Inteligência Artificial
Apontamentos para as aulas

Luís Miguel Botelho

Departamento de Ciências e Tecnologias da Informação

ISCTE-IUL - Instituto Universitário de Lisboa

Dezembro de 2016

Notas sobre *Prolog* Procedimental

NOTA: Estas notas são insuficientes em diversos aspetos. São notas avulsas. As explicações e os exemplos são também insuficientes. Elas resultam essencialmente de respostas dadas a pedidos de ajuda que nos fizeram chegar por *email*. Não servem, por isso, como material suficiente para se estudar a matéria. O livro "Programming in Prolog", de Clocksin & Mellish, é a fonte de consulta aconselhada para toda a linguagem *Prolog*.

Índice

1 Mecanismos de repetição em Prolog

Na linguagem de programação Prolog, existem essencialmente duas maneiras de originar um processo computacional repetitivo. Uma das maneiras, frequentemente usada na programação declarativa, é a recursividade. Quando um predicado se define com base em si mesmo, gera-se uma repetição.

A outra via para se obter uma repetição consiste em combinar a falha com a possibilidade de obter soluções alternativas para determinadas computações.

Estas notas sobre repetição dizem respeito exclusivamente a esta segunda via, isto é, à repetição por falha.

2 Efeito do *fail* na repetição

O efeito que o *fail* tem num programa tem que ver com a maneira como o Prolog funciona, em particular com o conceito de *backtracking* (retrocesso). Mas o *fail* não tem efeitos secundários, como tem o *cut* (!). Trata-se apenas de um predicado que falha.

O Prolog procura ter sucesso nos problemas que tem de resolver. Sempre que falha, numa determinada altura da execução, o Prolog retrocede para o passo anterior àquele em que falhou para ver se encontra uma solução alternativa nesse passo que lhe permita não falhar no passo seguinte. Se o passo anterior à falha não tiver soluções alternativas ou se as soluções alternativas que tem se esgotarem sem que o Prolog consiga ter sucesso, o Prolog retrocede dois passos em vez de um. E por aí adiante.

Sempre que pretendemos que um programa Prolog repita uma determinada computação (i.e., sempre que pretendemos um ciclo), temos de fazer com que o Prolog falhe (usando o *fail*, ou uma outra coisa que falhe).

Repara no programa

```
p :-
    member(X, [a, b, c]),
    write(X), nl,
    fail.
```

Se fores ao interpretador e exe*cut*ares este programa, o que acontece? Porquê?

```
?- p.
a
b
c
False
```

Quando o procedimento *p* é exe*cut*ado, o Prolog avalia o objetivo *member(X, [a, b, c])*, instanciando o X (X =a, que é a primeira solução de *member(X, [a, b, c])*). Seguidamente escreve o X e muda de linha. Depois falha (por causa do *fail*). Quando falha, retrocede para o *nl*. Mas o *nl* não tem soluções alternativas (só há uma maneira de mudar de linha). Portanto, retrocede para o *write(X)*. Mais uma vez, não há soluções alternativas, pois só se pode escrever algo de uma maneira. Então retrocede para *member(X, [a, b, c])*.

O objetivo *member(X, [a, b, c])* tem outra solução (X=b). Com esta solução, o Prolog vai agora avançar e tentar não falhar desta vez. Escreve o b e muda de linha. Como a cláusula da definição tem o *fail*, ele volta a falhar, volta a retroceder até chegar de novo ao *member(X, [a, b, c])*, o qual tem mais uma solução (X=c).

O Prolog volta a imprimir o X (X=c), volta a mudar de linha, volta a falhar, volta a retroceder até chagar de novo a member(X, [a, b, c]).

Desta vez, *member(X, [a, b, c])* já não tem soluções alternativas. Portanto, como não pode retroceder mais, o Prolog falha (apesar de ter feito tudo quanto devia). É essa a razão da mensagem **False** que o interpretador escreve no final da execução.

Para evitar esta falha, que não deve ocorrer, acrescenta-se uma nova cláusula do procedimento *p*, a qual serve apenas para não falhar:

```
p :-
     member(X, [a, b, c]),
     write(X), nl,
     fail.
p.
```

Agora, após esgotar as alternativas da cláusula 1, o Prolog passa para a cláusula 2. Esta não origina nenhuma espécie de computação, mas tem sucesso.

Mas há mais coisas a saber sobre este assunto, nomeadamente a interação do *fail* com o *cut*, e a necessidade de usar o *repeat* em certos programas.

A pergunta que vos deixo é: será que a seguinte solução funciona?

```
p :-
     member(X, [a, b, c]),
     write(X), nl,
     !,
     fail.
p.
```

Porquê?

