Programação Lógica
Parte 3

PLP-2019
Profa. Heloisa

Controle de retrocesso (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.
- É possível controlá-lo através de um predicado especial chamado **corte**, notado 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.

Controle de retrocesso (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.

Controle de retrocesso (corte)

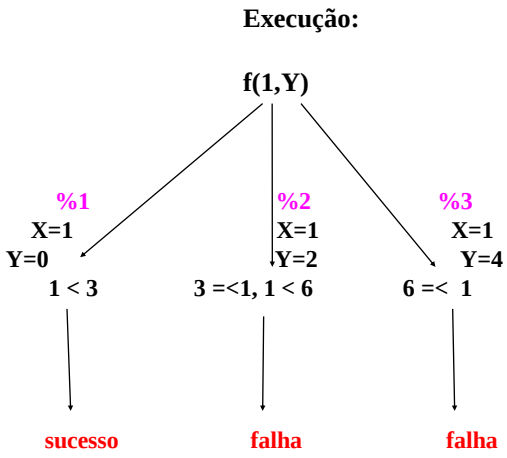
Exemplo:Construir um programa Prolog para implementar a função

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

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

Controle de retrocesso (corte)

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



Controle de retrocesso (corte)

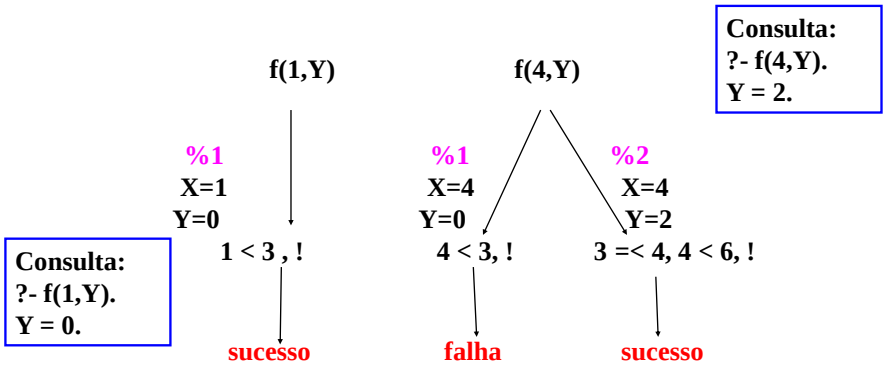
•Na busca, as regras 2 e 3 são tentadas inutilmente, resultando em falha.

•Sabemos, no momento da programação, que as regras representam casos mutuamente exclusivos.

•O uso do corte torna-se conveniente, para evitar esforço de busca desnecessário e tornar a execução da consulta mais eficiente.

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

Controle de retrocesso (corte)



Este tipo de corte é chamado de corte verde : se for retirado, o programa tem exatamente o mesmo significado. Altera-se apenas a eficiência da execução.

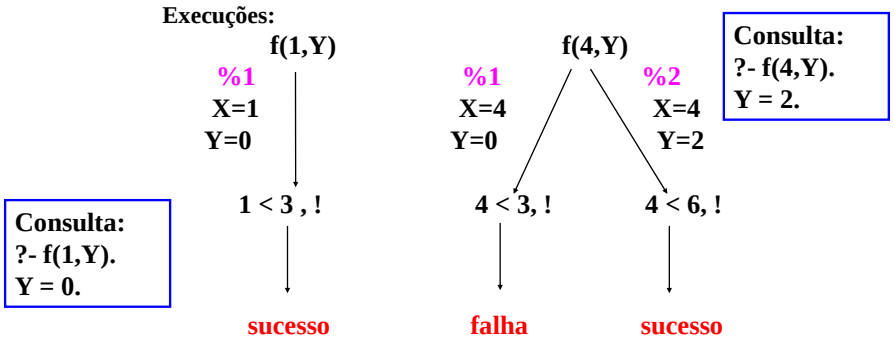
Controle de retrocesso (corte)

- O corte pode também ser usado para tornar o programa mais compacto, sem ter que escrever explicitamente as condições de aplicação de cada regra.

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

- Este tipo de corte é chamado de corte vermelho : quando retirado, o programa tem significado diferente, geralmente produzindo resultados errados.

Controle de retrocesso (corte)



Na consulta f(4,Y), quando a cláusula 1 é usada, 4<3 falha e o corte não é executado, assim existe backtracking e a cláusula 2 é usada.
Retirando-se os cortes, o programa produz resultados errados.

Controle de retrocesso (corte)

- Com o uso do corte, vários programas já estudados podem ser modificados para ficarem mais eficientes.
- Para usar o corte, deve ser analisada a operação que se espera realizar com o predicado.

