Programação em Linguagem Prolog

- Particularidades da linguagem, ordem das regras e das interrogações,
- Definição de operadores e aritmética (e runtime errors em Prolog),
- Recursividade,
- Tipos de dados em Prolog,
- Manipulação simbólica.
- Terminação
- Backtracking
- Cut
- Optimização Tail Recursion

Definições

Um programa em Prolog é uma colecção de regras (cláusulas) da forma:

```
A :- B_1, B_2, ... , B_n.
```

A diz-se ser a cabeça da regra e $B_1, B_2, ..., B_n$ o corpo. A virgula no corpo indica conjunção. O ponto final é usado para indicar o fim da definição da regra. Regras que não tem corpo chamam-se *factos* e.g.

```
pai(joao,rui).
```

O conjunto de regras que contém a mesma relação na cabeça definem um predicado. A ordem de apresentação destas regras vai ser determinante para a execução do programa. Um programa "*é executado*" fazendo uma pergunta (query) sobre um determinado predicado. Exemplo ?pai(X, rui).

Uma resposta em Prolog a uma computação pedida é uma de três possibilidades: falhanço (**no**), sucesso (**yes**) ou sucesso com as respectivas substituições para as variáveis presentas na query. No nosso exemplo teríamos: X = joao. Notar que os nomes das variáveis são sempre de inicial maiúscula e.g. X, Netos, Xs, MDcomum etc.

Uma das particularidades da Prolog (e da lógica) é que podemos usar um predicado em múltiplas formas. Isto é, um parâmetro do predicado tanto pode funcionar como parâmetro de entrada de dados como de saída. No nosso exemplo tanto podemos perguntar quem é o pai do rui como quem é filho do joao como quem é pai de quem. Assim tanto podemos instanciar o primeiro parâmetro e esperar a resposta no segundo como vice versa.

Interpretação Procedimental

A definição de um predicado (ou seja as cláusulas que o definem) pode ser visto como a definição de um procedimento numa linguagem imperativa.

Por exemplo, a cláusula que define o predicado A

```
A :- B1, B2, ..., Bn.
```

pode ser vista como o procedimento:

```
Procedure A
    call B1
    call B2
    .
    .
    .
    call Bn
end.
```

Os parâmetros instanciados numa query ?A ao predicado A podem ser vistos como os parâmetros de entrada de dados no procedimento A. Os parâmetros não instanciados (variáveis) podem ser interpretados como os parâmetros de saída de dados do procedimento.

Aritmética

As operações aritméticas básicas tem tipicamente uma forma funcional. Em lógica podemos redefini-las numa forma mais relacional mas artificial. Por exemplo, os números naturais podiam ser definidos como:

```
natural(0).
natural(suc(X)) :- natural(X).
```

e a soma como:

```
soma(0,X,X) :- natural(X).
soma(suc(X),Y,suc(Z)) :- soma(X,Y,Z).
```

Em Prolog podemos fazer uso de operadores não lógicos pré-definidos (pelo sistema) que implementam os operadores aritméticos e.g. $<$, $+$, $*$, mod etc.

A interrogação

$$?2 < 3.$$

tem como resposta

yes.

Esta faceta mostra uma componente não lógica do Prolog que é o seu processador de expressões aritméticas.

Avaliador de expressões aritméticas

O Prolog (interpretador) possui um processador de expressões deste tipo que é invocado com queries da forma:

Variavel is **expressao**

O predicado is denomina o avaliador aritmético. A query pode ser lida como "Variavel *toma o valor do resultado de* expressao". Ou seja expressao é processada como uma expressão aritmética e o seu resultado é unificado com Variavel. A query sucede ou não de acordo com o resultado desta unificação.

