# Paradigmas de Linguagens de Programação

Prof. Sergio D. Zorzo
Departamento de Computação - UFSCar
1º semestre / 2013
Aula 3

Adaptação do material do Prof. Daniel Lucredio

# Listas em Prolog

# Listas

- LISTA é uma seqüência ordenada de elementos.
  - Pode ter qualquer comprimento.
  - Elementos de listas podem ser simples ou estruturados (inclusive listas)
  - No Prolog, geralmente são denotadas por colchetes e elementos separados por vírgulas.
- Exemplos:
  - [ ] (lista vazia)
  - [a, b, c]
  - [maria, joao, pedro, carlos]
  - [1, 329, -15, par(a,b), X, [2, c, Y], 2000]

# Listas

- Listas são divididas em:
  - cabeça - primeiro elemento
  - cauda - o que resta tirando o primeiro elemento
- Exemplos:
  - [a, b, c]
    - cabeça: a
    - cauda: [b,c]
  - [X, Y, 234, abc]
    - cabeça: X
    - cauda: [Y, 234, abc]

# Listas

- As partes da lista são combinadas pelo funtor ● (ponto): ● (Cabeça, Cauda)

[a, b, c] equivale a

● (a, ● (b, ● (c, ●())))

# Listas

- A barra vertical separa a cabeça da cauda.
  - [X|Y] representa listas com pelo menos um elemento
- A barra vertical pode separar também vários elementos do início da lista do restante da lista:
  - [X,Y | Z] representa listas com pelo menos dois elementos
- Símbolos antes da barra são ELEMENTOS
- Símbolo após a barra é LISTA

# Unificação de listas

- Não há diferença
  - Mesmo algoritmo pode ser utilizado
  - Basta imaginar que [X|Y] é o mesmo que ●(X,Y)
- Ex:

  ●(a1, ●(a2, ●(a3, a4)))

  ●(X,Y)
- Resultado da unificação:
  - X=a1
  - Y=●(a2, ●(a3, a4))

# Unificação de listas

- Não há diferença
  - Mesmo algoritmo pode ser utilizado
  - Basta imaginar que [X|Y] é o mesmo que ●(X,Y)
- Ex:

  [a1,a2,a3,a4]

  [X|Y]
- Resultado da unificação:
  - X=a1
  - Y=[a2, a3, a4]

| **Lista 1** | **Lista 2** | **Resultado da unificação** |
|---|---|---|
| [a1, a2, a3, a4] | [X\|Y] | X = a1<br>Y = [a2, a3, a4] |
| [a1] | [X\|Y] | X = a1<br>Y = [] |
| [] | [X\|Y] | não unificam |
| [[a, b], c, d] | [X\|Y] | X = [a,b]<br>Y = [c,d] |
| [[ana, Y] \| Z] | [[X, foi], ao, cinema] | X = ana<br>Y = foi<br>Z = [ao, cinema] |
| [[ana, Y] \| Z] | [[X, foi], [ao, cinema]] | X = ana<br>Y = foi<br>Z = [[ao, cinema]] |
| [a, b, c, d] | [X, Y \| Z] | X = a<br>Y = b<br>Z = [c, d] |
| [ana, maria] | [X, Y \| Z] | X = ana<br>Y = maria<br>Z = [ ] |
| [ana, maria] | [X, Y, Z] | não unifica |

# Operações sobre listas

- Operações sobre listas freqüentemente usam busca recursiva.
- São o mecanismo principal para iteração em Prolog
- O programa é construído com base nas duas partes da lista: *cabeça* e *cauda*.

# Exemplo

- Verificar se um elemento é membro de uma lista.
- O raciocínio para construção de um programa em Prolog começa com a identificação de parâmetros envolvidos (listas, estruturas, elementos, etc) – que serão argumentos de uma relação – e da operação principal – que será definida como uma relação entre esses parâmetros.
- Neste exemplo, temos dois parâmetros, a lista e o elemento.
  - A operação principal é a verificação de pertinência ou não do elemento à lista.
- É necessário definir nomes para:
- relação : pertence  e parâmetros: elemento, lista

# Exemplo

- Depois de definidos os elementos envolvidos, estruturamos a solução como um processo recursivo, explorando o recurso de acessar diretamente a cabeça da lista e a cauda da lista.
- Estruturação da solução:
  - X é membro de L se:
    - X é a cabeça de L, ou
    - X é membro da cauda de L
- Demonstração (ex1)

# Exemplo

- Concatenar duas listas
  - conc(L1, L2, L3)
  - Raciocínio:
    - Se L1 é lista vazia, o resultado da concatenação é igual a L2
    - Se L1 não é vazia, é da forma [X|L]. O resultado da concatenação é [X|LR] onde LR é a concatenação de L com L2.
- Demonstração (ex2)

# Exemplo

- Adicionar um elemento como último de uma lista
- add_ultimo(E, L1, L2)
  - L2 deve conter todos os elementos de L1 e E no final

- Demonstração (ex3)

# Exemplo

- Eliminar um elemento de uma lista
- del(X,L1,L2)
  - L2 contém todos os elementos de L1, menos X

- Demonstração (ex4)

# Exercício

- Defina um predicado membro(X,L) que é satisfeito quando X é um elemento da lista L.

