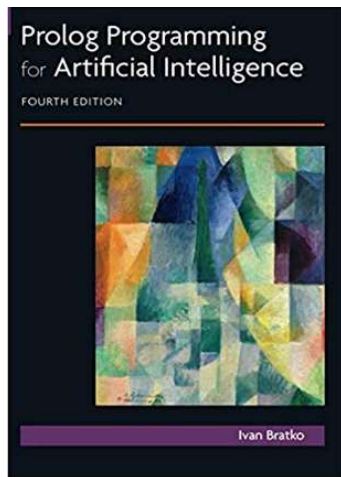
- 1) Crie uma regra que recebe duas listas e informe se a lista L1 é prefixo de L2.
- 2) Crie uma regra que recebe duas listas e informe se a lista L1 é sufixo de L2.
- 3) Crie uma regra que recebe uma lista e retorna outra lista somente com os pares.
- 4) Defina um predicado chamado `todos_as(L)`, que retorna verdadeiro somente se todos os elementos da lista L são o átomo a.
- 5) Defina o predicado `contem_1(L)`, que retorna verdadeiro se a lista L contém pelo menos um elemento 1.
- 6) Considere a seguinte base de conhecimento:

Crie uma regra, `lista_traducao(E,P)`, que traduz uma lista de numerais em inglês na lista correspondente com os numerais em português. Por exemplo, `lista_traducao([one,nine,two],X)` deve retornar `X = [um,nove,dois]`.

```
traducao(one,um).  
traducao(two,dois).  
traducao(three,tres).  
traducao(four,quatro).  
traducao(five,cinco).  
traducao(six,seis).  
traducao(seven,sete).  
traducao(eight,oito).  
traducao(nine,nove).
```


Referencias

Livros



Material online

INTRODUÇÃO À PROGRAMAÇÃO PROLOG Luiz A. M. Palazzo Editora da Universidade Católica de Pelotas / UCPEL Rua Félix da Cunha, 412 - Fone (0532)22-1555 - Fax (0532)25-3105 Pelotas - RS - Brasil EDUCAT

Programação em Prolog - UMA ABORDAGEM PRÁTICA, Eloi L. Favero - Departamento de Informática CCEN – UFPA (Versão 2006) favero@ufpa.br

Slides de Lógica Matemática da professora Joseluze de Farias Cunha