O operador = denomina unificação explicita. Por exemplo a query

$$?\mathsf{Y} = \mathsf{3}, \mathsf{X}\ \mathsf{is}\ \mathsf{5} + \mathsf{Y}.$$

tem com resultado

$$\mathsf{X} = \mathsf{8}, \mathsf{Y} = \mathsf{3}.$$

No entanto podemos ter situações de erro se a expressão a calcular não for totalmente instanciada. Por exemplo

$$?\mathsf{Y} = \mathsf{X} + \mathsf{3}, \mathsf{Z}\ \mathsf{is}\ \mathsf{Y} + \mathsf{1}, \mathsf{X} = \mathsf{5}.$$

Neste exemplo a primeira unificação é perfeitamente valida. É possível unificar uma variável com uma expressão. No entanto não é possível calcular uma expressão aritmética com variáveis não instanciadas!

Nesta query o Prolog interrompe a computação e produz uma mensagem de erro. No entanto existem Prologs mais sofisticados que conseguem suspender este tipo de computações até que uma substituição seja produzida para as variáveis não instanciadas na expressão.

Exemplos com Aritmética

```
gcd(I,0,I).
gcd(I,J,G) :- J > 0, R is I mod J, gcd(J,R,G).
```

```
factorial(0,1).
factorial(N,F) :- N1 is N - 1, factorial(N1,F1), F is N * F1.
```

De notar que interrogações do tipo

$$?\mathsf{gcd}(5, \mathsf{X}, 5).$$

ou

$$?\mathsf{factorial}(\mathsf{X}, 2).$$

dão origem a mensagens de erro e consequente paragem da execução da query.

Manipulação simbólica

Unificação associada ao uso de termos compostos fornece ao Prolog um enorme potencial para manipulação simbólica.

Um exemplo clássico é a manipulação de expressões matemáticas, como é o caso da diferenciação. Podemos escrever as regras de diferenciação como cláusulas Prolog. Por exemplo a regra:

$$\frac{\delta a x}{\delta x} = a$$

pode ser escrita como

```
derivada(A*X,X,A).
```

ou

$$\frac{\delta \frac{1}{f(x)}}{\delta x} = \frac{-\frac{\delta f(x)}{\delta x}}{f(x)*f(x)}$$

pode ser

```
derivada(1/F,X,-DF/(F*F)) :-
                derivada(F,X,DF).
```

assim como

$$\frac{\delta f(x)*g(x)}{\delta x} = f(x)\frac{\delta g(x)}{\delta x} + \frac{\delta f(x)}{\delta x}g(x)$$

é

```
derivada(F*G,X,F*DG+DF*G)) :-
                derivada(F,X,DF),
                derivada(G,X,DG).
```

O predicado derivada(Expressao, X, ExprDerivada) indica que ExprDerivada é a derivada em ordem a X de Expressao.

O programa com algumas das regras de derivação pode ser definido por:

```
derivada(C,X,0):- number(C).
derivada(X,X,1):- \+ compound(X).
derivada(A*X,X,A).
derivada(X^N,X,N*X^(N-1)).
derivada(sin(X),X,cos(X)).
derivada(cos(X),X,-sin(X)).
derivada(e^X,X,e^X).
derivada(F+G,X,DF+DG) :-
            derivada(F,X,DF), derivada(G,X,DG).
derivada(F-G,X,DF-DG) :-
            derivada(F,X,DF), derivada(G,X,DG).
derivada(F*G,X,F*DG+G*DF) :-
            derivada(F,X,DF), derivada(G,X,DG).
derivada(1/F,X,-DF/(F*F)) :-
            derivada(F,X,DF).
derivada(F/G,X,(G*DF-F*DG)/(G*G)) :-
        derivada(F,X,DF), derivada(G,X,DG).
```

A árvore de prova para a derivação da expressão $\frac{\delta(\frac{1}{x}+(x+4))}{\delta x}$ é

?derivada((1/X)+(X+4),X,W)

{W/DF+DG, F/(1/X), G/(X+4)}

?derivada(1/X,X,DF), derivada(X+4,X,DG)

{DF/(-DF1/(X*X)), F1/X}

?derivada(X,X,DF1), derivada(X+4,X,DG)

{DF1/1}

?derivada(X+4,X,DG)

{DG/(DF2+DG2), F2/X, G2/4}

?derivada(X,X,DF2), derivada(4,X,DG2)

{DF2/1}

?derivada(4,X,DG2)

{DG2/0}

□

W = -1/(X*X) + 1+0

Estruturas de dados em Prolog

Em Prolog não existem estruturas de dados predefinidas como nas linguagens convencionais. No entanto podemos simular as tradicionais estruturas de dados através de composição de termos Prolog chamados *functores*. Tipicamente usa-se estruturas binárias para este fim como é o caso das listas.

