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* Declarativo

NOTA: Não pretendemos, de forma alguma, que estas notas sejam suficientes nem sequer que abranjam toda a matéria de Prolog Declarativo Avançado.

Nesta altura, as notas limitam-se a uma análise relativamente detalhada de erros frequentes encontrados nas provas de avaliação de alunos de Inteligência Artificial, em exercícios de Prolog Declarativo.

O livro "Programming in Prolog", de Clocksin & Mellish, é a fonte de consulta aconselhada para toda a linguagem *Prolog*.

Índice

Compilação de Erros Frequentes no *Prolog* Declarativo

Esta compilação procura evidenciar erros frequentes de *Prolog* Declarativo e explicar as razões pelas quais são erros graves, se forem feitos por alunos que deveriam saber resolver exercícios simples de *Prolog*. O documento começa com a apresentação de erros muito gerais. Seguidamente mostram-se outros erros em dois exercícios concretos mas que surgem com infeliz frequência noutros exercícios.

1 Erros gerais

Em geral, a utilização do *fail* ou do *repeat*, num predicado que usa a recursividade para construir o seu resultado, está errada.

A segunda classe geral de erros é a que corresponde ao seguinte padrão em que é suposto que o predicado definido (*p/2*, mas poderia ser outro qualquer) deve devolver, no seu segundo argumento, uma lista que vai construindo recursivamente a partir de uma transformação de cada elemento de outra lista.

```
p([X|L1],  L2):- transformacao(X, Y), p(L1, [Y|L2]).
p([],  []).
```

Este padrão está errado e corresponde a um erro grave. Se a lista que se obtém de [X|L1] é L2, é impossível que o resultado que se obtém de L1 seja [Y|L2], a qual tem mais um elemento do que a lista resultado, L2.

O padrão correto seria o seguinte:

```
p([X|L1],  [Y|L2]):- transformacao(X, Y), p(L1, L2).
p([], []).
```

Neste padrão, a lista de saída correspondente a [X|L1] é [Y|L2] e a lista correspondente a L1 (com menos um elemento do que [X|L1]) é L (também com menos um elemento que [Y|L2]).

Finalmente, a tentativa de usar predicados, como se fossem funções, é um erro de lesa-majestade, por exemplo

```
X is list_count(L)
```

Se *list_count* é um predicado, a sua utilização correta será

```
list_count(L, X)
```

que significa X é o número de elementos de L.

Estes erros são suficientemente graves para que a classificação seja ZERO.

2 Lista dos elementos simétricos da lista original

Os exemplos de erros que se seguem são apresentados com base no exemplo da definição declarativa do predicado *symmetric_list/2* que relaciona uma lista com a lista de elementos simétricos da lista original. O predicado *symmetric/2* relaciona um elemento com o seu simétrico.

Exemplos:

```
?- symmetric(a, S).
S = z
?- symmetric(b, S).
S = y
?- symmetric(c, S).
S = x
?- symmetric_list([a, b, c], L).
L = [z, y, x]
```

A definição do predicado deve, em programação declarativa, seguir tão perto quanto possível, a sua definição rigorosa em língua natural:

A lista dos elementos simétricos duma lista original é uma lista cuja cabeça é o simétrico da cabeça da lista original e cuja cauda é a lista dos elementos simétricos da cauda da lista original.

A lista dos elementos simétricos duma lista vazia é uma lista vazia.

```
symmetric_list([X|L1], [S|L2]):-
    symmetric(X, S),
    symmetric_list(L1, L2).
symmetric_list([], []).
```

Antes de analisar erros concretos, convém referir que a existência de dois ou mais erros, na definição de um predicado tão simples como este, é motivo suficiente para uma classificação de zero, por mais pequenos que sejam os erros.

ERRO

```
symmetric_list([], L).
symmetric_list([X|L1], L):-
    symmetric(X, Y),
    symmetric_list(L1, L).
```

A primeira cláusula está errada porque diz que qualquer lista (L) é a lista dos elementos simétricos da lista vazia, o que não é verdade. A lista dos elementos simétricos da lista vazia é uma lista vazia.

