

# Retrocesso e corte

---

Marco A L Barbosa  
malbarbo.pro.br

Departamento de Informática  
Universidade Estadual de Maringá



Este trabalho está licenciado com uma Licença Creative Commons - Atribuição-CompartilhaIgual 4.0 Internacional.  
<http://github.com/malbarbo/na-proglog>

**Retrocesso**

- Retrocesso é basicamente uma forma de busca

- Suponha que o Prolog está tentando satisfazer a sequência de metas  $m_1, m_2$ . Quando o Prolog encontra um conjunto de unificações de variáveis que tornam a meta  $m_1$  verdadeira, ele se compromete com estas unificações e tenta satisfazer a meta  $m_2$ . Em seguida duas coisas podem acontecer

1. A meta  $m_2$  é satisfeita com um conjunto de unificações. O usuário pode pedir para o Prolog buscar soluções alternativas. Neste caso o Prolog desfaz o conjunto de unificações e tenta encontrar um outro conjunto de unificações que torne  $m_2$  verdadeiro, se não for possível, o Prolog retrocede para  $m_1$ , desfaz as unificações para esta meta e tenta satisfazer  $m_1$  novamente obtendo um (novo) conjunto de unificações
2. A meta  $m_2$  não é satisfeita. Nesta caso o Prolog retrocede para a  $m_1$ , desfaz as unificações para esta meta e tenta satisfazer  $m_1$  novamente obtendo um (novo) conjunto de unificações

- Retrocesso pode ocorrer para buscar soluções extras para o conjunto de metas ou para buscar a primeira solução quando um comprometimento com conjunto de unificações não levou a uma solução

- Dado o programa a seguir, qual o resultado produzido na consulta `possivel_par(X, Y)` se o usuário pressionar ; após cada resposta?

```
possivel_par(ana, pedro).
```

```
possivel_par(X, Y) :-  
    mulher(X),  
    homem(Y).
```

```
mulher(claudia).
```

```
mulher(marcia).
```

```
mulher(elsa).
```

```
homem(joao).
```

```
homem(paulo).
```

# Retrocesso

```
?- possivel_par(X, Y).  
X = ana,  
Y = pedro ;  
X = claudia,  
Y = joao ;  
X = claudia,  
Y = paulo ;  
X = marcia,  
Y = joao ;  
X = marcia,  
Y = paulo ;  
X = elsa,  
Y = joao ;  
X = elsa,  
Y = paulo.
```

- A maioria dos predicados pré-definidos do Prolog quando usados em metas com (várias) variáveis podem produzir mais do que uma resposta

```
?- member(3, [1, 3, 2, 4]).
```

```
true ;
```

```
false.
```

```
?- member(X, [1, 3, 2, 4]).
```

```
X = 1 ;
```

```
X = 3 ;
```

```
X = 2 ;
```

```
X = 4.
```

```
?- between(1, 4, X).
```

```
X = 1 ;
```

```
X = 2 ;
```

```
X = 3 ;
```

```
X = 4.
```

```
?- between(3, inf, X).
```

```
X = 3 ;
```

```
X = 4 ;
```

```
X = 5 ;
```

```
X = 6 ;
```

```
...
```

## Retrocesso

```
?- select(X, [a, b, c], R).
```

```
X = a,
```

```
R = [b, c] ;
```

```
X = b,
```

```
R = [a, c] ;
```

```
X = c,
```

```
R = [a, b] ;
```

```
false.
```

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

```
L = [d, a, b, c] ;
```

```
L = [a, d, b, c] ;
```

```
L = [a, b, d, c] ;
```

```
L = [a, b, c, d] ;
```

```
?- permutation([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] ;
```

```
false.
```

```
?- nth0(X, [a, b, c], E).
```

```
X = 0,
```

```
E = a ;
```

```
X = 1,
```

```
E = b ;
```

```
X = 2,
```

```
E = c.
```

```
?- append(X, Y, [1, 2, 3]).  
X = [],  
Y = [1, 2, 3] ;  
X = [1],  
Y = [2, 3] ;  
X = [1, 2],  
Y = [3] ;  
X = [1, 2, 3],  
Y = [] ;  
false.
```

```
?- prefix(A, [1, 2, 3]).
```

```
A = [] ;
```

```
A = [1] ;
```

```
A = [1, 2] ;
```

```
A = [1, 2, 3] ;
```

```
false.
```

## Exemplo 4.1

Defina um predicado `seleccionado(?N, ?L, ?R)` que é verdadeiro se a lista `R` é como a lista `L` mas sem o elemento `N`. (Mesmo comportamento do predicado pré-definido `select/3`).

## Exemplo 4.2

Defina um predicado `permutacao(+L, ?P)` que é verdadeiro se a lista `P` é uma permutação da lista `L`. (Mesmo comportamento do predicado pré-definido `permutation/2`).

