

PROGRAMAÇÃO EM LÓGICA

PROLOG

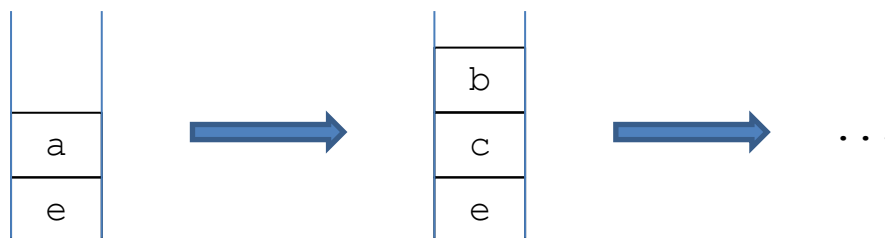
Modelo de Execução

- Implementação do interpretador abstracto numa linguagem de programação concreta
 - 2 decisões para concretizar
 - a *escolha arbitrária* do objectivo da resolvente a reduzir
 - a *escolha não determinística* da cláusula do programa para efectuar a redução
- Prolog
 - Execução sequencial, da esquerda para a direita, dos objectivos da resolvente
 - Pesquisa **sequencial** de uma cláusula unificável e **retrocesso** (*backtracking*)

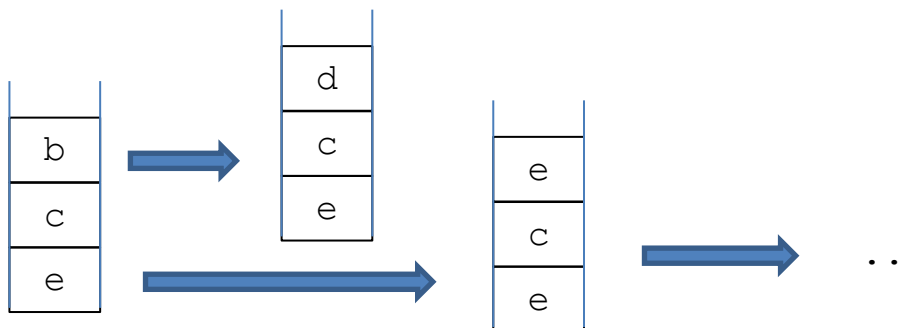
Modelo de Execução (2)

- Resolvente como uma pilha

a :- b, c.
b :- d.
b :- e.
c.
e.
?- a, e.



- Pesquisa sequencial e retrocesso
 - Escolhe a primeira cláusula cuja cabeça unifica com o objectivo
 - Se não houver, a computação é desfeita até à última escolha (**ponto de escolha**), e é escolhida a cláusula unificável seguinte



Computação de um Objectivo

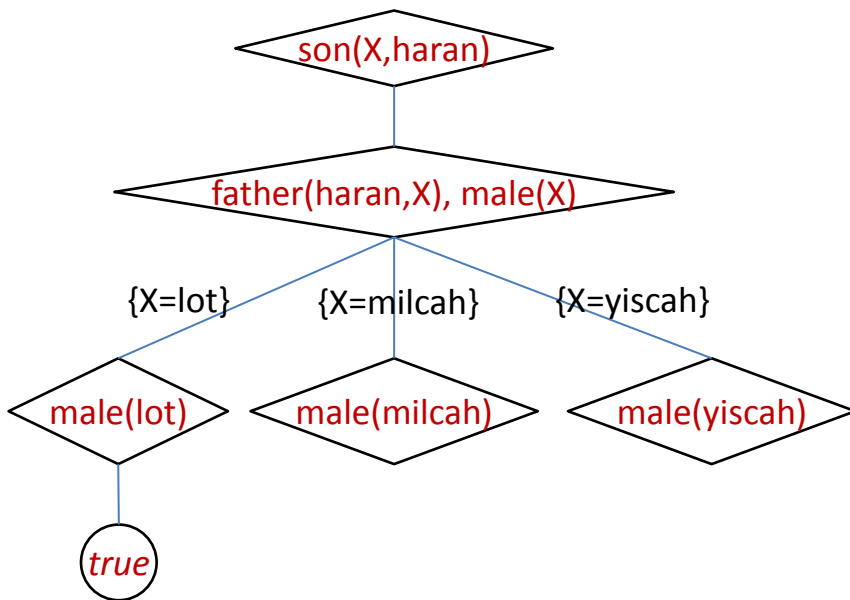
- A computação de um objectivo G em relação a um programa em Prolog P consiste em gerar todas as soluções de G
- Uma computação Prolog de um objectivo G é uma travessia completa em **profundidade primeiro** (*depth-first*) da árvore de pesquisa de G obtida escolhendo sempre o objectivo mais à esquerda
- A maior parte das implementações de Prolog:
 - pesquisa a árvore até encontrar a primeira solução
 - permite ao utilizador indicar que quer mais soluções através do símbolo $;$ (ponto e vírgula)

Alternativa: Paralelismo

- Pesquisa em profundidade não é completa
 - pode não encontrar uma solução (ramo infinito na árvore de pesquisa)
- Pesquisa em largura
 - explorar todas as escolhas possíveis **em paralelo**
 - é completa: encontra sempre uma solução, se existir
 - PARLOG, Concurrent Prolog, GHC, ...
- Paralelismo “ou”
 - percorrer em paralelo todos os ramos da árvore de pesquisa
- Paralelismo “e”
 - executar em paralelo todos os objectivos da resolvente

