Ficha Prática 6

1. Praticar a escrita de predicados para manipulação de listas.
2. Utilizar o `trace` para *debugging*.

6.1 Conceitos

6.1.1 Predicados sobre listas

A tabela 6.3 apresenta alguns dos predicados para manipulação de listas presentes no SWI-Prolog. Para mais informações, consultar as secções 4.29 a 4.31 do manual.

member(?Elem, ?List)	`Elem` existe em `List`
append(?List1, ?List2, ?List3)	`List3` é a concatnação de `List1` e `List2`
delete(+List1, ?Elem, ?List2)	`List2` é `List1` com todos os `Elem` removidos
select(?Elem, ?List, ?Rest)	`Elem` é um elemento de `List`, `Rest` são os restantes
nth0(?Index, ?List, ?Elem)	elemento na posição `Index` é `Elem` (contagem a partir de 0)
nth1(?Index, ?List, ?Elem)	elemento na posição `Index` é `Elem` (contagem a partir de 1)
last(?Elem, ?List)	`Elem` é o último elemento de `List`
reverse(+List1, -List2)	`List2` é a lista `List1` invertida
length(?List, ?Int)	`Int` é o número de elementos em `List`

Tabela 6.3: Alguns predicados sobre listas do SWI-Prolog

6.1.2 *Debug*

Numa visão procedimental da semântica de programas Prolog, cada predicado pode ser visto como uma caixa com quatro *portas* (ver figura 6.8):

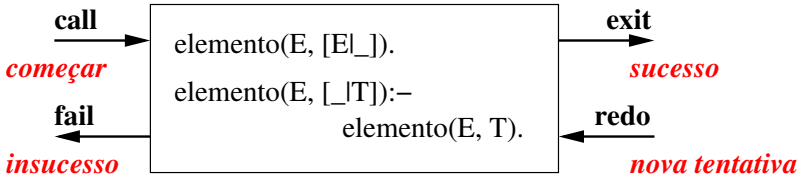

Figura 6.8: Predicados prolog como *caixas*

- **call** — corresponde a tentar aplicar o predicado;
- **exit** — corresponde a ter aplicado o predicado com sucesso;
- **redo** — corresponde a tentar encontrar uma nova solução;

- **fail** — corresponde à falha da última tentativa de aplicar o predicado.

Para fazer *debug* em Prolog utiliza-se o predicado `trace`. Quando o `trace` está activo, o interpretador pára sempre que uma das portas apresentadas acima é activada. De seguida apresenta-se um exemplo (texto em *itálico* corresponde a comentários):

```
?- trace.

Yes
```
% Foi activado o trace.
```
[trace]  ?- elemento(E,[a,b,c]).
```
% Call: vai ser utilizado o predicado elemento:
```
   Call: (6) elemento(_G286, [a, b, c]) ? creep
```
% `creep` corresponde a continuar (basta premir Enter).
```
   Exit: (6) elemento(a, [a, b, c]) ? creep
```
% Exit: sucesso! Utilizou a primeira regra (um facto) e a variável ficou unificada com o átomo `a`.

```
E = a ;
```
% Como respondemos com ";" (equivalente a forçar o Fail da solução apresentada) o Prolog vai procurar uma nova solução para o último Call terminado com sucesso. Para isso utiliza a porta Redo.
```
   Redo: (6) elemento(_G286, [a, b, c]) ? creep
```
% Utilizando a segunda regra, vai aplicar `elemento` *à cauda da lista (o número entre parêntesis indica o aninhamento da invocação de predicados).*
```
   Call: (7) elemento(_G286, [b, c]) ? creep
   Exit: (7) elemento(b, [b, c]) ? creep
```
% Sucesso!
```
   Exit: (6) elemento(b, [a, b, c]) ? creep
```
% Como conseguimos resolver o corpo da regra, resolvemos a cabeça.

```
E = b ;
   Redo: (7) elemento(_G286, [b, c]) ? creep
```
% Tenta-se sempre o Redo do último call. É por isso que aqui saltou para o nível sete...
```
   Call: (8) elemento(_G286, [c]) ? creep
   Exit: (8) elemento(c, [c]) ? creep
   Exit: (7) elemento(c, [b, c]) ? creep
   Exit: (6) elemento(c, [a, b, c]) ? creep

E = c ;
   Redo: (8) elemento(_G286, [c]) ? creep
```
% ... e aqui para o nível oito!
```
   Call: (9) elemento(_G286, []) ? creep
   Fail: (9) elemento(_G286, []) ? creep
```
% Para a lista vazia o predicado falha! (como não há mais regras que se possam aplicar, vai falhar a querie.)
```
   Fail: (8) elemento(_G286, [c]) ? creep
   Fail: (7) elemento(_G286, [b, c]) ? creep
   Fail: (6) elemento(_G286, [a, b, c]) ? creep

No
[debug]  ?-
```