Exemplos do uso de corte

Eliminar todas as ocorrências de um elemento de uma lista

Versão SEM corte:

```
del_todas(Elem,[ ],[ ]).
del_todas(Elem, [Elem|Y], Z) :-   del_todas(Elem,Y,Z).
del_todas(Elem,[Elem1|Y], [Elem1|Z]) :- Elem \== Elem1,
                                        del_todas(Elem,Y,Z).
```

Versão COM corte:

```
del_todas_2(Elem,[ ],[ ]) :- !.
del_todas_2(Elem, [Elem|Y], Z) :- del_todas_2(Elem,Y,Z), !.
del_todas_2(Elem,[Elem1|Y], [Elem1|Z]) :-
                                        del_todas_2(Elem,Y,Z).
```


Exemplos do uso de corte

Eliminar todas as ocorrências de um elemento de uma lista

```
?- del_todas(a,[a,b,c,a,4,a,c,b],L).
   L = [b, c, 4, c, b] ;
   false.

?- del_todas_2(a,[a,b,c,a,4,a,c,b],L).
   L = [b, c, 4, c, b].
```

Exemplos do uso de corte

Dada uma lista de números, separar em duas sendo uma com os positivos e o zero, e outra com os negativos

Versão SEM corte:

```
separa_sem_corte([ ],[ ],[ ]).
separa_sem_corte ([X|Y],[X|Z],W) :- X >= 0,
                                    separa_sem_corte (Y,Z,W).
separa_sem_corte ([X|Y],Z,[X|W]) :-  X < 0,
                                    separa_sem_corte(Y,Z,W).
```

Versão COM corte:

```
separa([ ],[ ],[ ]) :- !.
separa([X|Y],[X|Z],W) :- X >= 0, separa(Y,Z,W), !.
separa([X|Y],Z,[X|W]) :- separa(Y,Z,W).
```


Exemplos do uso de corte

Dada uma lista de números, separar em duas sendo uma com os positivos e o zero, e outra com os negativos

```
?- separa_sem_corte([1,3,-5,0,-64,37,0,19,-53],P,N).
P = [1,3,0,37,0,19] ,
N = [-5,-64,-53] ;
false.

?- separa([1,3,-5,0,-64,37,0,19,-53],P,N).
   P = [1, 3, 0, 37, 0, 19],
   N = [-5, -64, -53].
```

Exemplos do uso de corte

- Contar o número de ocorrências de um dado elemento no primeiro nível de uma lista:-

```
conta_ocorr(Elem,[ ],0) :- !.

conta_ocorr(Elem,[Elem|Y],N) :-
                                conta_ocorr(Elem,Y,N1),
                                N is N1 + 1, !.

conta_ocorr(Elem,[Elem1|Y], N) :-
                                 conta_ocorr(Elem,Y,N).
```


Exemplos do uso de corte

```
?- conta_ocorr(x,[e,34,x,[e,d,f],par(a,b),x,567,x],S).
   S = 3.

?- conta_ocorr(x,[e,34,x,[e,x,f], x, kfkfkf,5069],S).
    S = 2.

 ?- conta_ocorr(a,[e,l,e,m,e,n,t,o],S).
     S = 0.
```

Exemplos do uso de corte

- Fatorial

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

```
| ?- fatorial(0,F).
F = 1
| ?- fatorial(3,F).
F = 6
| ?- fatorial(-4,F).
false.
```


Alteração da Base de Dados

Alguns predicados do Prolog modificam a base de dados em tempo de execução

abolish(Pred) apaga todos os predicados especificados pelo seu argumento

+Pred especificação de predicado

(ou)

abolish(Pred,Arid)

+Funtor funtor

+Arid número de argumentos

Alteração da Base de Dados

Ex: Base de dados:

```
estudante('Joao').
estudante('Marcia').
estudante('Paulo').
faltas('Joao',5).
faltas('Marcia,2).
faltas('Paulo',3).
```

```
?- abolish(faltas/2).

?- abolish([estudante/1, faltas/2]).

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


Alteração da Base de Dados

assert(Claus) adiciona uma cláusula no fim das cláusulas associadas com seu predicado

+Claus cláusula (fato ou regra)

```
?- assert(gosta(maria,cinema)).
?- assert((gosta(ana,Pessoa) :-
                    gosta(Pessoa, computacao))).
```

Obs:-

- as regras devem aparecer entre parêntesis.

- variáveis não instanciadas no momento da execução do assert são consideradas variáveis.

Alteração da Base de Dados

assert(Claus, Pos) adiciona uma cláusula na posição especificada

+Claus	cláusula (fato ou regra)
+Pos	inteiro >= 0

asserta(Claus) adiciona uma cláusula no começo das cláusulas associadas

assertz(Claus) adiciona uma cláusula no fim das cláusulas associadas

Alteração da Base de Dados

retract(Claus) elimina uma cláusula que unifica a cláusula dada

+Claus	cláusula

Procura a primeira cláusula na base de dados que unifica com Claus, se encontrar apaga.

Variáveis são instanciadas.

Permite backtracking.

Alteração da Base de Dados

- Predicado dynamic

- Indica ao interpretador quais predicados serão alterados na base de dados
- Deve ser definido ANTES da definição do predicado
- Sintaxe:

  :- dynamic pred/1, nome/2,...

- Se um predicado não aparecer na declaração "dynamic" e for usado em um dos predicados de alteração da base de dados, ocorre erro

Alteração da Base de Dados