Traçado

- Escolha determinística pode conduzir a falhanço e retrocesso
 - **f**: falhanço (não há cláusulas cuja cabeça unifique com o objectivo)
 - objectivo a seguir a **f** é onde a computação prossegue ao retroceder
 - “;” indica continuação da computação para procurar mais soluções



```

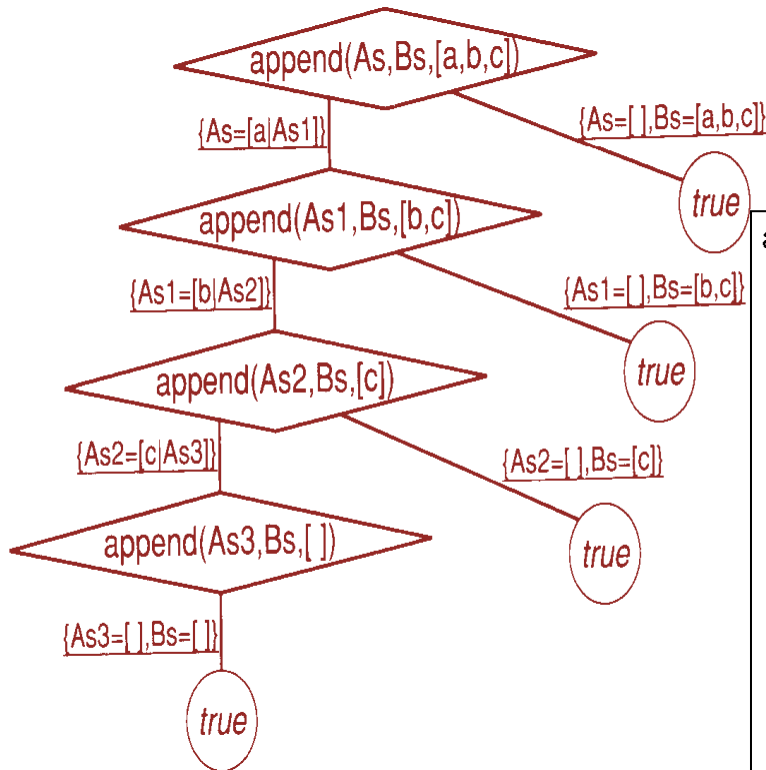
father(abraham, isaac).    male(isaac).
father(haran, lot).        male(lot).
father(haran, milcah).     female(yiscah).
father(haran, yiscah).     female(milcah).

son(X, Y) ← father(Y, X), male(X).
daughter(X, Y) ← father(Y, X), female(X).
    
```

```

son(X, haran)?
  father(haran, X)                X=lot
  male(lot)
  true
  Output: X=lot
  ;
  father(haran, X)                X=milcah
  male(milcah)                    f
  father(haran, X)                X=yiscah
  male(yiscah)                    f
  no (more) solutions
    
```

Traçado (2)



$\text{append}([X|Xs], Ys, [X|Zs]) \leftarrow \text{append}(Xs, Ys, Zs).$
 $\text{append}([], Ys, Ys).$

$\text{append}(Xs, Ys, [a, b, c])$	$Xs = [a Xs1]$
$\text{append}(Xs1, Ys, [b, c])$	$Xs1 = [b Xs2]$
$\text{append}(Xs2, Ys, [c])$	$Xs2 = [c Xs3]$
$\text{append}(Xs3, Ys, [])$	$Xs3 = [], Ys = []$
<i>true</i>	
<i>Output:</i> $(Xs = [a, b, c], Ys = [])$	
;	
$\text{append}(Xs2, Ys, [c])$	$Xs2 = [], Ys = [c]$
<i>true</i>	
<i>Output:</i> $(Xs = [a, b], Ys = [c])$	
;	
$\text{append}(Xs1, Ys, [b, c])$	$Xs1 = [], Ys = [b, c]$
<i>true</i>	
<i>Output:</i> $(Xs = [a], Ys = [b, c])$	
;	
$\text{append}(Xs, Ys, [a, b, c])$	$Xs = [], Ys = [a, b, c]$
<i>true</i>	
<i>Output:</i> $(Xs = [], Ys = [a, b, c])$	
;	
no (more) solutions	

Programação em Prolog

- A programação em lógica deveria permitir uma programação de alto nível
 - escrever axiomas definindo relações (**lógica**), ignorando a sua utilização pelo mecanismo de execução (**controlo**)
- Contudo: as escolhas do mecanismo de execução não podem ser ignoradas!
 - o modelo de execução do Prolog deve ser tido em conta
 - não basta uma axiomatização correcta e completa
 - eficiência e terminação (computação finita)

Ordem das Cláusulas

- Mudar a ordem das cláusulas de um procedimento corresponde a trocar os ramos da árvore de pesquisa construída para um objectivo
- Trocar os ramos provoca uma travessia da árvore por uma ordem diferente, e consequentemente uma ordem diferente nas soluções encontradas

<pre>parent(terach,abraham). parent(isaac,jacob).</pre>	<pre>parent(abraham,isaac). parent(jacob,benjamin).</pre>
---	---

<pre>ancestor(X,Y) ← parent(X,Y). ancestor(X,Z) ← parent(X,Y), ancestor(Y,Z).</pre>

```
?- ancestor(terach,X).  
X=abraham ;  
X=isaac ;  
X=jacob ;  
X=benjamin ;  
no
```

<pre>ancestor(X,Z) ← parent(X,Y), ancestor(Y,Z). ancestor(X,Y) ← parent(X,Y).</pre>

```
?- ancestor(terach,X).  
X=benjamin ;  
X=jacob ;  
X=isaac ;  
X=abraham ;  
no
```

Ordem das Cláusulas (2)

```
member(X, [X|Xs]).  
member(X, [Y|Ys]) ← member(X, Ys).
```

```
member(X, [Y|Ys]) ← member(X, Ys).  
member(X, [X|Xs]).
```

?- member(X, [1, 2, 3]).

```
member(X, [1, 2, 3])      X=1  
  X=1 ;  
member(X, [1, 2, 3])      X=X1  
  member(X1, [2, 3])      X1=2  
    X=2 ;  
  member(X1, [2, 3])      X1=X2  
    member(X2, [3])       X2=3  
      X=3 ;  
    member(X2, [3])       X2=X3  
      member(X3, [])  
        f
```

```
member(X, [1, 2, 3])      X=X1  
  member(X1, [2, 3])      X1=X2  
    member(X2, [3])       X2=X3  
      member(X3, [])  
        f  
    member(X2, [3])       X2=3  
      X=3 ;  
    member(X1, [2, 3])      X1=2  
      X=2 ;  
  member(X, [1, 2, 3])      X=1  
    X=1 ;  
  no
```