Listas

Tradicionalmente uma lista é representada pelo termo .(X, Y) em que X é um elemento e Y o resto da lista. nil denomina a lista vazia. Em Prolog usa-se uma notação mais conveniente: uma lista é o termo [X | Y] em que X é o primeiro elemento e Y é o resto da lista. Por analogia com as linguagens funcionais podemos dizer que X é a *head* da lista e Y é a *tail*. Lista vazia denomina-se pelo termo [ ].

Exemplos (termos equivalentes)

.(a, nil)	[a]
.(a, .(b, nil))	[a, b]
.(a, .(b, X))	[a, b | X]

Podemos definir lista como:

```
list([]).
list([X|Xs]) :- list(Xs).
```

Ao contrário das linguagens funcionais, em Prolog uma lista é um objecto polimórfico. Isto é, podemos ter listas com objectos de diferentes tipos e.g. [a, 1, ana, 5.33, [a, b], X, 4, [1, [2, 3], 4], fim].

Listas e Unificação

Exemplos de manipulação de listas usando unificação.

Uma lista com um só elemento [X].

Notar que a unificação

$$[X \mid Xs] = [1, 2, 3, 4]$$

resulta em

$$X = 1 \text{ e } Xs = [2, 3, 4].$$

Lista com três elementos

$$[X, Y, Z]$$

Uma lista não vazia

$$[X \mid \_]$$

assim [ ] não unifica com [X | _].

O símbolo '_' denomina variável incógnita que existe mas não é usada.

Exemplos:

$$[X, Y \mid Ys] = [1, 2, 3, 4, 5, 6]$$

neste caso temos que

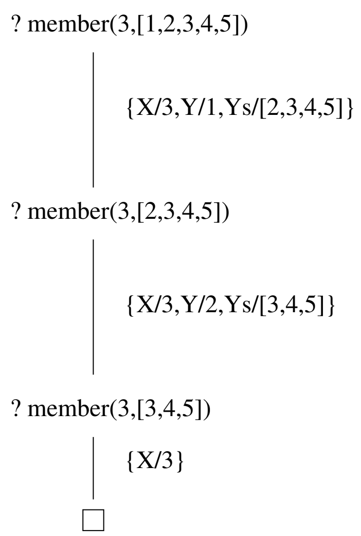
$$X = 1, Y = 2, Ys = [3, 4, 5, 6]$$

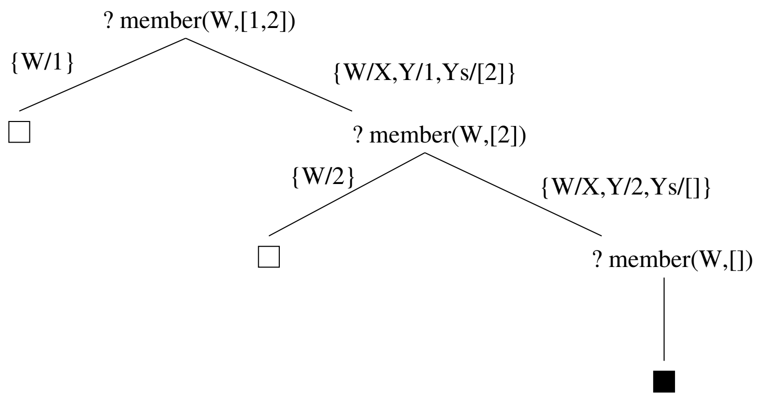
Predicados para manipular listas

Determinar se um elemento é membro de uma lista (member).

```
member(X,[X|_]).
member(X,[Y|Ys]) :- member(X,Ys).
```

Diferentes 'usos' de member.