# Resposta

- membro(X,[X|_]).
- membro(X,[_|Z]):- membro(X,Z).

# Aritmética

# Aritmética

- Operadores aritméticos são considerados funtores
  - 2+5 é representado internamente como +(2,5)
- Para ativar as operações é necessário usar o predicado “is”:
  - Sintaxe: X is <expressão>
  - onde X (variável)
  - <expressão> (expressão aritmética)
- calcula a expressão e instancia o resultado na variável X
- Demonstração

# Aritmética

- Porque a consulta 1 is sin(pi/2) retorna falso?

- Precisamos analisar a definição do predicado “is”
  - Vamos olhar no manual do Prolog
  - Mas antes...

# Convenção de notação

- Definição de Predicados
  - pred(+Arg1, ?Arg2, -Arg3)
- Modo de Declaração
  - + Argumento de entrada. Deve estar instanciado quando o predicado é chamado
  - - Argumento de saída. Deve ser uma variável não instanciada quando o predicado é chamado. Se o predicado der sucesso, será instanciada ao valor retornado
  - ? Argumento de entrada ou de saída. Pode estar instanciado ou não
- pred/3
  - Indica que pred tem 3 argumentos
- Prolog adota essa notação (manual, mensagens)

# Operador “is”

```
-Number is +Expr                                                    [ISO]
    True  when Number is the value to which Expr evaluates.   Typically,
    is/2  should be used with unbound left  operand.  If equality  is to
    be tested, =:=/2 should be used.  For example:

            ?- 1 is sin(pi/2).   Fails!.   sin(pi/2) evaluates
                                 to the float  1.0, which does
                                 not unify with the integer 1.
            ?- 1 =:= sin(pi/2).  Succeeds as expected.
```

- -Number is +Expr
- Ou, na notação infixa
- is(-Number, +Expr)
  - “is” deve ser usado com Number não atribuído, ou seja, Number é um “argumento de saída”

# Operadores aritméticos

- Alguns operadores aritméticos que podem ser usados com “is”:

X+Y
X-Y
X*Y
X/Y
X//Y (divisão inteira)
X^Y (exponenciação)
-X
X mod Y

abs(X)
exp(X)
ln(X)
log(X)
sin(X)
cos(X)
sqrt(X)

# Operadores relacionais

- E1 e E2 são expressões aritméticas, que são calculadas antes da aplicação do operador
  - Op(+E1, +E2)
- E1 > E2
- E1 < E2
- E1 >= E2
- E1 =< E2
- E1 =:= E2 *calcula E1 e E2 e testa igualdade*
- E1 =\= E2 *calcula E1 e E2 e testa desigualdade*
- Demonstração

# Comparação entre termos

- Predicado = (unifica termos)
- Sintaxe: Termo1 = Termo2
  - Ou =(?Termo1, ?Termo2)
- Retorna sucesso se os termos Termo1 e Termo2 unificam.
- Retorna os valores das variáveis instanciadas, quando elas aparecem em um dos termos.
- Demonstração

# Comparação entre termos

- Predicado \=
  - Equivalente a \(Termo1 = Termo2)
  - Faz a unificação
    - Se for bem sucedida, retorna false
    - Caso contrário, retorna true

# Comparação entre termos

- Predicado == (verifica se dois termos são
- idênticos)
  - Sintaxe: Termo1 == Termo2
  - Ou ==(?Termo1,?Termo2)
- Retorna sucesso se Termo 1 e Termo2 são idênticos. As variáveis NÃO são instanciadas. As expressões NÃO são calculadas.

# Predicado “==”

- Variáveis só são iguais se representam a mesma instância
- Átomos e Strings são comparados alfabeticamente
- Números são comparados por valor
- Termos compostos:
  - Primeiro checa-se a aridade
  - Depois checa-se o funtor (alfabeticamente)
  - Depois os argumentos, recursivamente
- Demonstração

# Comparação entre termos

- Predicado \== (verifica se dois termos não são idênticos)
- Sintaxe: Termo1 \== Termo2
  - Ou \==(?Termo1,?Termo2)
- Retorna sucesso se Termo 1 e Termo2 NÃO são idênticos. As variáveis NÃO são instanciadas. As expressões NÃO são calculadas.
- É equivalente a \(Termo1==Termo2)

# Ordenação de termos

- Outras comparações: Termo1 <op> Termo2
- onde <op> pode ser @>=, @>, @=<, @<
- Comparam Termo1 e Termo2 seguindo sua ordem natural
  - Variáveis < Números < Átomos < Strings < Compostos
- Variáveis: pela idade (mais velhas primeiro)
  - X @< Y se X foi declarada antes
- Átomos e Strings: pela ordem lexicográfica
- Números: pelos valores
- Compostos: pela aridade, depois pelo funtor (lexicograficamente), e em seguida pelos argumentos
- Demonstração