Terminação

- Se a árvore de pesquisa tiver um ramo infinito, a computação pode não terminar
 - o Prolog, fazendo pesquisa em profundidade, pode não conseguir encontrar a solução
- A não terminação tem origem nas regras recursivas

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

?- append(Xs,[c,d],Ys).

append(Xs,[c,d],Ys)	Xs=[X Xs1], Ys=[X Ys1]
append(Xs1,[c,d],Ys1)	Xs1=[X1 Xs2], Ys1=[X1 Ys2]
append(Xs2,[c,d],Ys2)	Xs2=[X2 Xs3], Ys2=[X2 Ys3]
append(Xs3,[c,d],Ys3)	Xs3=[X3 Xs4], Ys3=[X3 Ys4]
⋮	⋮

```
married(X,Y) ← married(Y,X).  
married(abraham,sarah).
```

?- married(abraham,sarah).

```
married(abraham,sarah)  
married(sarah,abraham)  
married(abraham,sarah)  
married(sarah,abraham)  
⋮
```

- Evitar *recursividade à esquerda*:

```
are_married(X,Y) ← married(X,Y).  
are_married(X,Y) ← married(Y,X).
```

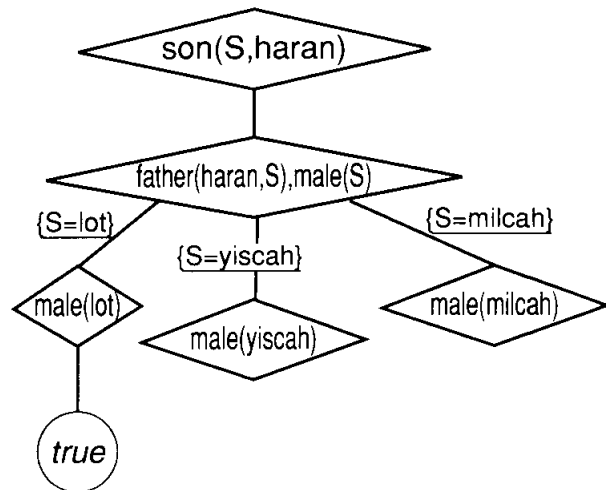
Análise de Programas Recursivos

- Determinar que perguntas terminam em relação a um programa recursivo
- `append`
 - termina se 1º ou 3º argumento é uma lista completa
 - não termina quando o 1º e 3º argumentos são listas incompletas unificáveis
- `member`
 - termina se o 2º argumento é uma lista completa
 - não termina se o 2º argumento é uma lista incompleta
- Cuidado com definições circulares:
$$\begin{aligned}\text{parent}(X,Y) &\leftarrow \text{child}(Y,X). \\ \text{child}(X,Y) &\leftarrow \text{parent}(Y,X).\end{aligned}$$

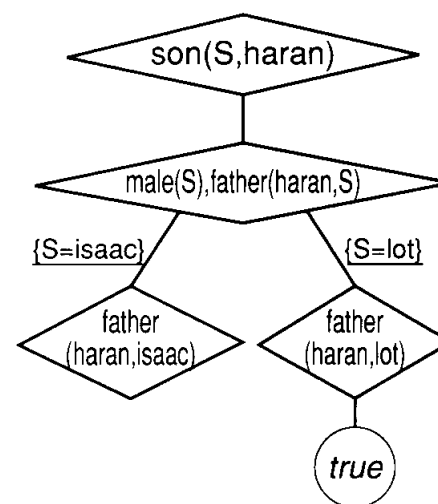
Ordem dos Objectivos

- Ordem das regras não determina a árvore de pesquisa
 - a ordem da travessia é que é diferente
- Ordem dos objectivos **determina a árvore de pesquisa**
 - ordem diferente obtém árvore de pesquisa diferente
 - logo, o esforço de pesquisa da solução será também diferente!

`son(X,Y) ← father(Y,X), male(X) .`



`son(X,Y) ← male(X), father(Y,X) .`



- Qual é melhor?
- Depende do uso: `son(sarah, X)` ?

Ordem dos Objectivos (2)

```
grandparent(X,Z) ← parent(X,Y), parent(Y,Z).
```

```
grandparent(X,Z) ← parent(Y,Z), parent(X,Y).
```

- Para perguntas do tipo `grandparent(abraham, GC) ?` a 1ª regra é melhor
- Para perguntas do tipo `grandparent(GP, isaac) ?` a 2ª regra é melhor
- Se a eficiência é um aspecto importante, definir relações distintas:

```
grandparent(X,Z) ← parent(Y,Z), parent(X,Y).
```

```
grandchild(Z,X) ← parent(X,Y), parent(Y,Z).
```

- Troca dos objectivos pode levar a *recursividade à esquerda*: ramo infinito?

```
ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y).
```

- Não `ancestor(X,Y) ← ancestor(Z,Y), parent(X,Z).`

– Mas:

```
reverse([ ],[ ]).
```

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

- termina se o 1º argumento é uma lista completa
- se os objectivos forem trocados, o critério de terminação passa para o 2º argumento

Heurísticas de Ordenação

- Colocar testes primeiro

```
partition([X|Xs],Y,[X|Ls],Bs) ← X ≤ Y, partition(Xs,Y,Ls,Bs).  
partition([X|Xs],Y,Ls,[X|Bs]) ← X > Y, partition(Xs,Y,Ls,Bs).  
partition([ ],Y,[ ],[ ]).
```

- Colocar primeiro os objectivos com menos soluções
 - depende da base de dados
- Colocar primeiro os objectivos mais instanciados
 - depende do uso
- Objectivo: **falhar o mais rápido possível!**
 - falhar significa podar a árvore de pesquisa, levando mais depressa à solução


