

# 1 Alguns predicados não lógicos

Até agora, temos nos ocupado mais com aspectos teóricos de programação lógica. Na prática, ainda temos alguns problemas a resolver.

A começar pela avaliação do programa: computadores comuns não vem equipados com uma placa de clarevidência, e, assim, podem não conseguir adivinhar bem o caminho para a prova de um goal. Isto é, a hipótese de não-determinismo não segue na prática. O ideal seria que, se um goal pode ser provado por um programa, o processo de prova seguisse um caminho direto, sem tentativas de unificações infrutíferas. Na prática, isso só é realizável para situações muito específicas e, no geral, serão tentadas diversas unificações infrutíferas antes de se chegar ao objetivo.

Para lidar com isso é criado um *choice point* logo antes de se tentar uma unificação. Se a tentativa resulta em falha, o processo volta ao estado de antes da tentativa, no que chamamos de backtracking, a possibilidade daquela unificação é eliminada e o processo continua. O goal falha quando todas as possibilidades foram eliminadas e tem sucesso quando não existirem mais unificações a serem feitas.

**Choice point**

**Backtracking**

Os *choice points* são um ponto chave na execução de um programa lógico. A quantidade deles está diretamente relacionada com a eficiência do programa: quanto mais choice points, em geral, mais ineficiente o programa é. Assim, se o mesmo goal puder ser provado de mais de uma forma diferente, é necessária a criação de *choice points* a mais, o que deve ser evitado. Da mesma forma, se existe mais de um resultado para uma computação (isto é, se o mesmo goal admite diversas substituições que resultam em sucesso), a eliminação das opções a mais implicariam na eliminação de *choice points*, o que seria vantajoso.

A quantidade de *choice points* tem muito a ver com a de “axiomas” (isto é, tem a ver com a quantidade de cláusulas e com tamanho do corpo das cláusulas). Geralmente, um programa com menos axiomas resulta em melhor legibilidade e maior eficiência. No entanto, alguns desses axiomas representam restrições que podem levar uma falha mais cedo na computação, o que, na prática, diminui a quantidade de *choice points* (os que seriam criados, não houvesse essa falha, não seriam mais) e de unificações desnecessárias, aumentando a eficiência do programa.

Veremos, então, métodos para lidar com essas situações. O mais importante, e mais polêmico, é o **corte**, o **!/0** (um funtor de nome “!” e aridade 0). Intuitivamente, o que ele faz é se comprometer com a escolha atual, descartando as demais. Só poderemos compreender completamente como ele funciona depois que desenvolvermos a interpretação de uma computação lógica como a busca em uma árvore, mas, enquanto isso, nossa interpretação intuitiva será o suficiente.

Como um exemplo de seu uso, considere o seguinte programa:

Código 1: Member

$\text{member}(X, [X | Xs])$ .

$\text{member}(X, [_|Xs]) :- \text{member}(X, Xs).$

O underline “\_” representa uma **variável anônima**. A utilizamos quando o nome daquela variável não é importante, o que acontece quando não a utilizarmos novamente (isso serve para não nos distrairmos com variáveis que não serão utilizadas: a variável anônima cumpre um papel meramente formal). Toda variável anônima é diferente.

**Variável  
anônima**

O goal  $\text{member}(X, Xs)?$  resulta em sucesso se  $X$  é um elemento de  $Xs$ <sup>1</sup>. Esse programa é usado primariamente de duas formas: para checar se um elemento é membro de uma lista; ou para gerar em  $X$  os elementos da lista. Por exemplo, para o goal  $\text{member}(X, [a, b, c])?$ ,  $X$  pode assumir o valor de  $a$ ,  $b$  ou  $c$ .