Outros predicados

Juntar uma lista com outra e obter uma terceira (append).

```
append([],Ys,Ys).
append([X|Xs],Ys,[X|Zs]) :- append(Xs,Ys,Zs).
```

? append([1,2],[4,5,6],W)

{X/1,Xs/[2],Ys/[4,5,6],W/[1|Zs]}

? append([2],[4,5,6],Zs)

{X/2,Xs/[],Ys/[4,5,6],Zs/[2|Zs1]}

? append([],[4,5,6],Zs1)

{Zs1/[4,5,6]}

□

Notar que W = [1 | [2 | [4, 5, 6]]]

ou seja W = [1, 2, 4, 5, 6]

Outros usos de append

Encontrar a diferença entre listas. Exemplo:

$$?\mathsf{append}(\mathsf{Xs}, [3,4], [1,2,3,4]).$$

dá como resposta $\mathsf{Xs} = [1,2]$.

Partir uma lista em duas sub-listas:

$$?\mathsf{append}(\mathsf{Xs}, \mathsf{Ys}, [1,2,3,4]).$$

obtemos varias respostas com por exemplo $\mathsf{Xs} = [1,2], \mathsf{Ys} = [3,4]$.

Definir adjacente que expressa que dois elementos X e Y são adjacentes numa lista Zs.

```
adjacente(X,Y,Zs) :- append(_,[X,Y|_],Zs).
```

Podemos também usar append para definir last i.e. último elemento da lista.

```
last(X,Xs) :- append(_,[X],Xs).
```

Outros predicados

```
reverse([],[]).
reverse([X|Xs],Zs) :- reverse(Xs,Ys), append(Ys,[X],Zs).
```

De notar que este versão de reverse é bastante ineficiente. O número de usos de append é igual ao número de elementos da lista a inverter.

Contar o número de elementos de uma lista.

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

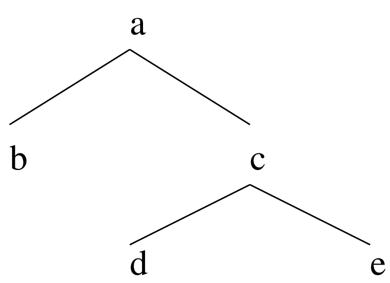
Eliminar todas as ocorrências de um elemento numa lista

```
delete([X|Xs],X,Ys) :- delete(Xs,X,Ys).
delete([X|Xs],Z,[X|Ys]) :- X =\= Z, delete(Xs,Z,Ys).
delete([],_,[]).
```

Árvores Binárias

São representadas pelo functor tree(Elemento, Esq, Dir).

A árvore:



é representada pelo termo

tree(a, tree(b, void, void), tree(c, tree(d, void, void), tree(e, void, void)))

O símbolo void representa folha da árvore.

Definição de árvore binária.

```
bintree(void).
bintree(tree(Elem,Left,Right)):-
        bintree(Left),
        bintree(Right).
```

Árvores (cont)

Verificar se um elemento X pertence a uma árvore:

```
treemember(X,tree(X,_,_)).
treemember(X,tree(Y,Left,Right)):-
            treemember(X,Left).
treemember(X,tree(Y,Left,Right)):-
            treemember(X,Right).
```

Travessias de árvores binárias.

Construir uma lista que corresponda a travessia *pré-ordem* de uma árvore binária.

```
preordem(void,[]).
preordem(tree(X,Left,Right),Xs):-
            preordem(Left,Ls),
            preordem(Right,Rs),
            append([X|Ls],Rs,Xs).
```

Backtracking

A estratégia de pesquisa nas árvores de prova usado pelo Prolog é composta pela combinação de pesquisa *depth-first* (em profundidade) com *backtracking*.