# Resumo dos operadores

| Operador | Descrição Formal | Propósito |
|---|---|---|
| **is** | -Number is +Expr | Realizar cálculo de expressões, com o objetivo de unificar o resultado com uma variável ainda não instanciada |
| **=:=**<br>**(=\=)** | +Expr1 =:= +Expr2 | Comparar a igualdade do valor de duas expressões, após as mesmas serem calculadas |
| **=**<br>**(\=)** | ?Termo1 = ?Termo1 | Unificar dois termos. Retorna o resultado da unificação (uma lista de substituições) caso seja bem sucedida, ou false |
| **==**<br>**(\==)** | ?Termo1 == ?Termo2 | Comparação entre termos, com base em seu nome, valor e estrutura |

# Resumo dos operadores

| Operador | Descrição Formal | Propósito |
|---|---|---|
| **<**<br>**>**<br>**=<**<br>**>=** | +Expr1 <op> +Expr2 | Comparação aritmética entre duas expressões, após as mesmas serem calculadas |
| **@<**<br>**@>**<br>**@=<**<br>**@>=** | ?Termo1 <op> ?Termo2 | Comparação entre termos segundo sua ordem natural |

# Exemplo

- Somar os elementos de uma lista numérica

# Resposta

```
soma([],0).
soma([Elem| Cauda], S) :- soma(Cauda,S1),
                          S is S1 + Elem.
```

# Exemplo

- Contar o número de elementos de uma lista

# Resposta

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

# Exemplo

- Dada uma lista de números, separar em duas sendo uma com os positivos e o zero, e outra com os negativos

# Resposta

```
separa([],[],[]).
separa([Cabeca|Cauda],[Cabeca|Negativos],NaoNegativos) :-
            Cabeca < 0,
            separa(Cauda,Negativos,NaoNegativos).
separa([Cabeca|Cauda],Negativos,[Cabeca|NaoNegativos]) :-
            Cabeca >= 0,
            separa(Cauda,Negativos,NaoNegativos).
```

# Controle de retrocesso

# Controle de retrocesso

- Processo de resolução é não-determinístico
  - Não há uma ordem específica para aplicar a resolução
  - Ou seja, os programas a seguir são equivalentes:
- P1:

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

- P2:

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

# Controle de retrocesso

- A resolução SLD consiste em justamente definir tal ordem
  - Regra de computação
  - Estratégia de busca
  - Retrocesso (backtracking)
- Nesse contexto, P1 e P2 (do slide anterior) são distintos
  - Mais especificamente, P1 causa um loop infinito

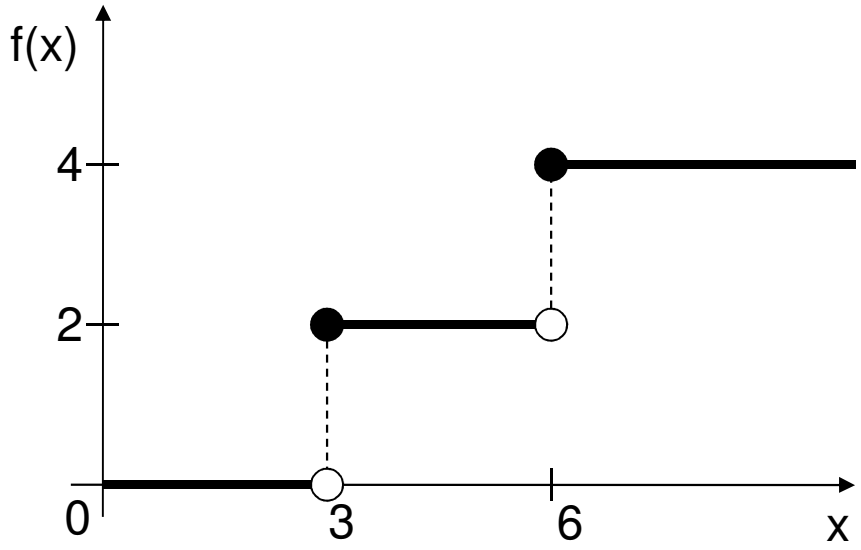
# Controle de retrocesso

- Problema: uma simples mudança de ordem não deveria ser crucial para a exatidão do programa
  - Ausência de controle explícito é uma das vantagens da programação lógica
- Mas é necessária para possibilitar a implementação eficiente
- Existe outra forma para aumentar a eficiência em programas lógicos
  - Faz parte da “impureza” do Prolog

# Controle de retrocesso

- Considere a seguinte função:

$$f(x) = \begin{cases} 0, \text{ se } x < 3 \\ 2, \text{ se } 3 \leq x < 6 \\ 4, \text{ se } x \geq 6 \end{cases}$$