3 E o *repeat*?

Imagina que queres escrever a palavra Olá repetidas vezes. O *fail* não funciona:

```
?- write(ola), nl fail.
ola
false
?-
```

O Prolog começa por escrever olá, depois muda de linha, depois falha. Quando falha, retrocede para o *nl*, procurando uma solução alternativa. O *nl* não tem soluções alternativas – não se pode mudar de linha de maneiras diferentes. Como o *nl* não tem soluções alternativas, o Prolog retrocede para o passo anterior, o *write(ola)*, o qual também não tem soluções alternativas (não se pode escrever olá de várias maneiras). Não tendo mais para onde retroceder, o *Prolog* falha mesmo (daí a mensagem **false** que o interpretador escreve no fim).

Conclusão: o *fail* não originou uma repetição porque aquilo que pretendemos repetir não tem soluções alternativas. Quando isso acontece, temos de usar o *repeat*, mas ….

O *repeat* não repete coisa nenhuma:

```
?- repeat, write(ola), nl.
ola
true
```

O *repeat* é um predicado que tem sempre soluções alternativas, o que só por si, não origina nenhum repetição.

Para que o *Prolog* possa repetir seja o que for, sem recorrer à recursividade, é necessário conjugar a geração de soluções alternativas com a falha. Quando o *Prolog* falha, retrocede até ao passo anterior àquele em que falhou, e tenta encontrar uma solução alternativa nesse passo que lhe permita não falhar no passo seguinte. Dado que o *repeat* tem sempre soluções alternativas, pode combinar-se o *repeat* com o *fail* para originar uma repetição.

O que aconteceria na seguinte interação?

```
?- repeat, write(ola), nl, fail.
```

O *Prolog* repetiria indefinidamente a escrita do átomo *olá*.

Já arranjámos uma maneira de repetir algo que não tem soluções alternativas (escrever olá), combinando as indefinidas soluções alternativas do *repeat* com a falha do *fail*. Depois de escrever *olá* e mudar de linha, o *Prolog* falha devido ao *fail*. Retrocede até ao *nl*, mas não encontra soluções alternativas. Volta a retroceder e volta a não encontrar soluções alternativas. Finalmente, retrocede até ao *repeat*, o qual tem sempre mais uma solução. Com essa nova solução, o *Prolog* avança procurando não falhar: escreve *olá*, muda de linha, e falha. Volta a retroceder até ao *repeat*, volta a avançar, volta a escrever *olá*, volta a falhar, volta a retroceder… e assim indefinidamente.

Isto é quase o que pretendemos: repetir um processo que originalmente não tinha soluções alternativas. Mas queremos que esta repetição termine; não queremos que ela prossiga indefinidamente.

Como conseguir parar a repetição? Como o *fail* falha sempre, e como é a falha que obriga o *Prolog* a retroceder até encontrar o *repeat*, em vez do *fail*, temos de usar algo que falhe enquanto pretendermos repetir, e que não falhe, quando já não pretendermos repetir. Supondo que o predicado *cond* falha em certas condições e tem sucesso nas outras condições, o seguinte código faria o que pretendemos:

```
?- repeat, write(ola), nl, cond.
```

Isto repetiria até que a condição *cond* fosse verdade. Se *cond* falhar, tudo se passa como quando usámos o *fail*. Porém, assim que *cond* tiver sucesso, o *Prolog* não falha (não tendo por isso que retroceder em busca de soluções alternativas que lhe permitissem não falhar). O predicado *cond* não existe. Em cada caso, temos de implementar algo que funcione dessa maneira. Esta é a base de todos os programas que repetem um processo, combinando o *repeat* com a falha.

O seguinte programa gera um inteiro entre 1 e 5 e depois implementa um ciclo em que pede ao utilizador para adivinhar o número gerado. Se o utilizador adivinhar, o ciclo termina. Se o utilizador não adivinhar, o programa volta a pedir-lhe a sua nova tentativa. Isto repete-se até que o utilizador acerta.

```
guess_number :-
      N is random(5) + 1,
      repeat,
         acquire_guess(G),
         process_guess(G, N).

acquire_guess(G):-
      write('Input your guess (from 1 to 5): '),
      read(G).
```

Na execução do procedimento *guess_number/0*, o *Prolog* começa por instanciar a variável *N* com um inteiro aleatório entre 1 e 5. Seguidamente, o *Prolog* passa pelo *repeat*, o qual se limita a ter sucesso. Depois, pede ao utilizador para inserir uma tentativa e lê o número introduzido pelo utilizador. Finalmente, o *Prolog* terá de processar a tentativa do utilizador (G). É o procedimento *process_guess/2* que se encarrega de efetuar esse processamento.