Se a programadora quiser saber apenas se um certo  $X$  faz parte de uma certa lista  $Xs$ , ela pode usar o corte, “cortando” as demais soluções:

#### Código 2: Member

$\text{member}(X, [X|Xs]).$   
 $\text{member}(X, [_|Xs]) :- \text{member}(X, Xs), !.$

O que o corte “diz” essencialmente é: “Tudo bem, agora que chegamos a este ponto, não olhe para trás”. O uso desse novo programa para *member* é o mesmo do anterior, exceto que, agora, ao gerar membros da lista, só é gerado um membro e, ao checar se outro elemento pertence a lista, assim que se chega ao sucesso, o processo para.

Mencionamos anteriormente que o corte é usado para fins de eficiência. Quando ele é usado somente para esse fim, ou seja, se ele só serve para fazer o programa rodar melhor, não interferindo no seu significado, dizemos que o corte é verde. Quando o corte não é verde, dizemos que ele é vermelho. O que acabamos de ver foi um corte vermelho: goals como  $\text{member}(b, [a,b,c])?$  e  $\text{member}(c, [a,b,c])?$  não estão mais no significado. Perceba que o corte não é um predicado lógico, mas sim operacional: ele nos diz algo sobre “como” rodar um programa mas, a princípio, não sobre “o que” ele é.

**Corte verde  
Corte Ver-  
melho**

O corte é, provavelmente, o predicado não lógico mais importante do Prolog. O que o torna polêmico é justamente o fato de ser não lógico. Dissemos anteriormente que a ideia da programação lógica era lidar com a parte da lógica (o “o que fazer”), abstraindo a parte procedural (o “como fazer”). O corte é um exemplo em que isso não foi obtido.

Apesar disso, se deixarmos de lado nosso preconceito, o corte pode contribuir de maneiras interessantes para a lógica do programa.

Por exemplo, o  $\text{not}/1$  pode ser implementado de maneira similar à seguinte:

<sup>1</sup>Note que estamos assumindo tácitamente, neste programa e nos demais, que a ordem de execução é “de cima para baixo, da direita para a esquerda” (o interpretador “enxerga” primeiro a cláusula que aparece “antes”), que é como as implementações usuais de Prolog funcionam

### Código 3: Not

```
not(Goal) :- Goal, !, fail.  
not(Goal).
```

Onde o predicado fail/0 é um predicado do prolog que falha ao ser executado.

Similarmente, podemos, a partir do corte, gerar outras formas de controle, familiares a programadores de linguagens procedurais:

### Código 4: Se entao se nao

```
se_entao_senao(A, B, R) :-  
    A, !, B.  
se_entao_senao(A, B, R) :-  
    R.
```

### Código 5: Ou

```
or(A, B) :- A, !.  
or(A,B) :- B.
```

Perceba que, diferentemente do que ocorre em programas puramente lógicos<sup>2</sup>, a ordem das cláusulas e dos termos em cada cláusula podem ser fundamentais. Pode ser que, por exemplo, na cláusula  $p(a) :- b, c.$ ,  $b$  resulte em falha e  $c$  em um processo interminável. Neste caso, a mudança de ordem seria fatal.

Os programas acima demonstram uma possível forma de se definir, respectivamente, o “se,então, se não” e o “Ou”<sup>3</sup>, mas essas construções já existem no Prolog por default, como:

- se, então, se não:  $A \rightarrow B ; R.$  (lê-se: se  $A$ , então  $B$ , se não  $R$ );
- ou:  $A ; B.$  (lê-se:  $A$  ou  $B$ ).

## Referências

---

<sup>2</sup>Você pode se perguntar se algum do programa nesse texto é puramente lógico. Alguns dos mais simples, como o *Natural*, do capítulo 2, podem ser, mas a situação geral é que os programas que veremos são só “meio que” lógicos.

<sup>3</sup>Na verdade, poderíamos pensar como o processo de backtracking como o nosso “ou” natural, mas às vezes pode ser conveniente usar um “ou” em uma cláusula, seja por efeito de legibilidade ou para evitar o backtracking.