Backtracking é simplesmente o processo de retroceder na árvore de prova até ao último nó onde ocorreu ramificação (i.e. várias opções para resolver uma subquery com a cabeça de uma cláusula).

Ou seja, backtracking indica pesquisa de soluções alternativas.

Exemplo

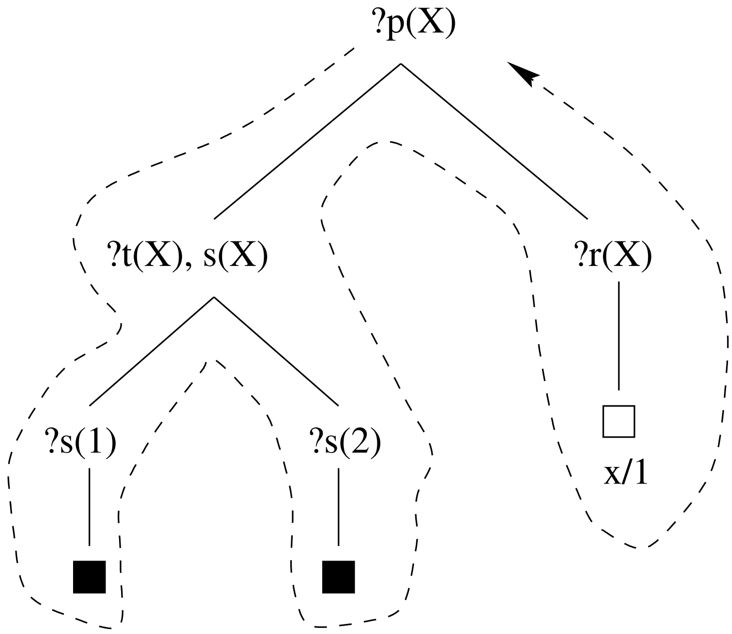
```
p(X) :- t(X), s(X).
p(X) :- r(X).

r(1).
t(1).
t(2).
s(3).
```

Com a query inicial ?`p(X)` backtracking ocorre entre as subqueries ?`t(X)` e ?`s(X)`. Por exemplo, resolvendo ?`t(X)` com a primeira cláusula dá origem a subquery ?`s(1)`. Esta subquery falha e o Prolog retrocede (faz backtracking) para a anterior subquery ?`t(X)`. Resolvendo esta subquery com a segunda cláusula de `t` dá origem a subquery ?`s(2)` que resulta novamente em falhanço. Finalmente a segunda cláusula que define `p` é tentada para resolver a query inicial. Este passo de resolução gera a subquery ?`r(X)` que sucede com `X/1` que é a única solução para a query original.

Backtracking (cont)

Vejamos em termos de árvore de prova o comportamento computacional do Prolog para a mesma query ?`p`(X).




```
p(X) :- t(X), s(X).
p(X) :- r(X).
```

Conclusão: Na nossa programação temos de aprender a “usar” o efeito de backtracking e assim optimizar os nossos programas. Por exemplo tentando provocar que ocorra um falhanço o mais cedo possível.

Uso de *cut*

Como tinha sido estudado anteriormente, o operador *cut* permite cortar computações redundantes da árvore de prova de uma query. Em Prolog este operador tem exactamente o mesmo efeito e funciona como um predicado de sistema que sucede sempre.

O uso do cut funciona como uma optimização pois reduz-se o espaço possível de ramos da árvore de prova a percorrer.

Sumáriamente podemos dizer que em Prolog um cut tem o seguinte comportamento:

a query referente ao cut sucede e faz com que o Prolog se comprometa com todas as escolhas feitas desde que a query inicial unificou com a cabeça da cláusula onde o cut ocorre.

Mas cuidado com a sua utilização !!

Como o cut fornece uma forma de implementar exclusão mutua pode haver a tentação de escrever programas declarativamente errados. Isto porque pela leitura do cut pode haver a tentação de retirar condições do corpo das cláusulas.