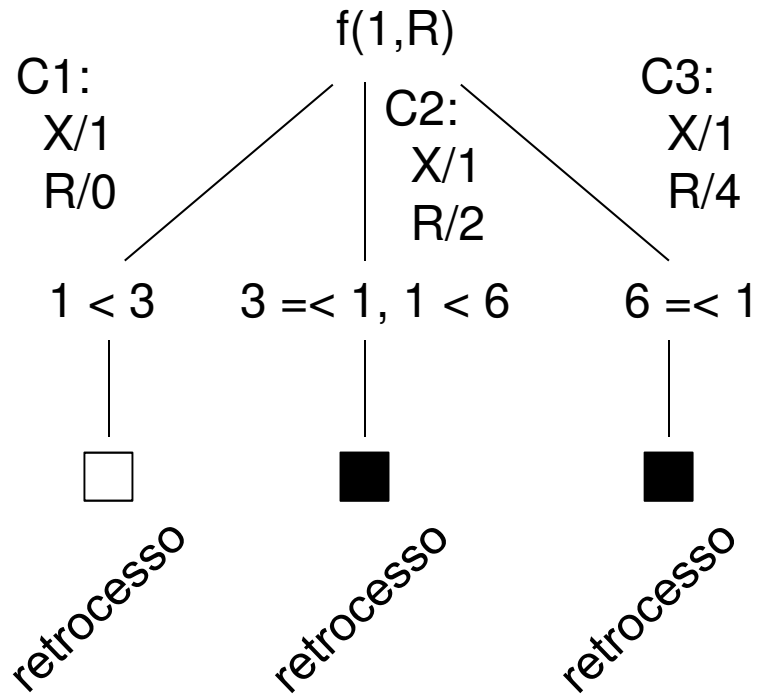
# Controle de retrocesso

- O seguinte programa Prolog implementa a função do slide anterior:

```
f(X,0) :- X < 3.                    %C1
f(X,2) :- 3 =< X, X < 6.            %C2
f(X,4) :- 6 =< X.                   %C3
```

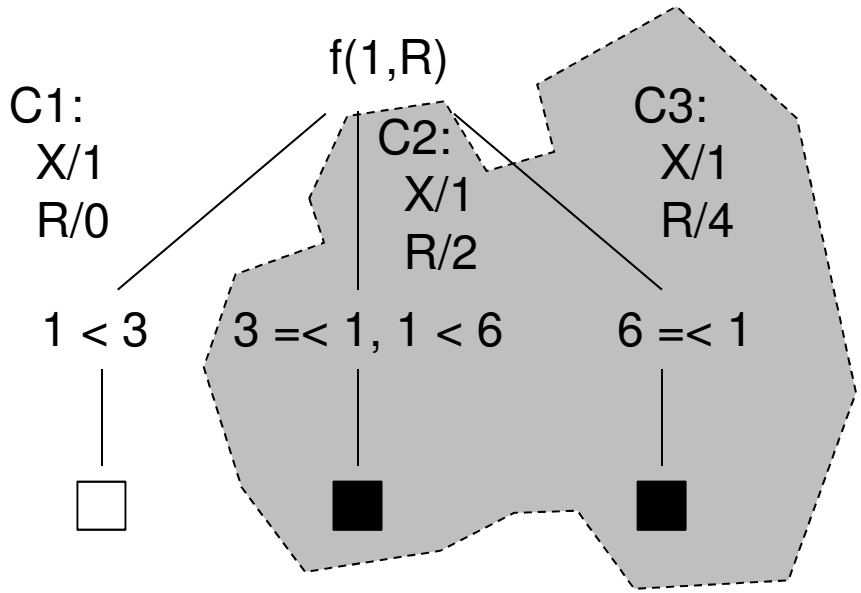
# Controle de retrocesso

- Exemplo de consulta: f(1,R)
  - Demonstração



# Controle de retrocesso

- Nesse programa, sabemos que as três cláusulas de f(X) são mutuamente exclusivas
  - Lembrando: nosso conhecimento permite afirmar isso, o programa não diz nada sobre isso
  - Idealmente, poderíamos “podar” essa árvore