Mas os dois aspetos do erro da segunda cláusula seriam, só por si, suficientes para que esta resolução tivesse zero. O primeiro aspeto do erro consiste em dizer que a lista dos elementos simétricos de [X|L1] é L, se a lista dos elementos simétricos de L1 for também L. Como é possível a mesma lista ser a lista dos elementos simétricos de uma lista e simultaneamente da sua cauda? O segundo aspeto do erro é que, apesar de ter sido determinado o simétrico (Y) da cabeça da lista original (X), esse simétrico não foi usado para nada.

ERRO

```
symmetric_list([X|L1], L):-
    symmetric_list(L1, Laux),
    symmetric(X, Y),
    append(Laux, [Y], L).
symmetric_list([], []).
```

Esta resolução tem apenas um erro: os elementos da lista de saída surgem por ordem inversa daquela que deveriam surgir. Não seria um erro muito grave noutras circunstâncias, mas é um erro que resulta da falsa necessidade sentida pelo programador de usar o *append/3* para a criação da lista de saída. O programador esquece-se que pode arrumar o resultado da transformação da cabeça da lista de entrada na cabeça da lista de saída. Ou então, o que é pior, julga que esse processo dá origem a uma lista cujos elementos ficam arrumados na ordem errada. Então, na tentativa de obter a ordem desejada, usa o *append/3*.

ERRO

```
symmetric_list([X|L1], [Y|L]):-
    symmetric(X, Z),
    Y is Z,
    symmetric_list(L1, L)
symmetric_list([], []).
```

Esta definição tem um erro ligeiro. Infelizmente, esse erro resulta de uma falta de assimilação do estilo da programação em Prolog. Por que razão se usa a variável Y para construir a lista se a variável Z tem exatamente aquilo que é necessário? Esta falta de a vontade com a linguagem origina o erro. `Y is Z`, para além de ser desnecessário, apenas resulta se Z for um número. Se Z for um valor não numérico, `Y is Z` origina uma exceção e a execução do programa aborta.

3 Número de elementos de uma lista

Os exemplos de erros que se seguem são apresentados com base no exemplo da definição declarativa do predicado *list_count/2* que determina o número de elementos de uma lista.

O número de elementos de uma lista com cabeça e cauda é igual ao número de elementos da cauda mais um. O número de elementos de uma lista vazia é zero.

```
list_count([_|L], N):-
     list_count(L, N1),
     N is N1 + 1.
list_count([], 0).
```

Um número significativo de alunos vê esta definição e apresenta diversas perguntas que não fazem sentido relativamente à programação, independentemente da linguagem usada:

- Em que problemas é que a chamada recursiva se faz primeiro? Em que problemas é que a chamada recursiva se faz no fim?
- Depois da chamada recursiva, a execução do programa ainda chega à conta `N is N1+1`?

Ainda bem que fazem estas perguntas, mas cabe-me mostrar que Por que razão a primeira pergunta é absurda? É absurdo porque, na programação, como em muitas outras atividades de resolução de problemas, é raro haver receitas. É preciso pensar sobre o problema e sobre a forma de o resolver. A grande vantagem de usar programação declarativa é que podemos pensar apenas nas definições de conceitos, nas condições em que determinadas afirmações são verdadeiras. Quando se usa programação declarativa não se deve pensar na computação que está a ser feita, não se deve pensar num procedimento que pudesse produzir o resultado.

A segunda pergunta também seria escusada, no domínio da programação. Se o programa não tiver instruções que o impeçam, a sua execução pode passar por todas as linhas do seu código. Na primeira

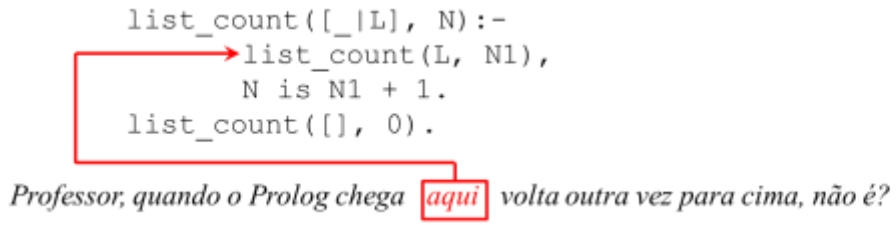
cláusula da definição de *list_count/2*, existem duas linhas, uma a seguir à outra. Para todos os efeitos, é uma definição com o formato

```
P :-
      Q1,
      Q2,
      …,
      Qn.
```

