

Ficha Prática 9

9.1 Objectivos

1. Exercícios com *cut* e *fail*.
2. Exercícios com Árvores de Prova (continuação).

9.2 Conceitos

9.2.1 *cut* — !

O *cut* (!) é um predicado que impede o *backtracking*. Dito de outro modo, o *cut* permite “cortar” ramos de uma árvore de prova. Assim, pode ser utilizado para melhorar a eficiência de um programa Prolog, cortando ramos que à partida se sabe que irão falhar.

Considere o seguinte predicado:

```
maior(X,Y,X) :- X>=Y.  
maior(X,Y,Y) :- X<Y.
```

A árvore de prova para a *query* `?- maior(4,2,R)` é a apresentada na figura 9.10. Como

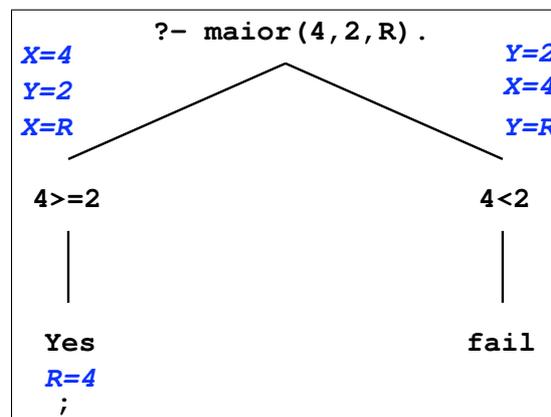


Figura 9.10: Árvore de prova para maior

se pode verificar pela árvore apresentada, o interpretador tenta aplicar a segunda regra durante o processo de *backtracking* para procurar soluções alternativas à solução inicial (`R=4`). Como seria de esperar a segunda regra falha. Na verdade as duas regras são mutuamente exclusivas.

Numa situação de regras mutuamente exclusivas, é possível utilizar o *cut* para indicar que, quando uma das regras é aplicável, não deverão ser tentadas as restantes. Neste caso o predicado ficaria:

```
maior2(X,Y,X) :- X>=Y, !.
maior2(X,Y,Y) :- X<Y.
```

e a árvore de prova passaria a ser a apresentada na figura 9.11. Neste caso o ramo direito

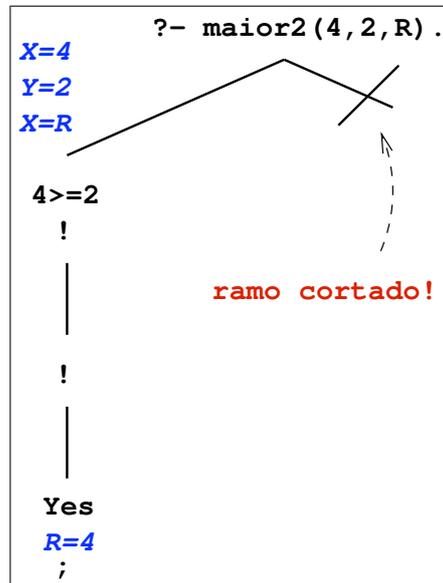


Figura 9.11: Árvore de prova para maior2

da árvore é cortado pois, quando o *backtracking* é iniciado e se tenta fazer o *redo* do *cut*, o objectivo falha sem se tentarem outras soluções.

green cut

É importante notar que, no caso apresentado, o *cut* apenas corta ramos da árvore que vão falhar. Se o *cut* fosse removido o predicado produziria os mesmos resultados mas calcularia árvores de prova maiores. A este tipo de *cut*, que não altera a semântica do predicado, chama-se um *green cut*.

red cut

Considere-se agora a seguinte versão do predicado:

```
maior3(X,Y,X) :- X>=Y, !.
maior3(_,Y,Y).
```

Nesta nova versão, procura aproveitar-se o facto de o *cut* da primeira regra impedir o *backtracking* para a segunda regra para simplificarmos esta última. No entanto, se agora removermos o *cut* da primeira regra o significado do predicado é afectado. A este tipo de *cut*, que altera a semântica do predicado, chama-se *red cut*.

Deve evitar-se a utilização de red cuts pois, para além de tornarem a compreensão do código mais difícil, podem também dar origem a resultados inesperados quando mal aplicados (e aplicar bem um *red cut* é mais difícil que aplicar bem um *green cut*!). Tente, por exemplo, a seguinte *query*:

```
?- maior3(4,2,2).
```

O que se passou?!

9.2.2 fail — negação por falha

O *fail* é um predicado que falha sempre. Utilizando o *cut* e o *fail*, é possível escrever o predicado negação:

```
negacao(A) :- A, !, fail.
negacao(_).
```

Note-se que este predicado não produz unificações. Porquê?!

9.3 Exercícios

9.3.1 Cut e Árvores de Prova

1. Para a definição:

```
p(1).
p(2):- !.
p(3).
```

Diga qual o resultado das seguintes queries:

- $p(X)$.
- $p(X), p(Y)$.
- $p(X), !, p(Y)$.

Para cada uma das queries faça a respectiva árvore de prova.

2. Para o programa Prolog que se apresenta:

```
q(a).
q(b).
q(c).
r(b,b1).
r(c,c1).
r(a,a1).
r(a,a2).
p(X,Y):- q(X), r(X,Y).
p(d,d1).
p1(X,Y):- q(X), r(X,Y), !.
p1(d,d1).
p2(X,Y):- q(X), !, r(X,Y).
p2(d,d1).
p3(X,Y):- !, q(X), r(X,Y).
p3(d,d1).
```

efectue as árvores de prova para as queries seguintes:

- $p(X, Y)$.
- $p1(X, Y)$.
- $p2(X, Y)$.
- $p3(X, Y)$.

