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

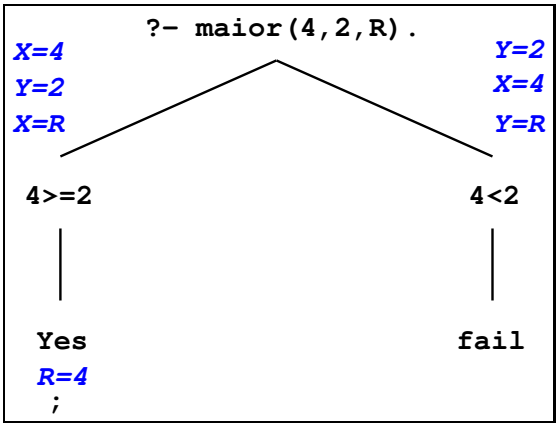

Figura 9.10: Árvore de prova para maior

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 paresentada na figura 9.11. Neste caso o ramo direito

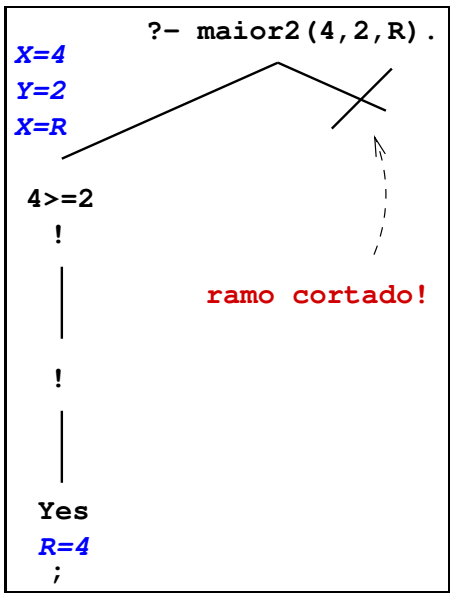

Figura 9.11: Árvore de prova para maior2

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 progama 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).`

9.3.2 Exercícios com cuts

Para cada um dos seguintes enunciados, escreva predicados recorrendo a *green cuts* sempre que possível:

1. Crie um predicado `classe/2`, que determine se um determinado número é `positivo`, `negativo` ou `zero`.
2. Utilize o predicado definido anteriormente por forma a criar um outro, denominado `split/3`, que dada uma lista de números inteiros, origine duas outras listas, uma com os números positivos e zero e uma outra com os números negativos.

   Faça a árvore de prova para a *query*:
   `?- split([1,-1,3,0,-5,-2], LPositivosZero, LNegativos).`
3. Crie um predicado `ifthenelse/3`, que implemente a estrutura condicional if...then...else.
4. Recorrendo ao uso de *cuts*, formule o predicado `interseccao/3` que implementa a intersecção de listas (vistas como conjuntos).
5. Crie um predicado que dada uma lista de inteiros, devolva a lista dos inteiros que ocorrem mais do que duas vezes nessa lista.
6. Crie um predicado `diferentes/2`, que dê verdadeiro se os dois parâmetros do predicados forem de facto diferentes.
7. Crie um predicado `diferenca/3`, que efectue a diferença de listas, isto é, cria uma lista com os elementos da primeira lista que não se encontram na segunda. Utilize o predicado `negação` definido na secção 9.2.2.
8. Considere o seguinte predicado:

   ```
   % norep(L1,L2) :- L2 é a lista L1 sem elementos repetidos
   norep([],[]).
   norep([H|T1],L) :- member(H,T1), norep(T1,L).
   norep([H|T1],[H|T2]) :- \+ member(H,T1), norep(T1,T2).
   ```

   O predicado funciona mas tem um problema:

   ```
   ?- norep([a,d,a,d,a],L).
   L = [d, a] ;
   L = [d, a] ;
   No
   ```

   O predicado calcula o resultado certo, mas mais que uma vez!

   Utilizando o `trace` e/ou a árvore de prova existente na página de PPIII, identifique onde está o problema e procure resolvê-lo utilizando *cuts*.
9. Escreva o predicado `del/3`:

   ```
   % del(E,L1,L2) :- L2 é a lista L1 com todas as ocorrências
   %                 de E removidas
   ```
10. Escreva o predicado `quicksort/2`:

    ```
    % quicksort(L1,L2) :- L2 é uma versão ordenada de L1
    %                     (utilizando o algoritmo quicksort)
    ```