O que pretendemos então que o procedimento *process_guess/2* faça? Se o G for diferente do N, queremos repetir a interação com o utilizador. Para isso, se G ≠ N, *process_guess/2* tem de falhar, obrigando o *Prolog* a retroceder. Se a tentativa do utilizador estiver correta, nós pretendemos que o *Prolog* termine a execução do programa. Para isso, se G=N, *process_guess/2* não deve falhar, de modo que o programa termine.

```
process_guess(N, N):-
      write('ACERTASTE'), nl.
process_guess(I, N):-
      I \= N,
      write('Não, não!'), nl,
      fail.
```

No programa *guess_number/0*, as instruções desde o *repeat* até ao *process_guess/2* seguem o modelo de todos os processos de repetição por falha que recorrem ao *repeat* para gerar soluções alternativas (onde elas não existiam originalmente), e à falha que obriga o *Prolog* a retroceder até ao *repeat*.

Portanto, as lições importantes são:

Se um processo computacional não tem soluções alternativas, o *fail* não obriga o *Prolog* a repetir nada. O *Prolog* retrocede em busca de alternativas que lhe permitam não falhar, só que não as encontra e, por isso, termina com falha.

Portanto, para que o *Prolog* realmente efetue um processo repetitivo, é necessário introduzir um predicado que gere soluções alternativas. É esse o papel do *repeat*. Mas o *repeat* tem sempre soluções alternativas, por isso, a utilização do *fail* com o *repeat* dá origem a uma repetição indefinida.

Para evitar a repetição indefinida, em vez do *fail*, tem de se recorrer a um processo computacional que falha em certas circunstâncias mas não falha noutras.

Todos os programas que implementam uma repetição por falha, recorrendo ao *repeat* têm a mesma estrutura que as instruções compreendidas entre o *repeat* e o *process_guess/2*, inclusive.

4 É fácil cometer erros quando se usam *cuts*

Atenta na definição do predicado *maior/3* que serve pare determinar o maior de dois números:

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

Esta definição funciona perfeitamente para determinar o maior de dois números:

```
?- maior(2, 8, X).
X = 8
?- maior(10, 5, X).
X = 10
```

O predicado funciona bem se o terceiro argumento for uma variável não instanciada. Se for uma constante, o predicado poderá comportar-se erradamente:

```
?- maior(7, 2, 7). % Funciona bem
True
?- maior(7, 2, 1). % Funciona bem
False
?- maior(7, 2, 2). % ERRADO. Funciona mal
True
```

Voltemos à definição para perceber a causa do mau funcionamento:

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

Na chamada `maior(7, 2, 2)`, a primeira cláusula não pode ser usada porque a cabeça da cláusula, maior(X, Y, X), não emparelha com o objetivo, maior(7, 2, 2). Portanto, a execução não chega ao *cut*. Assim, o *Prolog* pode recorrer à segunda cláusula. Como a cabeça da segunda cláusula, maior(_, Y, Y), emparelha com o objetivo, maior(7, 2, 2), o *Prolog* tem sucesso. Mas deveria ter falhado, pois o maior de 7 e 2 é 7.

Como se corrige?

Como o problema surge apenas quando o predicado é chamado com uma constante no terceiro argumento, a solução consiste em chamar o predicado com uma variável (já vimos que funciona bem) e, depois, verifica-se se o valor devolvido emparelha com o argumento usado na chamada. A nova versão do predicado chama-se *maximum/3*:

```
maximum(X, Y, M):-
      maior(X, Y, Z),
      M = Z. % Z é uma variável sem valor
maior(X, Y, X):- X >= Y, !.
maior(_, Y, Y).
```

Outra solução

Na literatura é frequente encontrar solução diferente:

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

Mas então o *cut* não deveria servir para evitar computações repetidas? A segunda cláusula volta a comparar as variáveis X e Y, apesar de a primeira cláusula já o ter feito. Então que vantagens tem este *cut*?

Em termos de eficiência, apenas se evita que a segunda cláusula seja usada se a primeira tiver tido sucesso. Se X for maior que Y, a execução do programa passa pelo *cut*, comprometendo-se

com a primeira cláusula. No entanto, se X for menor que Y, a execução passa para a segunda cláusula voltando a comparar X com Y. Os ganhos de eficiência são menores

Mas o programa ganha em legibilidade. Esta solução com o *cut* tem exatamente a mesma leitura que a solução sem o *cut*. Ou seja, a introdução deste tipo de *cut* não altera a leitura do programa, em relação à definição puramente declarativa. O *cut* usado sem altear a leitura lógica do programa em relação à definição declarativa pura chama-se *green cut*.