O cut é também uma forma de negação.

Cut (cont)

Em termos de leitura de cláusulas o cut tem o seguinte comportamento:

- na cláusula onde ocorre, o cut corta todas as alternativas que existam nos literais à esquerda do cut.
- e corta todas as alternativas que existam em cláusulas abaixo da cláusula onde ocorre o cut.

Usando o programa atrás referido (durante a explicação de backtracking):

```
p(X) :- t(X), !, s(X).
p(X) :- r(X).
```

Aplicando a definição descrita e para a query ?p(X) teríamos que o cut corta:

- todas as alternativas para ?t(X),
- e todas as alternativas para ?p(X) abaixo da primeira cláusula.

Exemplo (junção ordenada de duas listas ordenadas) merge.

```
merge([X|Xs],[Y|Ys],[X|Zs]) :- X < Y, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) :- X = Y, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) :- X > Y, merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs).
merge([],Ys,Ys).
```

Versão com cuts

```
merge([X|Xs],[Y|Ys],[X|Zs]) :- X < Y, !, merge(Xs,[Y|Ys],Zs).
merge([X|Xs],[Y|Ys],[X,Y|Zs]) :- X = Y, !, merge(Xs,Ys,Zs).
merge([X|Xs],[Y|Ys],[Y|Zs]) :- X > Y, !, merge([X|Xs],Ys,Zs).
merge(Xs,[],Xs):- !.
merge([],Ys,Ys):- !.
```

O uso de cuts neste programa força que a definição de merge seja determinística. O cut para além de optimizar em tempo de execução, optimiza o espaço utilizado pois menos informação é necessária reter para fazer backtracking.

Implementar $if - then - else$

```
ifthenelse(P,Q,R) :- P, !, Q.
ifthenelse(P,Q,R) :- R.
```

Notar que seria mais "correcto" adicionar a condição not P ao corpo da segunda cláusula (veremos depois como funciona o operador not).

Optimização Tail Recursion

Principio geral de que programas recursivos podem ser executados como se fossem uma versão iterativa. Assim o espaço de memória necessário para executar o programa é considerávelmente reduzido.

Considere a resolução da query

$$?\mathsf{A1}$$

com a cláusula

```
A2 :- B1, B2, ..., Bn.
```

Neste caso e para o unificador mais geral $\theta = \mathsf{m.g.u.}(\mathsf{A1}, \mathsf{A2})$ podemos aplicar esta optimização ao ultimo literal Bn que corresponde à ultima "call". Podemos re-usar a memória alocada para a query ?A1 para a subquery ?Bn.

A condição fundamental para aplicar esta optimização é que não existam alternativas para resolver a query ?A1. Para além desta condição é necessário que as subqueries $(\mathsf{B1}, \mathsf{B2}, ..., \mathsf{Bn} - 1)\theta$ sejam de resolução determinística.

Uma forma de garantir a existência das precondições é obtida adicionando um cut antes do ultimo literal no corpo da cláusula em questão:

```
A2 :- B1, B2, ...,! ,Bn.
```

Note-se que o cut corta todas as alternativas quer para a query ?A quer para a computação $(\mathsf{B1}, \mathsf{B2}, ..., \mathsf{Bn} - 1)\theta$.

Terminação

O modelo computacional do Prolog induz situações de computações infinitas. O facto da travessia das árvores de prova ser em profundidade (depth-first) pode levar a problemas. Por exemplo se a computação de uma query enveredar por um ramo infinito da árvore então a computação não termina. Isto é, o Prolog não consegue encontrar uma solução para a query inicial apesar de ela existir e estar num ramo finito da árvore de prova.