Soluções Redundantes

- Havendo várias formas de obter a mesma solução, a árvore de pesquisa é desnecessariamente maior
 - logo, mais demora a computação
- Será melhor manter a árvore de pesquisa o mais pequena possível
- Possíveis origens da redundância:
 - cobrir o mesmo caso com regras diferentes

$\text{minimum}(X,Y,X) \leftarrow X \leq Y.$
 $\text{minimum}(X,Y,Y) \leftarrow Y \leq X.$  $\text{minimum}(X,Y,X) \leftarrow X \leq Y.$
 $\text{minimum}(X,Y,Y) \leftarrow Y < X.$

- casos especiais a mais (por vezes motivados por questões de eficiência)

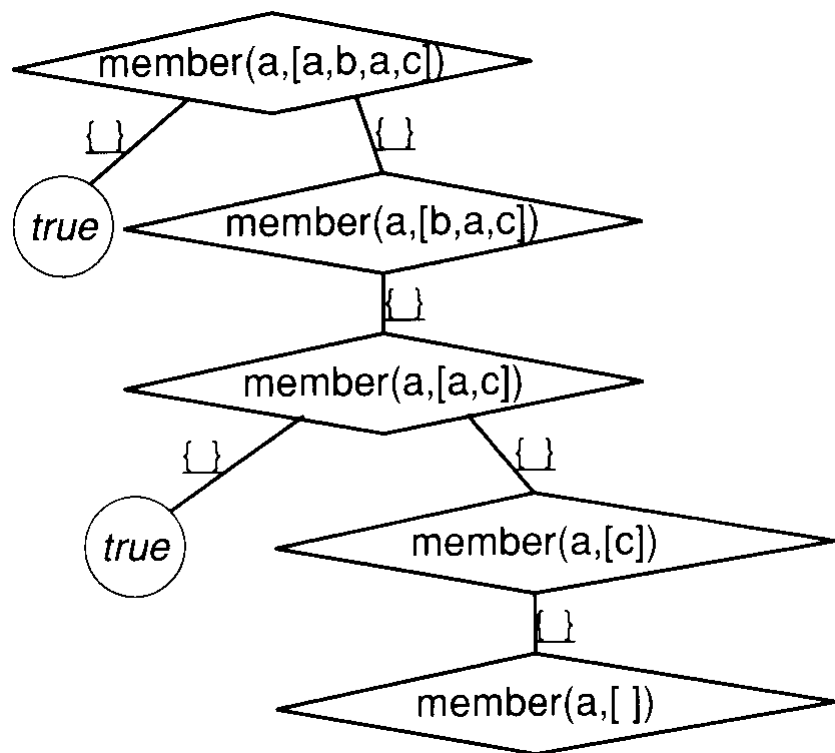
$\text{append}([X|Xs], Ys, [X|Zs]) \leftarrow \text{append}(Xs, Ys, Zs).$
 $\text{append}([], Ys, Ys).$
 $\text{append}(Xs, [], Xs).$

 $\text{append}([X|Xs], [Y|Ys], [X|Zs]) \leftarrow \text{append}(Xs, [Y|Ys], Zs).$
 $\text{append}([], [Y|Ys], [Y|Ys]).$
 $\text{append}(Xs, [], Xs).$

Soluções Redundantes (2)

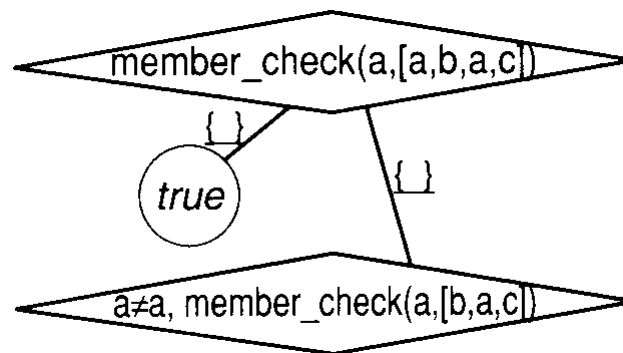
```
member(X, [X|Xs]).
member(X, [Y|Ys]) ← member(X, Ys).
```

?- member(a, [a,b,a,c]).



```
member_check(X, [X|Xs]).
member_check(X, [Y|Ys]) ← X ≠ Y, member_check(X, Ys).
```

?- member_check(a, [a,b,a,c]).



Aritmética

- Predicados de sistema (*built-in*) complementam o “Prolog puro”
 - perguntas usando estes predicados são tratadas de forma especial (*avaliação* em vez de redução)
 - Aritmética
 - maior eficiência: usar capacidades aritméticas do computador
 - desvantagem: perda de generalidade
 - operações aritméticas menos genéricas do que versões baseadas em lógica
 - avaliador aritmético: **is (Value, Expression)**
 - Value **is** Expression
 - a expressão Expression é **avaliada** e o resultado é **unificado** com Value
 - a expressão não pode conter variáveis não instanciadas
 - tem sucesso se a unificação tiver sucesso
- | | | |
|-----------|-----------|-------------|
| X is 3+5? | 8 is 3+5? | 3+5 is 3+5? |
| X=8 | true | false |
- N is N+1? nunca poderá ter sucesso!
 - N instanciado: falha; N variável: erro (expressão não pode ser avaliada)

Operadores

- Aritméticos: $+$, $-$, $*$, $/$, mod
- Unificação: $=$, $\backslash =$
- Comparação: $<$, $=<$, $>$, $>=$, $=:=$, $=\backslash =$
 - ambas as expressões são avaliadas

$1 < 2?$

true

$3+5 = 4+4?$

false

$N = 3?$

$N = 3.$

$2+3 = N, N \text{ is } 5?$

false

$N = 2+3, N \backslash = 5?$

$N = 2+3.$

$3-2 < 2*3+1?$

true

$3+5 =:= 4+4?$

true

$N \text{ is } 3?$

$N = 3.$

$2+3 = N, N =:= 5?$

$N = 2+3.$

$N = 2+3, N = 5?$

false.