```
 % Base de dados Matricula
:- dynamic estudante/1.
estudante('Joao').
estudante('Marcia').
estudante('Paulo').
faltas('Joao',5).
faltas('Marcia',2).
faltas('Paulo',3).
curso('Algoritmos');
aceita(Nome, Curso) :- estudante(Nome), curso(Curso),
                       faltas(Nome,N),
                       N < 5.
?- aceita(Aluno,'Algoritmos').
   Aluno = 'Marcia' ;
   Aluno = 'Paulo'.
```

Base de dados antes do retract

```
:- dynamic estudante/1.
estudante('Joao').
estudante('Marcia').
estudante('Paulo').
faltas('Joao',5).
faltas('Marcia',2).
faltas('Paulo',3).
curso('Algoritmos');
aceita(Nome, Curso) :- curso(Curso),estudante(Nome),faltas(Nome,N), N < 5.
```

```
?- listing(estudante/1).
   :- dynamic estudante/1.

   estudante('Joao').
   estudante('Marcia').
   estudante('Paulo').

   true.

?- estudante(X).
   X='Joao';
   X= 'Marcia';
   X= 'Paulo'.
```


Base de dados antes do retract

```
:- dynamic estudante/1.
estudante('Joao').
estudante('Marcia').
estudante('Paulo').
faltas('Joao',5).
faltas('Marcia',2).
faltas('Paulo',3).
curso('Algoritmos');
aceita(Nome, Curso) :- curso(Curso),estudante(Nome),faltas(Nome,N), N < 5.
```

```
?- retract(estudante(X)).
   X='Joao';
   X= 'Marcia';
   X= 'Paulo'.

?- listing(estudante(X)).
   :- dynamic estudante/1.

   true.
```

Base de dados depois do retract

```
faltas('Joao',5).
faltas('Marcia',2).
faltas('Paulo',3).
curso('Algoritmos');
aceita(Nome, Curso) :- curso(Curso),
                       estudante(Nome),
                       faltas(Nome,N),
                       N < 5.
```

Alteração da Base de Dados

Base de dados antes do retract

```
:- dynamic estudante/1, aceita/2.
estudante('Joao').
estudante('Marcia').
estudante('Paulo').
faltas('Joao',5).
faltas('Marcia',2).
faltas('Paulo',3).
curso('Algoritmos').
aceita(Nome, Curso) :- curso(Curso), estudante(Nome),faltas(Nome,N), N < 5.
```

```
?- retract((aceita(N,C) :- B, D, F, E)).
D = curso(C),
B = estudante(N),
F = faltas(N, _G2659),
E = (_G2659<5).
```

Base de dados depois do retract

```
estudante('Joao').
estudante('Marcia').
estudante('Paulo').
faltas('Joao',5).
faltas('Marcia',2).
faltas('Paulo',3).
curso('Algoritmos').
```


Entrada e Saída

- Predicado read
- Sintaxe: read(Termo)

  onde ?Termo (variável ou átomo)
- Lê um termo do dispositivo de entrada corrente e unifica com Termo. O termo dado deve ser seguido de . (ponto).

Entrada e Saída

Exemplos

```
?- read(X),Y is X + 1.
 |: 3.
 X = 3 ,
Y = 4.

?- read(X), read(Y), Z is X+Y.
  |: 3.
  |: 8.
  X = 3,
 Y = 8,
  Z = 11.
```


Entrada e Saída

- Predicado write
- Sintaxe: write(Termo)
  onde ?Termo (termo)
- Escreve o termo no dispositivo de saída corrente
- Predicado nl
- muda para próxima linha no dispositivo de saída

Entrada e Saída

Exemplos

```
?- write(palavra).
palavra
true.
?- write([a,b,c]).
[a,b,c]
true.
?- write(primeira), write('    '), write(segunda).
primeira    segunda
true.
?- write(primeira), nl, write(segunda).
primeira
segunda
true.
```


Predicados sem argumentos e predicados com mesmo nome

- Um predicado é identificado pelo seu nome e pela aridade(número de argumentos).
- Predicados com o mesmo nome e com número de argumentos diferentes são considerados diferentes.
- Os predicados sem argumentos são normalmente usados para identificar procedimentos que usam read e write ou para iniciar programas com muitos predicados.

Predicados sem argumentos e predicados com mesmo nome

Exemplo: Soma dos elementos de uma lista numérica