# Controle de retrocesso

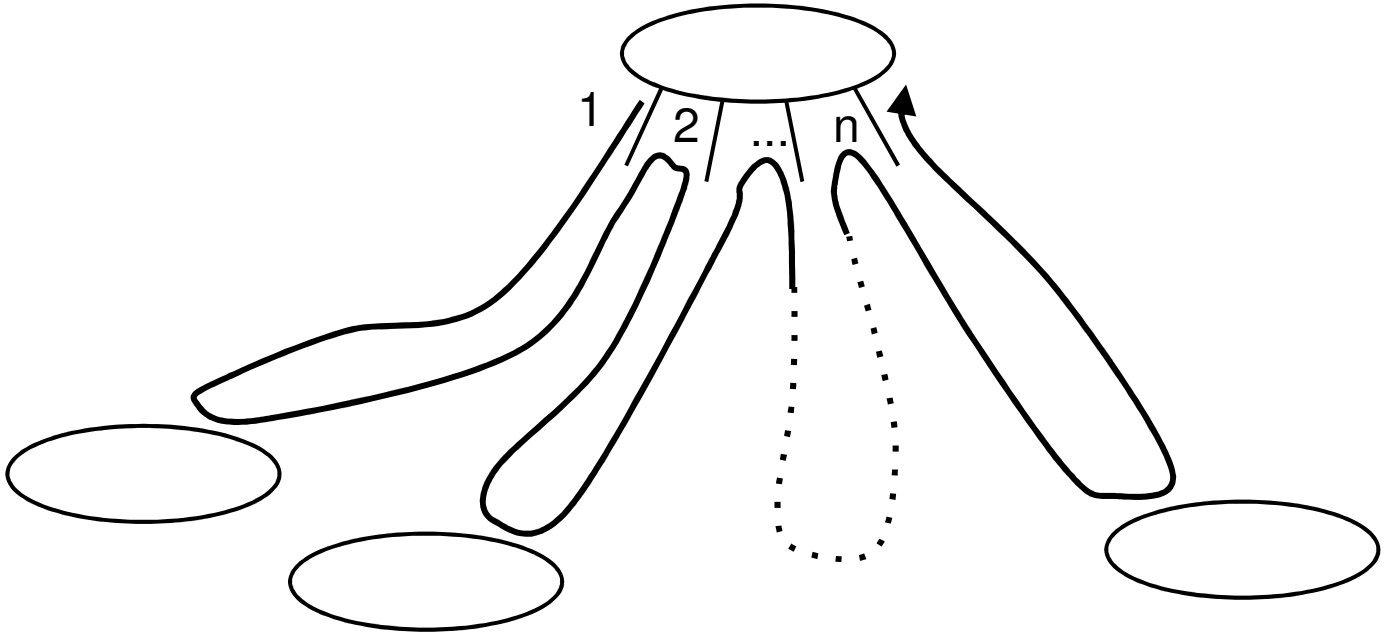
- Predicado de “corte”
  - Símbolo !
  - É uma cláusula com valor sempre verdadeiro
    - Mas possui um efeito colateral
- Ex:

```
a :- b, c, !, d, e
```

  - Depois de avaliar “b”, “c” e “!”, o interpretador automaticamente elimina:
    - Todas as formas *ainda não testadas* de avaliar a, b e c
  - De forma que em futuros retrocessos, caminhos alternativos a partir desses literais são ignorados