$2 < 1?$

false

$3+5 \text{ is } 4+4?$

false

$N =:= 3?$

ERROR

Programas Aritméticos

```
plus(0,X,X) ← natural_number(X).  
plus(s(X),Y,s(Z)) ← plus(X,Y,Z).
```



```
plus(X,Y,Z) ← Z is X+Y.
```

- Restrição nos usos múltiplos

```
plus(3,X,8)?
```

ERROR

```
factorial(0,s(0)).  
factorial(s(N),F) ← factorial(N,F1), times(s(N),F1,F).
```



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

- Perda da estrutura recursiva dos números
 - necessário o cálculo explícito de $N-1$
 - condição $N > 0$ para garantir terminação

Recursividade vs. Iteração

- Computações iterativas são mais eficientes do que as recursivas
 - n invocações recursivas \Rightarrow espaço linear em n
 - programa iterativo usa tipicamente um espaço constante (independente do número de iterações)
- No Prolog, a recursividade é usada para especificar algoritmos recursivos e iterativos
 - **cláusula iterativa**: chamada recursiva é o último objectivo do corpo
 - **procedimento iterativo**: se contiver apenas factos e cláusulas iterativas

Procedimentos Iterativos

```
int factorial(int n) {  
    int i=0, t=1;  
    while(i<n) {  
        i+=1;  
        t*=i;  
    }  
    return t;  
}
```

inicialização

```
factorial(N,F) ← factorial(0,N,1,F).  
factorial(I,N,T,F) ←  
    I < N, I1 is I+1, T1 is T*I1, factorial(I1,N,T1,F).  
factorial(N,N,F,F).
```

- em Prolog não temos variáveis auxiliares para guardar resultados intermédios
- solução: aumentar o procedimento com argumentos extra – *acumuladores*
 - tipicamente um acumulador terá o resultado da computação aquando da terminação
- não esquecer: as variáveis lógicas são “write-once”!
 - daí que seja necessário passar uma nova variável lógica com o novo valor acumulado

```
factorial(N,F) ← factorial(N,1,F).  
factorial(N,T,F) ←  
    N > 0, T1 is T*N, N1 is N-1, factorial(N1,T1,F).  
factorial(0,F,F).
```

Mais Exemplos

```
between(I,J,I)  $\leftarrow$   $I \leq J$ .  
between(I,J,K)  $\leftarrow$   $I < J$ , I1 is I+1, between(I1,J,K).
```

```
sumlist([I|Is],Sum)  $\leftarrow$  sumlist(Is,IsSum), Sum is I+IsSum.  
sumlist([ ],0).
```

```
sumlist(Is,Sum)  $\leftarrow$  sumlist(Is,0,Sum).  
sumlist([I|Is],Temp,Sum)  $\leftarrow$   
    Temp1 is Temp+I, sumlist(Is,Temp1,Sum).  
sumlist([ ],Sum,Sum).
```

```
maxlist([X|Xs],M)  $\leftarrow$  maxlist(Xs,X,M).  
maxlist([X|Xs],Y,M)  $\leftarrow$  maximum(X,Y,Y1), maxlist(Xs,Y1,M).  
maxlist([ ],M,M).  
maximum(X,Y,Y)  $\leftarrow$   $X \leq Y$ .  
maximum(X,Y,X)  $\leftarrow$   $X > Y$ .
```

```
length([X|Xs],N)  $\leftarrow$   $N > 0$ , N1 is N-1, length(Xs,N1).  
length([ ],0).
```

```
length([X|Xs],N)  $\leftarrow$  length(Xs,N1), N is N1+1.  
length([ ],0).
```

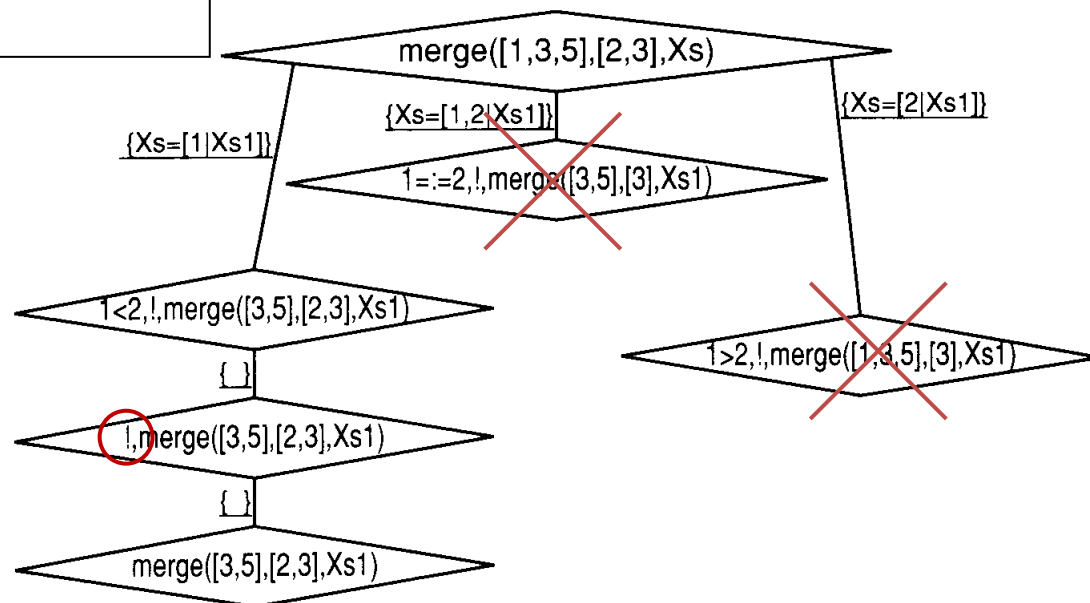
Cut – !