6.2 Exercícios

6.2.1 Manipular listas

Escreva os seguintes predicados sobre listas (tal como na Ficha 5, para alguns deles já existe equivalente em Prolog — continua a valer a pena o exercício)::

1. `conc/3`:

   `conc(L1,L2,L)` :- `L` é a concatenação de `L1` e `L2`.

2. `inverte/2`:

   `inverte(L1,L2)` :- a lista `L2` é a lista `L1` invertida.

3. `interseccao/3`:

   `interseccao(L1,L2,L)` :- `L` é a intersecção de `L1` e `L2`.
   Considere que as listas `L1` e `L2` não possuem elementos repetidos e estão ordenadas por ordem crescente.

4. `filtrapar/2`:

   `filtrapar(L1, L2)` :- a lista `L2` tem os elementos em posição par na lista `L1`.

5. `apaga_primeiro/3`:

   `apaga_primeiro(L1,X,L2)` :- `L2` é o resultado de retirar a `L1` a primeira ocorrência de `X`.

6. `apaga/3`:

   `apaga(E,L1,L2)` :- `L2` é a lista `L1` com os elementos iguais a `E` removidos.

6.2.2 Problemas de *Debugging*

1. Considere a definição de elemento apresentada na figura 6.8. Com esta definição o predicado repete soluções. Faça um *dry run* da query

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

   para descobrir a causa do problema. Confirme o seu *dry run* utilizando o `trace` no Prolog.

2. Considere a seguinte definição:

   ```
   conta(E, L, N) :- conta(E, L, 0, N).
   conta(_, [], Ac, Ac).
   conta(E, [E|T], Ac, N) :- Ac1 is Ac+1, conta(E, T, Ac1, N).
   conta(E, [_|T], Ac, N) :- conta(E, T, Ac, N).
   ```

   Este predicado não está correctamente definido:

   ```
   ?- conta(a, [a,b,a,c], X).
   X = 2;
   X = 1;
   X = 1;
   X = 0;
   No
   ```

   Faça um *dry run* da query para descobrir a causa do problema. Confirme o seu *dry run* utilizando o `trace` no Prolog.

6.2.3 Problemas com Listas

1. Relembre o problema 3.3.2 (página 13), escreva os seguintes predicados:

   (a) `ha_ligacao/3`:

   `ha_ligacao(A,B,C)` :- existe ligação entre `A` e `B` percorrendo o caminho `C`.

   Considere que os caminhos deverão ser representados por listas de triplos `O-T-D`, em que cada triplo indica que se vai de `O` até `D` utilizando o transporte `T`.

   (b) `ha_ligacao/4`:

   `ha_ligacao(A,B,C,T)` :- existe ligação entre `A` e `B` percorrendo o caminho `C` e utilizando apenas o meio de transporte `T`.

   Neste caso os caminhos deverão ser representados por listas contendo os nomes das cidades por onde se passa.

2. Relembre o exercício 5 da secção 4.3.2 (página 16).

   Escreva o predicado `serieFibonacci/2`:

   `serieFibonacci(N, L)` :- `L` é uma lista contendo a série de Fibonacci até ao `N`-ésimo elemento.