A utilização de *green cuts* consiste num bom compromisso entre a legibilidade do código e a eficiência. Quando se remove um green *cut*, o programa continua a funcionar (só que envolvendo mais computação).

Os *cuts* cuja remoção resultar em programas que deixam de funcionar chamam-se *red cuts*. Há situações em que é necessário usar *red cuts*, mas eles deve ser usados apenas se estritamente necessário.

5 Compilação de Erros Frequentes no *Prolog* Procedimental

As duas secções seguintes analisam erros encontrados em provas de avaliação de alunos de Inteligência Artificial. A análise tem por objetivo chamar a atenção dos alunos para erros a evitar.

5.1 Repetir um processo que não tem soluções alternativas

Nesta secção apresentam-se exemplos de dois erros frequentes em procedimentos que repetem processos computacionais que não têm soluções alternativas, recorrendo à repetição por falha. Como aquilo que se pretende repetir não tem soluções alternativas, é necessário usar o predicado *repeat/0*, o qual gera um número indefinido de soluções alternativas. Em geral, qualquer solução que não use o *repeat/0* ou que não tenha uma falha está totalmente errada.

O procedimento que serve de enquadramento aos erros que se exemplificam serve para imprimir, no monitor do computador, os números de contribuinte de pessoas cujo nome é inserido, pelo utilizador, através do teclado do computador. O ciclo implementado pelo programa termina quando o utilizador introduzir o átomo *fim*.

Exemplo de interação com o programa, e que o utilizador introduz sucessivamente nomes de pessoas e o programa imprime os seus NIFs, até que o utilizador introduz o átomo *fim*:

```
?- nifs.
Introduz um nome: mariana.
123456789
Introduz um nome: catarina.
345678912
Introduz um nome: fim.
True.
```

Na implementação do procedimento *nifs/0*, partimos do princípio que existe uma base de dados com os NIFs de todas as pessoas, como nos seguintes exemplos:

```
nif(mariana, 123456789).
nif(ze, 234567891).
nif(catarina, 345678912).
```

O predicado de leitura dos nomes das pessoas e o corpo principal do procedimento *nifs/0* são os seguintes:

```
ler_pessoa(Nome):-
  write('Introduz um nome: '),
  read(Nome).

nifs :-
  repeat,
    ler_pessoa(Nome),
    processar_pessoa(Nome), !.
```

Para que o procedimento *nifs/0* repita o processo de leitura de um nome e impressão do NIF correspondente, o procedimento *processar_pessoa/1* tem de falhar, obrigando o Prolog a retroceder, em busca de soluções alternativas que lhe permitam evitar a falha. Como a leitura de nomes não tem soluções alternativas, e como cada pessoa apenas tem um NIF, o Prolog retrocede até encontrar o *repeat/0*, o qual tem sempre (mais) soluções alternativas. Quando o utilizador introduzir o átomo fim, em vez de um nome, o procedimento *processar_pessoa/1* não pode falhar, para que a repetição termine. A definição correta de *processar_pessoa/1* é a seguinte:

```
processar_pessoa(fim):- !.
processar_pessoa(Nome):-
     nif(Nome, NIF),
     write(NIF), nl,
     fail.
```

O cut na primeira cláusula serve para evitar testar, na segunda cláusula, se o nome é diferente de *fim*. Com o cut, temos a certeza que a execução chega à segunda cláusula apenas se o nome não for o átomo *fim*.

ERRO

```
processar_pessoa(fim):- !.
processar_pessoa(Nome):-
   write(nif(Nome, Nif)),
   nl,
   fail.
```

Este é um erro muito grave que consiste em acreditar que `nif(Nome, Nif)` se comporta como uma função que seria avaliada e devolveria o NIF da pessoa especificada. Este erro revela três conceções absolutamente erradas:

1. O predicado *nif/2* não é nenhuma função e, por isso, não devolve nada.
2. Se o *Prolog* possibilitasse a definição de funções, a função *nif* teria apenas um argumento e não dois. Receberia o nome de uma pessoa e retornaria o seu NIF (return).
3. Mesmo que fosse uma função (com um argumento), o Prolog não avalia expressões funcionais passadas como argumento, por isso, o procedimento continuaria errado

Qualquer destes erros seria suficiente para que a classificação fosse zero. Mas os três juntos deveriam chumbar a pessoa para sempre.