- O *cut* permite afectar o comportamento procedimental dos programas
 - principal função: reduzir o espaço de procura podando dinamicamente a árvore de pesquisa
 - reduz tempo de computação
 - reduz espaço, pois alguns pontos de escolha deixam de ser necessários

```
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).
```

- Expressar exclusividade mútua:

```
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) ← !.
```

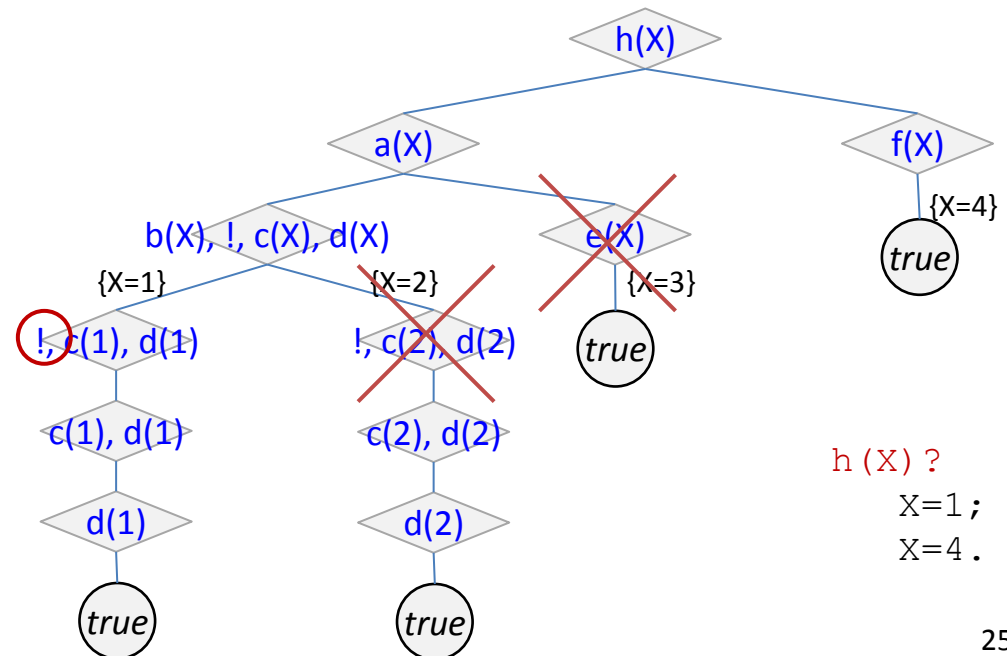


Cut – ! (2)

O *cut* sucede e compromete o Prolog com todas as escolhas feitas desde que o objectivo pai foi unificado com a cabeça da cláusula onde o *cut* ocorre.

- portanto:
 - o *cut* corta todas as cláusulas (do mesmo predicado) abaixo dele
 - o *cut* corta todas as soluções alternativas para os objectivos à sua esquerda na cláusula
 - o *cut* não afecta os objectivos à sua direita!
 - em caso de retrocesso no *cut*, a pesquisa continuará a partir da última escolha feita antes da escolha desta cláusula

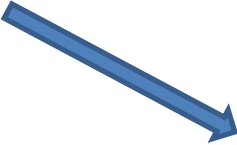
```
h(X) :- a(X) .  
h(X) :- f(X) .  
a(X) :- b(X), !, c(X), d(X) .  
a(X) :- e(X) .  
b(1). b(2) .  
c(1). c(2) .  
d(1). d(2) .  
e(3) .  
f(4) .
```



Remove Computações Redundantes

```
sort(Xs,Ys) ←  
  append(As,[X,Y|Bs],Xs),  
  X > Y,  
  append(As,[Y,X|Bs],Xs1),  
  sort(Xs1,Ys).  
sort(Xs,Xs) ← ordered(Xs).
```

não interessa quais dois elementos são trocados,
desde que $X > Y$
(no final a lista ordenada só tem uma solução)



```
sort(Xs,Ys) ←  
  append(As,[X,Y|Bs],Xs),  
  X > Y,  
  !,  
  append(As,[Y,X|Bs],Xs1),  
  sort(Xs1,Ys).  
sort(Xs,Xs) ←  
  ordered(Xs),  
  !.
```

Optimização com Recursividade em Cauda

- Possibilidade de executar um procedimento recursivo como iterativo (espaço constante)
- $A' \leftarrow B_1, B_2, \dots, B_n$.
 - possível aplicação da optimização ao último objectivo da cauda (B_n) desde que não haja pontos de escolha desde que esta cláusula foi seleccionada
 - não há cláusulas alternativas
 - não há pontos de escolha para B_1, \dots, B_{n-1} (i.e., este objectivo conjuntivo foi resolvido de forma determinística)

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

- com lista completa no 1º argumento, a invocação recursiva cumpre os requisitos!
 - requer análise da segunda cláusula
- Técnica:
 - cláusulas da forma $A' \leftarrow B_1, B_2, \dots, !, B_n$.
 - predicado determinístico

```
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) ← !.
```

Implementação da Negação

```
not X ← X, !, fail.  
not X.
```

- `fail`: predicado que provoca falhanço
- `not G` falha se `G` sucede e sucede se `G` falha
- A ordem das regras é essencial
 - se as regras forem trocadas não é apenas a ordem das soluções que se altera, mas sim o significado do programa!
- Se `G` termina, `not G` também; se `G` não termina, `not G` pode terminar ou não
 - termina (sem sucesso) se for encontrada uma solução para `G` antes de um ramo infinito
- Esta é uma implementação incompleta da negação por falha

```
p(s(X)) ← p(X).  
q(a).
```

`not(p(X),q(X))?`

```
unmarried_student(X) ← not married(X), student(X).  
student(bill).  
married(joe).
```

`unmarried_student(X)?`

- Se usado com objectivos não totalmente instanciados, `not` pode não funcionar

Cuts Verdes e Cuts Vermelhos

- *Cut verde*
 - não altera o significado do programa: o mesmo conjunto de soluções é encontrado com ou sem o *cut*
 - corta apenas caminhos de computação que não levam a novas soluções

```
minimum(X,Y,X) ← X ≤ Y, !.  
minimum(X,Y,Y) ← X > Y, !.
```