O facto de, na primeira cláusula de *list_count/2*, a primeira linha da condição ser uma chamada recursiva não altera em nada o facto de se tratar de uma definição com o padrão idêntico à da definição de P.

A execução não passa numa determinada linha da definição de uma cláusula, apenas se falhar numa linha anterior. Em qualquer outro caso, a execução do programa passa por todas as linhas da definição da cláusula. Depois de Q_1, a execução passa por Q_2; depois de Q_2, passa por Q_3, etc, etc, até chegar a Q_n. A execução só não passa por Q_i, se falhar em Q_j, com j < i, ou se nem chegar a entrar na cláusula em causa.

A razão profunda desta pergunta (a execução passa em `N is N1+1`?) reflete uma confusão enorme cuja raiz está talvez na utilização incorreta da língua portuguesa, e numa atitude incorreta também face à programação declarativa. Deparo frequentemente com a seguinte descrição em português da execução da primeira cláusula da definição de *list_count/2* e de todas as definições com recursividade:



Sem dúvida que muitos alunos percebem o que estão a dizer, mas estou convencido que esta descrição em português reflete conceções erradas e conduzirá muitos dos que a ouvem a pensar de forma errada. O problema é a expressão *"Volta para cima"*. Não volta nada para cima, nem volta para baixo, nem para os lados. Aquilo que acontece é o que aconteceria em qualquer circunstância em que a definição de um conceito recorre à definição de outro conceito, isto é, a execução da cláusula recorre à definição do predicado que é usado. Neste caso, o predicado usado é o mesmo predicado cuja definição está a ser executada, mas isso não deve ser considerado especial.

Reparem nas seguintes duas cláusulas da definição de *concomitante*, encontrada no dicionário Priberam:

1. Que acompanha ou coexiste. = COEXISTENTE
2. Que acontece ou se faz ao mesmo tempo. = SIMULTÂNEO

Ninguém diz, da primeira cláusula, que o significado de *concomitante* volta para cima (no dicionário, coexistente ocorre antes – acima – de concomitante). Nem ninguém diz, da segunda cláusula, que o significado de *concomitante* vai para baixo. Ninguém diz porque, embora seja verdade, isso não tem qualquer espécie de relevância. A única coisa que é relevante é que, na primeira cláusula, o significado de *concomitante* se define à custa do significado de *coexistente*; e, na segunda cláusula, à custa do significado de *simultâneo*. O mesmo se passa exatamente, sem tirar nem por, com a definição de um predicado à custa de outro (e, em particular, à custa de si mesmo). Quando a execução da definição de um predicado encontra outro, passa a executar a definição desse outro e, quando termina essa (sub) execução, continua no sítio onde estava.

Por que razão é tão mau dizer *"Volta para cima"*? É mau porque se confunde com retrocesso (*backtracking*), o que produz as mais estranhas conceções que se possa imaginar.

Em termos de programação em *Prolog*, *"Voltar para cima"* quer dizer retroceder, o que é muitíssimo diferente de chamar um predicado, incluindo o próprio predicado que faz a chamada. Chamar o predicado Q_i significa "executar a definição de Q_i e ver se tem sucesso". Retroceder até Q_i significa "verificar se há mais alguma solução alternativa para Q_i e, se houver, voltar a executar as instruções que se seguem a Q_i (abaixo de Q_i), com a nova solução de Q_i, na tentativa de encontrar um desfecho com sucesso".

Em termos de atitude face à programação declarativa, pensar *"Volta para cima"* reflete a atitude errada, porque procura recorrer à computação, ao procedimento computacional que produz um resultado. A vantagem da programação declarativa é que permite evitar ter de perceber os detalhes do processo computacional. Se alguém pretende embrenhar-se nesses detalhes, enquanto aprende o básico da programação declarativa, está a desperdiçar aquela que é considerada uma grande vantagem.