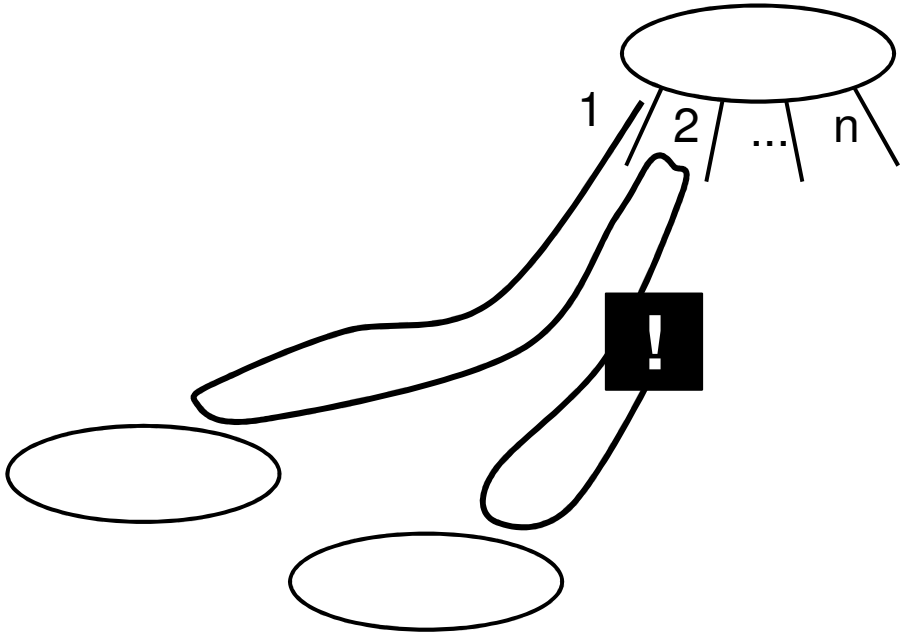
# Controle de retrocesso

- Como funciona o retrocesso em um nó



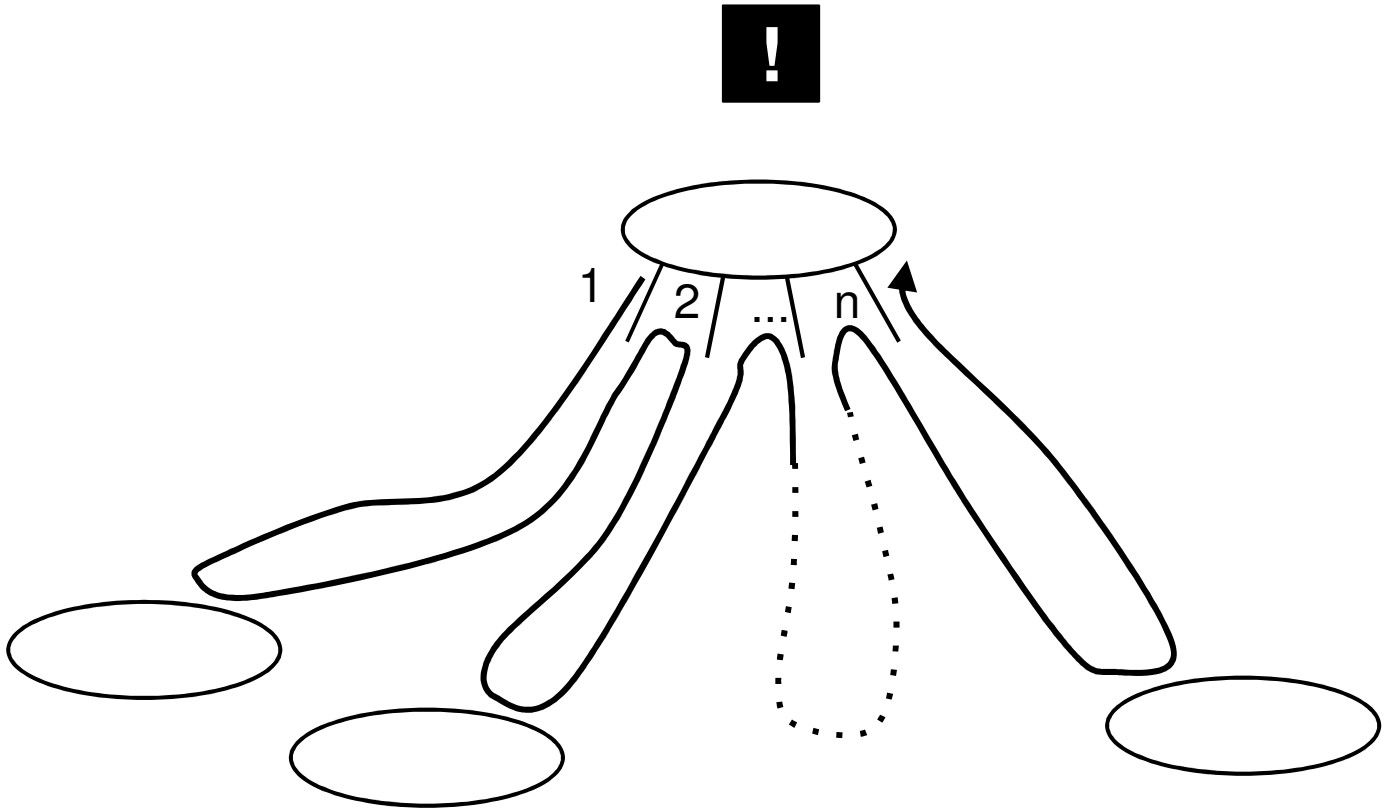
# Controle de retrocesso

- O predicado de corte “pra baixo” de um nó encerra todo retrocesso naquele nó



# Controle de retrocesso

- Se o corte estiver “pra cima”, não influencia no processo daquele nó



# Controle de retrocesso

- Melhorando a eficiência do programa anterior:
  - Demonstração

```
f(X,0) :- X < 3, !.                %C1
f(X,2) :- 3 =< X, X < 6, !.        %C2
f(X,4) :- 6 =< X.                  %C3
```

# Controle de retrocesso

- Melhorando (ainda mais) a eficiência do programa anterior:
  - Demonstração

```
f(X,0) :- X < 3, !.
f(X,2) :- X < 6, !.
f(_,4).
```

# Outro exemplo

- Sem corte:

```
membro(X, [X|L]).
membro(X, [Y|L]) :- membro(X, L).
```

- Com corte:

```
membro(X, [X|L]) :- !.
membro(X, [Y|L]) :- membro(X, L).
```

# Exemplo

- Contar o número de ocorrências de um dado elemento no primeiro nível de uma lista:

```
conta_occor(Elem,[ ],0) :- !.
conta_occor(Elem,[Elem|Y],N) :-
  conta_ocorr(Elem,Y,N1), N is N1 + 1, !.
conta_occor(Elem,[Elem1|Y], N) :-
  conta_ocorr(Elem,Y,N).
```

# Exemplo

- Eliminar todas as ocorrências de um elemento de uma lista:

```
del_todas(Elem,[ ],[ ]) :- !.
del_todas(Elem, [Elem|Y], Z) :-
  del_todas(Elem,Y,Z), !.
del_todas(Elem,[Elem1|Y], [Elem1|Z]) :-
  del_todas(Elem,Y,Z).
```

# Exemplo

- Dada uma lista de números, separar em duas sendo uma com os positivos e o zero, e outra com os negativos:

```
separa([ ],[ ],[ ]) :- !.
separa([X|Y],[X|Z],W) :- X >= 0,
  separa(Y,Z,W), !.
separa([X|Y],Z,[X|W]) :- separa(Y,Z,W).
```