- *Cut vermelho*
 - se retirado, altera o significado do programa: o conjunto de soluções será diferente
 - a ordem das cláusulas passa a ser fixa!

```
not X ← X, !, fail.  
not X.
```

Omissão de Condições

```
minimum(X,Y,X) ← X ≤ Y, !.  
minimum(X,Y,Y) ← X > Y, !.
```



```
minimum(X,Y,X) ← X ≤ Y, !.  
minimum(X,Y,Y).
```

mas: `minimum(2, 5, 5)` ? sucede!



```
minimum(X,Y,Z) ← X ≤ Y, !, Z=X.  
minimum(X,Y,Y).
```

- A omissão de condições transforma *cuts* verdes em vermelhos

```
delete([X|Xs],X,Ys) ← !, delete(Xs,X,Ys).  
delete([X|Xs],Z,[X|Ys]) ← X ≠ Z, !, delete(Xs,Z,Ys).  
delete([ ],X,[ ]).
```

```
delete([X|Xs],X,Ys) ← !, delete(Xs,X,Ys).  
delete([X|Xs],Z,[X|Ys]) ← !, delete(Xs,Z,Ys).  
delete([ ],X,[ ]).
```

```
if_then_else(P,Q,R) ← P, !, Q.  
if_then_else(P,Q,R) ← R.
```

- *cut* vermelho evita dupla computação de P (`not P` omitido na 2ª regra)

Predicados de Tipo

```
integer(X) ← X is an integer.  
atom(X) ← X is an atom.  
real(X) ← X is a floating-point number.  
compound(X) ← X is a compound term.  
number(X) ← X is integer or real.  
atomic(X) ← X is an atom or a number.
```

```
flatten([X|Xs],Ys) ←  
    flatten(X,Ys1), flatten(Xs,Ys2), append(Ys1,Ys2,Ys).  
flatten(X,[X]) ←  
    constant(X), X≠[ ].  
flatten([ ],[ ]).  
  
constant(X) ← atomic(X).
```

```
flatten(Xs,Ys) ← flatten(Xs,[ ],Ys).  
flatten([X|Xs],S,Ys) ←  
    list(X), flatten(X,[X|S],Ys).  
flatten([X|Xs],S,[X|Ys]) ←  
    constant(X), X≠[ ], flatten(Xs,S,Ys).  
flatten([ ],[X|S],Ys) ←  
    flatten(X,S,Ys).  
flatten([ ],[ ],[ ]).  
list([X|Xs]).
```

Inspecção de Estruturas

- **functor(Term,F,Arity) / arg(N,Term,Arg)**
 - decomposição de termos
 - `functor(father(haran,lot),X,Y)?`
`{X=father,Y=2}`
 - `arg(2,father(haran,lot),X)?`
`X=lot`
 - criação de termos
 - `functor(T,father,2)?`
`T=father(X,Y)`
 - `arg(1,father(X,lot),haran)?`
`X=haran`
- **Term =.. List**
 - construir um termo a partir de uma lista
 - `X =.. [father,haran,lot]?`
`X=father(haran,lot)`
 - construir uma lista a partir de um termo
 - `father(haran,lot) =.. Xs?`
`Xs=[father,haran,lot]`

Inspecção de Estruturas (2)

```
subterm(Term,Term).
subterm(Sub,Term) ←
    compound(Term), functor(Term,F,N), subterm(N,Sub,Term).
subterm(N,Sub,Term) ←
    N > 1, N1 is N-1, subterm(N1,Sub,Term).
subterm(N,Sub,Term) ←
    arg(N,Term,Arg), subterm(Sub,Arg).
```

```
subterm(t, a(b(c),d(t)))?
    true
subterm(X, a(b(c),d(t)))?
    X = a(b(c),d(t)) ;
    X = b(c) ;
    X = c ;
    X = d(t) ;
    X = t ;
    false
```

```
subterm(Term,Term).
subterm(Sub,Term) ←
    compound(Term), Term =.. [F|Args], subterm_list(Sub,Args).
subterm_list(Sub,[Arg|Args]) ←
    subterm(Sub,Arg).
subterm_list(Sub,[Arg|Args]) ←
    subterm_list(Sub,Args).
```

Predicados Meta-Lógicos

- **var(Term) / nonvar(Term)**

var(X)?

true

var(a)?

false

var([X|Xs])?

false

```
plus(X,Y,Z) ← nonvar(X), nonvar(Y), Z is X+Y.  
plus(X,Y,Z) ← nonvar(X), nonvar(Z), Y is Z-X.  
plus(X,Y,Z) ← nonvar(Y), nonvar(Z), X is Z-Y.
```

plus(1,2,X)?

X=3

plus(1,X,3)?

X=2

plus(X,2,3)?

X=1

plus(X,Y,3)?

false

testes meta-lógicos

```
length(Xs,N) ← nonvar(Xs), length1(Xs,N).  
length(Xs,N) ← var(Xs), nonvar(N), length2(Xs,N).
```

```
ground(Term) ←  
    nonvar(Term), constant(Term).  
ground(Term) ←  
    nonvar(Term),  
    compound(Term),  
    functor(Term,F,N),  
    ground(N,Term).
```

```
ground(N,Term) ←  
    N > 0,  
    arg(N,Term,Arg),  
    ground(Arg),  
    N1 is N-1,  
    ground(N1,Term).  
ground(0,Term).
```

Predicados Meta-Lógicos (2)

- `==` / `\==`

- verificar se dois termos são ou não idênticos
- `X == Y`? tem sucesso se `X` e `Y` são:
 - constantes idênticas
 - variáveis idênticas
 - estruturas com o mesmo nome e aridade, e recursivamente cada `Xi == Yi`? tem sucesso para todos os argumentos de `X` e `Y`, respectivamente

`X == 5?`
false

`X \== Y?`
true

`a(b) == a(X)?`
false

```
occurs_in(X,Term) ←  
  subterm(Sub,Term), X == Sub.
```