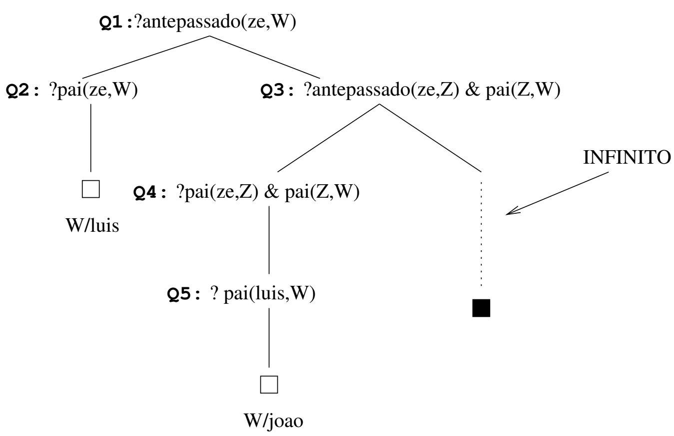
A não terminação aparece com as regras recursivas.

Um exemplo das regras típicas de provocar este efeito são as *left-recursive rules.* Isto é regras em que no corpo o primeiro literal recursivo é o mais a esquerda.

Recordemos a definição de antepassado usada anteriormente:

```
antepassado(X,Y) :- pai(X,Y).
antepassado(X,Y) :- antepassado(X,Z), pai(Z,Y).
pai(ze,luis).
pai(luis,joao).
```

A árvore de prova é:



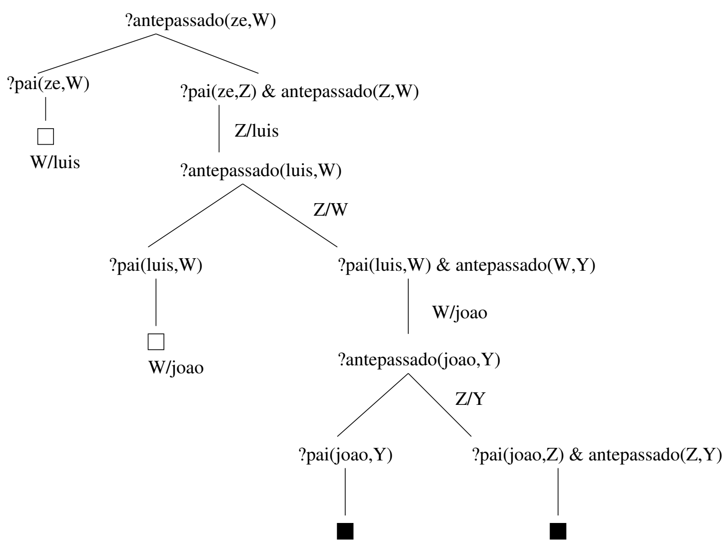
Notar que a query Q3 é uma variante de Q1 o que sugere uma repetição da computação o que explica o ramo infinito.

Como evitar ?

- Reescrever as regras left-recursive em right-recursive,
- Tabulação,
- Bottom-up.

A primeira opção nem sempre é possível de ser obtida. No entanto no nosso exemplo podíamos transformar a segunda regra de antepassado em:

```
antepassado(X,Y) :- pai(X,Z), antepassado(Z,Y).
```



Notar árvore finita!

Tabulação

Notar que nas regras left-recursive a mesma subquery repete-se em vários nós da árvore. Evita-se ramos infinitos utilizando a técnica de memorização ou tabulação. Certos Prologs mais sofisticados incluem estes mecanismos e.g. XSB Prolog.

A ideia base é criar tabelas com subqueries e respectivas respostas. Subqueries são primeiro tentadas a resolver com as respostas guardadas nas tabelas. Existe um teste de inclusão para verificar se uma subquery pode ser respondida com as respostas de outra subquery já calculada.

Bottom-up

Outra alternativa para evitar não terminação. A computação pega nos factos e unifica com o corpo das regras gerando novos "factos". Mas esta computação não é query driven como o Prolog. Isto é, as consequências lógicas derivadas não tem em conta a query inicial a responder!

Outra fonte importante de não terminação são as definições circulares. Por exemplo descrever comutatividade de uma relação.

$$\mathsf{casado(X, Y) :-casado(Y, X).}$$