# Exemplo

- Dada uma lista de números, separar em duas sendo uma com os positivos e outra com os negativos, descartando o zero:

```
separa_sz([ ],[ ],[ ]) :- !.
separa_sz([X|Y],[X|Z],W) :-X > 0,
  separa_sz(Y,Z,W), !.
separa_sz([X|Y],Z,[X|W]) :-X < 0,
  separa_sz(Y,Z,W), !.
separa_sz([X|Y],Z,W) :- separa_sz(Y,Z,W).
```

# Corte “verde” / “vermelho”

- Voltemos ao exemplo

```
f(X,0) :- X < 3, !.
f(X,2) :- 3 =< X, X < 6, !.
f(X,4) :- 6 =< X.
```

- O que acontece se removermos o corte?
  - Resposta: menor eficiência
    - Mas o programa continua produzindo respostas corretas

# Corte “verde” / “vermelho”

- Voltemos ao exemplo

```
f(X,0) :- X < 3, !.
f(X,2) :- X < 6, !.
f(_,4).
```

- O que acontece se removermos o corte?
  - O programa não irá mais funcionar corretamente

# Corte “verde” / “vermelho”

- Alguns autores distinguem entre dois usos de corte
  - Verde: aquele que pode ser removido sem alterar a lógica
    - Ou: sem o corte, o programa NUNCA leva a respostas erradas
  - Vermelho: aquele que, quando removido, altera a lógica
    - Ou: sem o corte, o programa ALGUMA VEZES leva a respostas erradas

# Corte “verde” / “vermelho”

- O corte permite um estilo de programação “preguiçoso”
  - Ao invés de pensar nas relações lógicas, o programador adota uma estratégia de programação lógica com estilo procedural
  - Dessa forma, acaba produzindo os tais cortes “vermelhos”
  - É uma forma de impureza na programação lógica
    - E que deve ser evitada a todo custo

# Mais um exemplo

- Encontrar o mínimo múltiplo comum
  - Demonstração

```
mmc(X, Y, Z) :- positivo(Z),
     Z mod X =:= 0,
     Z mod Y =:= 0,
     !.
positivo(1).
positivo(Z) :- positivo(W), Z is W+1.
```

# Corte “vermelho”

- Na verdade, o problema desse programa não é o corte, e sim a lógica – errada!
- O predicado mmc na verdade testa apenas se dois números possuem um múltiplo em comum
  - Mas não diz nada sobre ele ser mínimo!!
- Ou seja, a lógica desse programa é que deveria ser chamada de “vermelha”

# Corte “vermelho”

- Mas mesmo se a lógica estiver correta, a inserção de cortes pode levar a um resultado diferente, além da eficiência
  - O que pode acontecer é produzir respostas incompletas (isto é, “podar” resultados válidos)
  - Ou seja: sem o corte,
    - NUNCA produz uma resposta incorreta
    - TODAS as respostas corretas são encontradas
  - Com o corte,
    - NUNCA produz uma resposta incorreta
    - ALGUMAS respostas corretas podem ser deixadas de lado
  - Nesse caso, também não é considerado um corte “verde”, pois altera o resultado do programa

# Solução para o problema do MMC sem “corte”

- Atenção: antes de partir para o próximo slide, tente resolver!
- Idéia geral de (uma) solução:
  - *mmc(a, b) = a*b/ mdc(a,b)*
- Onde mdc é o máximo divisor comum

# Solução para o problema do MMC sem “corte”

```
mmc(X1,X2,Y) :- mdc(X1,X2,Y1),
          Y is X1*X2/Y1.
mdc(X,0,X).
mdc(X,X,X).
mdc(X1,X2,Y) :- X1 < X2,
     mdc(X2, X1, Y).
mdc(X1,X2,Y) :- X1 > X2,
          X2 =\= 0,
          X3 is X1 mod X2,
     mdc(X2, X3, Y).
```

# Verificação de tipo

# Verificação de tipo

| | |
|---|---|
| atom(X) | Testa se X é átomo |
| atomic(X) | Testa se X é átomo, string, inteiro ou real |
| compound(X) | Testa se X é um termo composto |
| float(X) | Testa se X é um número em ponto flutuante |
| integer(X) | Testa se X é número inteiro |
| number(X) | Testa se X é um número |
| nonvar(X) | Testa se X não é variável |
| var(X) | Testa se X é uma variável |
| is_list(X) | Testa se X é uma lista (SWI-Prolog) |

# Verificação de tipo

- Exemplo: construir um programa para desparentizar uma lista, ou seja, eliminar os delimitadores das listas internas, exceto da lista vazia:
- Dada a lista
  - [a,b,[c,d[e,f,[g]],d, [q,[w]], d], []]
- obtemos
  - [a,b,c,de,f,g,d, q,w, d, []]
- Demonstração