`Y = b(X,c), subterm(a,Y)?`

`Y=b(a,c), X=a`

`Y = b(X,c), occurs_in(a,Y)?`
false

`Y = b(X,c), occurs_in(c,Y)?`
`Y=b(X,c)`

Meta-Variável

- Equivalência entre programas e dados
 - ambos podem ser representados como termos lógicos
- Converter um termo num objectivo
 - **call(X)** invoca o objectivo X
- **Meta-variável**: variável usada como um objectivo no corpo de uma cláusula
 - durante a computação, aquando da sua invocação a variável deverá estar instanciada com um termo (se não, erro)
 - **call(G) \equiv G**
- Principais utilizações:
 - meta-programação: meta-interpretadores, *shells*
 - definição da negação
 - definição de predicados de ordem mais elevada

```
read(G), call(G)?  
|: write(qwe).  
   qwe  
   G=write(qwe)
```

X ; Y \leftarrow X.	or(X,Y) :- X.
X ; Y \leftarrow Y.	or(X,Y) :- Y.

Predicados Extra-Lógicos

- Produzem efeitos colaterais quando satisfeitos
- I/O
 - `read(X)`
 - lê termo do canal de entrada e unifica-o com `X`
 - `write(X)`
 - escreve `X` no canal de saída
 - `get_char(Char)` / `put_char(Char)`
 - `nl`
 - muda de linha (*new line*)

```
writeln([X|Xs]) ← write(X), writeln(Xs).  
writeln([ ]) ← nl.
```

```
read(X), writeln(['The value of X is ',X])?  
|: qwe  
The value of X is qwe
```

Strings e Códigos ASCII

- String (entre aspas) corresponde a uma lista de inteiros que são os códigos ASCII de cada carácter na string
 - "The ": [84, 104, 101, 32]
- name (X, Ys)
 - converte átomo X na lista Ys com os códigos ASCII dos caracteres de X
 - name ('The ', Ys) .
Ys = [84, 104, 101, 32]
- put (N)
 - escreve o carácter cujo código ASCII é N
 - put (104)
h
- get0 (N)
 - lê carácter e unifica o seu código ASCII com N
- get (N)
 - lê carácter não branco

Ficheiros

- `see (F)`
 - abre um canal de leitura para o ficheiro `F`
 - as leituras passam a ser feitas a partir de `F`
- `tell (F)`
 - abre um canal de escrita para o ficheiro `F`
 - as escritas passam a ser feitas para `F`
- `seeing (F) / telling (F)`
 - `F` é unificado com o nome do ficheiro no canal corrente
- `seen / told`
 - fecha o canal corrente

Acesso e Manipulação do Programa

- `listing / listing(Pred)`
 - lista as cláusulas do programa/do predicado no canal de saída
- `clause(Head, Body)`
 - procura cláusula cuja cabeça unifica com `Head` (argumento instanciado)
 - em retrocesso, sucede uma vez por cada cláusula unificável
- `assertz(Clause) / asserta(Clause)`
 - adiciona `Clause` como última/primeira cláusula do procedimento
 - no caso de regras, é necessário aplicar parêntesis
 - `assertz((a :- b, c)).`
- `retract(C)`
 - remove a primeira cláusula que unifica com `C`
 - `retract((a :- Body)).`
 - em retrocesso, sucede uma vez por cada cláusula unificável
- `consult(File)`
 - lê e adiciona (`assert`) as cláusulas do ficheiro `File`
- `reconsult(File)`
 - substitui os predicados definidos no ficheiro (`retract` das cláusulas antes de `assert`)

Memorização

- Guardar resultados anteriores por questões de eficiência

```
lemma(P) ← P, asserta((P ← !)).
```

- da próxima vez que **P** for tentado, será utilizada a nova solução (*asserta***a**)
- **!** impede a utilização de cláusulas posteriores

- Torres de Hanói

- N discos: solução tem $2^N - 1$ movimentos
- solução consiste em retirar $N-1$ discos de cima do maior, mover o maior e voltar a deslocar $N-1$ discos para cima dele
- memorizar solução para mover $N-1$ discos

```
hanoi(1,A,B,C,[A to B]).  
hanoi(N,A,B,C,Moves) ←  
    N > 1,  
    N1 is N-1,  
    lemma(hanoi(N1,A,C,B,Ms1)),  
    hanoi(N1,C,B,A,Ms2),  
    append(Ms1,[A to B|Ms2],Moves).
```

```
test_hanoi(N,Pegs,Moves) ←  
    hanoi(N,A,B,C,Moves), Pegs = [A,B,C].
```

Ciclos

- Baseiam-se em efeitos colaterais (e.g. leitura/escrita)
- Programa interativo

```
echo ← read(X), echo(X).  
echo(X) ← last_input(X), !.  
echo(X) ← write(X), nl, read(Y), !, echo(Y).
```

- análogo a ciclo “while”
- é iterativo e determinístico (optimização com recursividade em cauda)
- Ciclo baseado em falha
 - análogo a ciclo “repeat”

```
echo ← repeat, read(X), echo(X), !.  
echo(X) ← last_input(X), !.  
echo(X) ← write(X), nl, fail.
```

```
repeat.  
repeat ← repeat.
```

```
tab(N) ← between(1,N,I), put_char(' '), fail.  
tab(N) ← !.
```

Interpretador de Comandos

```
shell ←  
    shell_prompt, read(Goal), shell(Goal).  
shell(exit) ← !.  
shell(Goal) ←  
    ground(Goal), !, shell_solve_ground(Goal), shell.  
shell(Goal) ←  
    shell_solve(Goal), shell.
```

ciclo
interactivo

```
shell_solve(Goal) ←  
    Goal, write(Goal), nl, fail.  
shell_solve(Goal) ←  
    write('No (more) solutions'), nl.
```

ciclo baseado
em falha

```
shell_solve_ground(Goal) ←  
    Goal, !, write('Yes'), nl.  
shell_solve_ground(Goal) ←  
    write('No'), nl.  
  
shell_prompt ← write('Next command? ').
```