**Geradores**

- Predicados que produzem mais de uma resposta podem ser chamados de gerados
- As vezes é interessante ter predicados que possam ser satisfeitos infinitas vezes, gerando um conjunto de unificações a cada vez. Isto é particularmente interessante na estratégia gerar e testar

## Exemplo 4.3

Defina um predicado `natural(?N)` que é verdadeiro se `N` é um número natural. Se `N` não estiver instanciado, o predicado deve gerar números naturais.

**Corte**

- As vezes é necessário interromper o processo de retrocesso. Para isso utilizamos o operador de corte
- A meta corte é especificada com o predicado !
- Quando a meta ! é encontrada ela é satisfeita imediatamente mas não pode ser ressatisfeita, isto é, quando um corte é encontrado como uma meta, o sistema compromete-se com todas as escolhas feitas desde que a meta pai foi invocada. Todas as outras alternativas são descartadas. Ou seja, uma tentativa de ressatisfação feita em uma meta entre a meta pai e a meta corte irá falhar

- No exemplo abaixo o Prolog irá fazer o retrocesso entre as metas  $a$ ,  $b$ ,  $c$  até que a meta  $c$  seja satisfeita. Quando a meta  $c$  for satisfeita a meta corte será satisfeita em seguida. Deste momento em diante o retrocesso pode acontecer entre as metas  $d$ ,  $e$ ,  $f$ , mas uma vez que a meta  $d$  não possa mais ser satisfeita, o Prolog não retrocede para tentar ressatisfazer as metas  $a$ ,  $b$  e  $c$ , e portanto a meta  $m$  irá falhar (possivelmente depois de ser satisfeita várias vezes)

```
m :- a, b, c, !, d, e, f.
```

- Existem alguns usos comuns para o corte, entre eles estão
  - Confirmação de escolha
  - Junto com o predicado `fail`
  - Na estratégia gerar e testar

## Confirmação de escolha

- Até o momento usamos o operador de corte para confirmar uma escolha, ou seja, para informar para o Prolog que ele não precisa buscar respostas alternativas
- Este uso ocorre comumente quando queremos que um predicado se parece com uma função (veja os exemplos de dados compostos que utilizam corte).

## Confirmação de escolha

- Nestes casos o corte é apenas uma otimização, sendo possível escrever o predicado com a mesma semântica sem utilizar o corte. Considere o exemplo onde exista duas escolhas excludentes para um determinado predicado que depende de uma determinada condição

a :- b, c.

a :- \+b, d.

- O Prolog pode tentar satisfazer a meta b duas vezes. Isto pode ser ineficiente se a satisfação de b for custosa. Neste caso, o uso do corte pode tornar o predicado mais eficiente

a :- b, !, c.

a :- d.

## Combinação com fail

- As vezes é necessário dizer ao Prolog para falhar imediatamente uma meta específica sem ter que tentar soluções alternativas
- Por exemplo, suponha que uma determinada meta possa ser satisfeita de várias maneiras, mas é possível identificar condições que a meta deve falhar (sem ter que tentar estas muitas maneiras). Então é possível combinar o corte com o predicado `fail` para evitar que estas muitas maneiras sejam verificadas

## Exemplo 4.4

Defina um predicado aprovado ( $A$ ) que é verdadeiro se o aluno  $A$  foi aprovado. Um aluno pode ser aprovado se ele obteve média maior ou igual a 6 ou se a após o exame ele obteve média maior ou igual a 5. Em ambos os casos ele deve ter pelo menos 75% de presença.

# Implementação de falha e negação

- Como implementar o predicado `fail`?

```
fail :- 0 = 1.
```

- Como implementar o operador de negação?
- O operador `\+` pode ser implementado da seguinte forma

```
\+(P) :- call(P), !, fail.
```

```
\+(P).
```

- Uma estratégia comum utilizada em programas Prolog envolve a geração de diversas possíveis soluções (via retrocesso) para um determinado problema seguido do teste para validar a solução. Após encontrar a solução o processo é interrompido com o operado de corte.

## Exemplo 4.5

Defina um predicado `primeiro_primo(+N, ?P)` que é verdadeiro se `P` é o primeiro primo maior que ou igual a `N`. Use a estratégia gerar e testar, gere números inteiros a partir de `N` e teste se ele é primo.

## Exemplo 4.6

Defina um predicado `caminho_hamiltoniano(+G, -C)` que é verdadeiro se `C` é um caminho hamiltoniano (lista de vértices) do grafo `G`. Um caminho hamiltoniano é um caminho que passa exatamente uma vez por cada vértice de `G`. Use a estratégia gerar e testar, gere permutações dos vértices e verifique se forma um caminho.

## Referências

- Capítulo 4 do livro Programming in Prolog
- Capítulo 5 da apostila Paradigmas de programação - Prolog
- Capítulo 10 do livro Learn Prolog Now