ERRO

```
processar_pessoa(Nome):-
   nif(Nome, Nif),
   write(Nif), nl,
   fail.
processar_pessoa(_).
```

Esta resolução tem um erro mais subtil, mas também muito grave. Quando o procedimento *processar_pessoa/1* recebe um nome (diferente de *fim*), ele deve obter o NIF da pessoa (através do predicado *nif/2*), deve imprimir o NIF obtido, e deve falhar.

De facto, a primeira cláusula obtém corretamente o NIF da pessoa recebida, imprime corretamente o NIF, e falha. Tudo parece bem, mas não está. Ao falhar, o Prolog retrocede em busca de uma solução que não falhe. As ações *nl/0* e *write/1* não têm soluções alternativas, e o predicado nif/2 também não (cada pessoa tem apenas um NIF). Isto significa que esta primeira cláusula se esgota (é incapaz de fornecer uma solução para o problema). Assim sendo, o Prolog passa para a segunda cláusula, a qual não falha. Consequentemente, como o procedimento *processar_pessoa/1* não falha, o procedimento *nifs/0* não repete. Mas nós pretendíamos que ele repetisse.

5.2 Repetir um processo que tem soluções alternativas

Nesta secção apresenta-se exemplo de um erro num procedimento que repete um processo computacional enquanto este tiver soluções alternativas, recorrendo à repetição por falha. Como aquilo que se pretende repetir tem soluções alternativas, não se pode usar o predicado *repeat/0*. Em geral, qualquer solução que use o *repeat/0* ou que não tenha uma falha está totalmente errada.

O procedimento que serve de enquadramento ao erro exemplificado serve para imprimir, no monitor do computador, os identificadores e os preços dos produtos de uma dada classe existentes numa loja. O ciclo implementado pelo programa termina quando não existirem na loja mais produtos da classe especificada.

Na implementação do programa, parte-se do princípio que cada produto, a classe a que pertence e o seu preço é mantido numa base de dados constituída por factos do predicado product/3, como os seguintes exemplos:

```
product(cup1, cup, 10).          product(cup2, cup, 5).
product(p1, plate, 7).           product(g1, glass, 3).
```

Exemplo de funcionamento do procedimento pretendido:

```
?- print_products(cup).
cup1: 10
cup2: 5
true
```

O procedimento *print_products/1* é um ciclo que repete a consulta de um produto da classe especificada (*cup*, no exemplo), a impressão do identificador e do preço do produto obtido da base de dados, enquanto houver produtos da classe especificada.

A implementação correta do procedimento *print_products/1* é a seguinte:

```
print_products(Class):-
      product(Prod, Class, Price),
      write(Prod), write(': '), write(Price), nl,
      fail.
print_products(_).
```

A primeira cláusula da definição de *print_products/1* efetua todo o trabalho mas falha quando já não há produtos da classe especificada. Nessa altura, a execução do procedimento passa para a segunda cláusula e o programa termina com sucesso. Resoluções que não tenham a segunda cláusula estão erradas porque o predicado, ainda que fizesse o que se pretende, falharia. Resoluções que não tenham a falha no final da primeira cláusula estão erradas porque não haveria repetição.

Apresenta-se seguidamente um erro grave e uma sua variação igualmente grave.

ERRO

```
print_produto(Class):-
   assert(produto(Produto, Class, Preco)),
   write(Produto:Preco),
   nl,
   fail.
print_produto(_).
```

A definição da primeira cláusula segue um padrão semelhante ao padrão correto mas apresenta um erro absurdo que reflete mais do que uma concessão errada:
`assert(produto(Produto, Class, Preco)).`

1. A ação *assert/1* serve para criar uma cláusula nova (um facto, neste caso); não serve para efetuar uma consulta
2. Não há lógica nenhuma na criação do facto especificado na ação *assert/1* pois o identificador do produto não é conhecido e o seu preço também não.
3. Finalmente, o procedimento não tem nada que criar factos; tem de os consultar.

Uma variação muito mais frequente deste erro, mas muito mais aceitável, é a utilização do *retract/1*, em vez do *assert/1*:

```
print_produto(Class):-
   retract(produto(Produto, Class, Preco)),
   write(Produto:Preco),
   nl,
   fail.
print_produto(_).
```

Este programa funcionaria, mas apagaria a base de dados com os produtos. Portanto, seria um erro também muito grave. De facto a ação *retract/1* instancia as variáveis do seu argumento com os valores encontrados no facto removido, por isso, o programa consegue imprimir o identificador do produto e o seu preço, de acordo com o desejado. Mas a ação *retract/1* remove os factos que emparelham com o seu argumento, o que é totalmente inaceitável.