```
soma :- write('Digite uma lista de numeros'),
        read(Lista),
        soma(Lista,Resultado),
        write('A soma dos elementos da lista e = '),
        write(Resultado),
        nl.
soma([ ],0).
soma([Elem| Cauda], S) ;- soma (Cauda,S1),
                           S is S1 + Elem.
| ?- soma.
Digite uma lista de numeros|: [4,5,6,4.4,0.3,-7].
A soma dos elementos da lista e = 12.7
true.
```


Meta variáveis

- Aparecem no lugar de uma estrutura prolog que pode ser executada.
- **Meta-variável-condição**
- Aparece como um sub-objetivo no corpo de uma regra.
- Não pode aparecer na cabeça da regra.
- Na hora da execução deve estar instanciada com:
  - um átomo
  - um termo composto

Meta variáveis

Exemplo
(Relembrando os exemplos anteriores de conta e soma:)

```
conta([],0).
conta([_|Cauda],N) :-
                conta(Cauda,N1),
                N is N1 + 1.


soma([],0).
soma([Elem|Cauda],S) :- soma(Cauda,S1),
                S is S1 + Elem.
```


Meta variáveis

```
programa1:- write('Entre com a lista de elementos'),
            read(Lista),
            conta(Lista,N),
            write('O numero de elementos e '),
            nl, write(N), nl.

programa2 :-write('Entre com a lista de numeros'),
            read(Lista),
            soma(Lista,N),
            write('A soma dos elementos e '),
            nl, write(N), nl.
```

Meta variáveis

```
principal :- write('Digite o nome do programa a ser
              executado - programa1 ou programa2 - '),
              read(NP),
              NP,
              write(NP), nl.
```


Meta variáveis

```
| ?- principal.
Digite o nome do programa a ser
          executado - programa1 ou programa2 - |:
  programa1.
Entre com a lista de elementos |: [1,2,3,4].
O numero de elementos e
4
programa1
true.
```

Meta variáveis

```
| ?- principal.
Digite o nome do programa a ser
            executado - programa1 ou programa2 - |:
  programa2.
Entre com a lista de numeros|: [3, -5, 0, 45, 1, 2, 7, -3].
A soma dos elementos e
50
programa2
true.
```


Meta variáveis

Outro exemplo

```
programa(X) :- X,
            write(X),nl.

| ?- programa(soma([1,4,6,0,-5,4.7],N)).
soma([1,4,6,0,-5,4.7],10.7)
N = 10.7

| ?- programa(conta([a,b,[a,c,d],[],5],N)).
conta([a,b,[a,c,d],[],5],5)
N = 5
```

Paradigma Lógico – Conclusão
Revisão dos principais conceitos

- Usa cláusulas da lógica de Primeira Ordem
- Processamento Simbólico
  usa símbolos e conceitos ao invés de números e expressões

```
autor('Russel & Norvig').
livro(titulo('Inteligencia Artificial'), autor('G. Bittencourt')).

pai_de(henrique, filhos(eduardo,elizabeth2)).
```


Paradigma Lógico – Conclusão
Revisão dos principais conceitos

- Declarativo
  - Foco da programação: especificar O QUE deve ser feito, sem detalhes de operações da arquitetura da máquina
  - Variáveis não são vistas como células da memória
  - Não existe operação de atribuição
  - Programas são tipicamente especificações de relações

```
separa([ ],[ ],[ ]) :- !.
separa([X|Y],[X|Z],W) :- X >= 0, separa(Y,Z,W), !.
separa([X|Y],Z,[X|W]) :- separa(Y,Z,W).
```

Paradigma Lógico – Conclusão
Revisão dos principais conceitos

- Estrutura principal: listas

[a,b,c]
[X,Y, 25, [3|Z] |Z]

- São baseadas em regras

predecessor(X,Y) :- pai_de(X,Y).
predecessor(X,Y) :- pai_de(X,Z), predecessor(Z,Y).

Paradigma Lógico – Conclusão
Revisão dos principais conceitos

- A ordem de especificação das regras não é fundamental
- A ordem de execução é determinada na própria execução

```
%programa
conta([],0).
conta([_|Cauda],N) :- conta(Cauda,N1), N is N1 + 1.
pertence(X,[X|_]):-!.
pertence(X,[_|Y]):- pertence(X,Y).
soma([],0).
soma([Elem|Cauda],S) :- soma(Cauda,S1), S is S1 + Elem.
programa(S):- write('Digite uma lista de números '),
      read(L),
      soma(L,S),
      nl, write('A soma eh ').
```

```
?-programa(S).
Digite uma lista de números [4,6,-1,0,-5,6,12].

A soma eh
S = 22.
```