# Alteração na base de dados

# Alteração na base de dados

- Alguns predicados do Prolog modificam a base de dados em tempo de execução
  - `abolish(Pred)` apaga todos os predicados especificados pelo seu argumento
    - +Pred especificação de predicado
  - `abolish(Pred,Arid)`
    - +Pred símbolo de predicado
    - +Arid número de argumentos
- Demonstração

# Alteração na base de dados

- `assert(Claus)` adiciona uma cláusula no fim das cláusulas associadas com seu predicado
  - +Claus cláusula (fato ou regra)
- Obs:
  - as regras devem aparecer entre parêntesis.
  - variáveis não instanciadas no momento da execução do assert são consideradas variáveis.
- Demonstração

# Alteração na base de dados

- `assert(Claus, Pos)` adiciona uma cláusula na posição especificada
  - +Claus cláusula (fato ou regra)
  - +Pos inteiro >= 0
- `asserta(Claus)` adiciona uma cláusula no começo das cláusulas associadas
- `assertz(Claus)` adiciona uma cláusula no fim das cláusulas associadas

# Alteração na base de dados

- `retract(Claus)` elimina uma cláusula que unifica com a cláusula dada
  - +Claus cláusula
- Procura a primeira cláusula na base de dados que unifica com Claus, se encontrar apaga.
  - Variáveis são instanciadas.
  - Permite backtracking.
- Demonstração

# Entrada e saída

# Entrada e saída

- Predicado `read`
  - Sintaxe: read(Termo)
    - onde ?Termo (variável ou átomo)
  - Lê um termo do dispositivo de entrada corrente e unifica com Termo. O termo dado deve ser seguido de . (ponto)
- Predicado `write`
  - Sintaxe: write(Termo)
    - onde ?Termo (termo)
  - Escreve o termo no dispositivo de saída corrente
- Predicado `nl`
  - Muda para próxima linha no dispositivo de saída
- Demonstração

# Predicados sem argumentos e predicados com o mesmo nome

# Predicados sem argumentos e predicados com o mesmo nome

- Um predicado é identificado pelo seu nome e pela aridade (número de argumentos).
- Predicados com o mesmo nome e com número de argumentos diferentes são considerados diferentes.
  - Os predicados sem argumentos são normalmente usados para identificar procedimentos que usam read e write ou para iniciar programas com muitos predicados.
- Demonstração

# Meta-variáveis

# Meta-variáveis

- Aparecem no lugar de uma estrutura prolog que pode ser executada.
  - Meta-variável-condição
- Aparece como um sub-objetivo no corpo de uma regra.
  - Não pode aparecer na cabeça da regra.
  - Na hora da execução deve estar instanciada com:
    - um átomo
    - um termo composto
- Demonstração

# Negação

# Negação

- É comum a afirmação de que “Prolog tem um problema com negação”
  - Na verdade, essa afirmação não é 100% correta
  - Trata-se da forma com que a resolução funciona em sua essência
  - E com o que assumimos ser verdadeiro ou falso, com relação ao mundo
- Pergunta: tudo o que não é verdadeiro, posso automaticamente assumir que é falso?
  - Resposta:
    - Depende

# Exemplo

- Maria gosta de todos os animais, exceto cobras

```
gosta_de(maria, X) :- animal(X),
                      não-cobra(X).
animal(passaro).
animal(cachorro).
animal(cobra).
não-cobra(passaro).
não-cobra(cachorro).
```

- Inferência dedutiva é computacionalmente completa
  - Mas em muitos casos não é conveniente
  - Principalmente com relação a informação negativa

# Negação

- Existe uma forma de resolver esse problema
- Simplesmente tomando uma posição com relação ao mundo
  - E ao que consideramos verdadeiro/falso
- Pressuposição de mundo fechado

# Inferência padrão (default)

- Modo de raciocínio complementar à inferência dedutiva
- Conceito de negação-por-falha
- predicado `fail`
  - Sempre causa falha na resolução
- Permite implementar uma semântica negativa
- Com base na pressuposição de mundo fechado

# Negação-por-falha

- Exemplo anterior:

```
gosta_de(maria, X) :- X = cobra,
                      !,
                      fail.
gosta_de(maria, X) :- animal(X).
animal(passaro).
animal(cachorro).
animal(cobra).
```

# Negação-por-falha

```
nao(P) :- P, !, fail.
nao(_).
```

# Negação-por-falha

- SWI-Prolog (e muitos) tem predicados para isso
  - not(G)
  - \+ G
  - Onde G é uma meta a ser testada
- \+ é utilizado como operador
  - Ex:
    - not(cobra(X)).
    - \+cobra(X).
- Mesma lógica que o exemplo anterior
  - Retorna verdadeiro se G for verdadeiro
  - Retorna falso se não for possível provar que G é verdadeiro
- Cuidado especial quando G possui alguma variável livre (não instanciada)

# Bibliografia

- Nesta aula, utilizamos conceitos dos seguintes livros:
  - Sebesta, R.W. Conceitos de Linguagens de Programação – 5ª edição. Bookman, 2003.
  - Hogger, C.J. Essentials of Logic Programming. Oxford, 1990.

# Fim