Para perceber a definição de *list_count/2*

```
list_count([_|L], N):-
      list_count(L, N1),
      N is N1 + 1.
list_count([], 0).
```

basta perceber o seguinte: O número de elementos (N) de uma lista com cabeça e cauda é igual ao número de elementos (N1) da cauda mais um. O número de elementos de uma lista vazia é zero.

Antes de analisar erros concretos, convém referir que a existência de dois ou mais erros, na definição de um predicado tão simples como este, é motivo suficiente para uma classificação de zero, por mais pequenos que sejam os erros.

Agora os erros concretos mais comuns.

ERRO

```
list_count([], _).
list_count([X|L1], N):- N1 is N+1, list_count(L1, N1).
```

A primeira cláusula está errada porque significa "O número de elementos de uma lista vazia é qualquer coisa", o que é um autêntico disparate.

A segunda cláusula está errada por várias razões. Primeiro, não se pode calcular N1 porque não é conhecido. Segundo, não se pretende calcular N1 antes da chamada recursiva. N1 é o número de elementos da cauda da lista, sendo o seu valor calculado na chamada recursiva `list_count(L1, N1)` e não antes dela. Finalmente, ela significaria que o número de elementos da cauda da lista é igual ao número de elementos da lista mais um, o que é absurdo.

ERRO

```
list_count([], _).
list_count(L, N):- list_count(L1, N1), N is N1+1.
```

Esta definição é apenas um disparate total; nem dá gozo desmontá-la. A primeira cláusula diz que o número de elementos de uma lista é qualquer coisa, não importa o quê. Na segunda cláusula diz-se que o número de elementos da lista L é igual ao número de elementos de outra lista qualquer (L1) mais um. Repara que não há nenhuma relação entre L1 e L.

ERRO

```
list_count([], []).
list_count([X|L], N):- list_count(L, N1), N is N+1. % N em vez de N1
```

A primeira cláusula diz que o número de elementos de uma lista vazia é uma lista vazia. Então uma lista vazia é um número?

A segunda cláusula tens dois erros, um deles concetual, e o outro que viola as regras do Prolog. Começando pela violação das regras do Prolog: `N is N+1` falha sempre porque não existe nenhuma quantidade (N) que seja igual a si (N) mesma mais um. De onde vem este erro? Vem das outras linguagens de programação. Numa linguagem de programação convencional, uma variável é usada como um espaço de memória para guardar informação (neste caso, informação numérica). Nessas linguagens, a expressão `N is N+1` tem significado procedimental apenas. Significa substituir a grandeza armazenada no espaço de memória da variável N pela grandeza que lá estava armazenada mais um.

Em Prolog isto não faz sentido porque uma variável é uma entidade lógica. Se tem um valor, não tem outro.

ERRO

```
list_count([], 0).
list_count([X|L1], N):- N > 0, list_count(L1, N1), N is N1+1.
```

Esta segunda cláusula tem um erro, fruto de uma incompreensão da definição do conceito. O erro consiste em testar o valor de N antes de a variável ter valor. A incompreensão tem que ver com a futilidade de se testar, à entrada, o valor de uma variável que pretendemos que seja o programa a calcular. Este teste poderia ser feito, mas apenas nos casos em que fosse dado ao programa um valor já instanciado de N. Caso contrário, o teste `N > 0`, em que N não está instanciado, gera uma exceção e a execução aborta.

ERRO

```
list_count([], 0).
list_count([X], 1).
list_count([X|L], N):- list_count(L, N1), N is N1+1.
```

Este é um erro pequeno, motivo apenas de um pequeno desconto na classificação. A segunda cláusula está contemplada na terceira. De facto uma lista com um só elemento tem também uma cabeça e uma cauda. No entanto, o programa não está errado porque tem uma cláusula inútil. Está errado porque essa cláusula origina soluções repetidas:

```
?- list_count([a], N).
N = 1;
N = 1
```