PROGRAMAÇÃO EM LÓGICA

PROLOG AVANÇADO

Henrique Lopes Cardoso, DEI/FEUP, 2013

Não Determinismo

- Como escolher criteriosamente o próximo passo?
- Geração e teste

```
find(X) ← generate(X), test(X).
```

  - tira partido do mecanismo de retrocesso
  - o gerador tenta escolher correctamente a solução e o testador verifica se essa escolha está correcta
- Exemplos:

```
verb(Sentence,Word) ← member(Word,Sentence), verb(Word).
noun(Sentence,Word) ← member(Word,Sentence), noun(Word).
article(Sentence,Word) ← member(Word,Sentence), article(Word).
```

```
noun(man).      noun(woman).
article(a).     verb(loves).
```

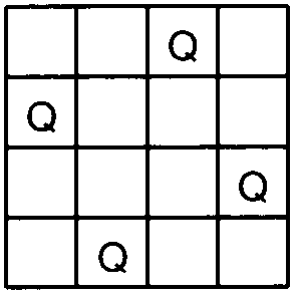
```
verb([a,man,loves,a,woman],V)?
   V=loves
```

```
intersect(Xs,Ys) ← member(X,Xs), member(X,Ys).
```

```
sort(Xs,Ys) ← permutation(Xs,Ys), ordered(Ys).
```

Melhorar Geração de Soluções

- Geração e teste: programas são mais fáceis de construir do que obter directamente a solução, mas são também menos eficientes
- Como optimizar?
  - dirigir a geração de soluções possíveis incluindo o teste nessa geração



```
queens(N,Qs) ←
    range(1,N,Ns), permutation(Ns,Qs), safe(Qs).

safe([Q|Qs]) ← safe(Qs), not attack(Q,Qs).
safe([ ]).

attack(X,Xs) ← attack(X,1,Xs).

attack(X,N,[Y|Ys]) ← X is Y+N ; X is Y-N.
attack(X,N,[Y|Ys]) ← N1 is N+1, attack(X,N1,Ys).
```

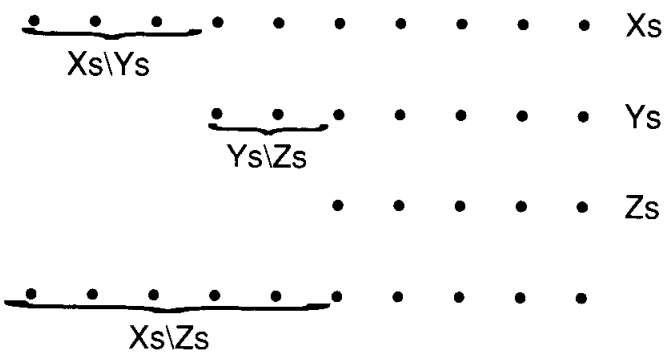
```
queens(N,Qs) ← range(1,N,Ns), queens(Ns,[ ],Qs).

queens(UnplacedQs,SafeQs,Qs) ←
    select(Q,UnplacedQs,UnplacedQs1),
    not attack(Q,SafeQs),
    queens(UnplacedQs1,[Q|SafeQs],Qs).
queens([ ],Qs,Qs).
```

Listas de Diferença

- Representação alternativa para listas: diferença entre pares de listas
  - `[1,2,3]`
    - `[1,2,3,4,5]\[4,5]` `[1,2,3,8]\[8]` `[1,2,3]\[]`
    - usando listas incompletas: `[1,2,3|Xs]\Xs`
  - como as expressões lógicas são unificadas (e não avaliadas), o functor é arbitrário: `\`, `-`, …
  - qualquer lista pode ser representada como lista de diferença:
    - uma lista `L` fica `L\[]` lista vazia: `As\As`

- Vantagem: eficiência
  - duas listas de diferença incompletas podem ser concatenadas em tempo constante numa terceira lista de diferença
    - o `append` concatena listas em tempo linear no tamanho da 1ª lista

Concatenação com Listas de Diferença




```
append_dl(Xs\Ys, Ys\Zs, Xs\Zs).
```

- Condição: para concatenar `As\Bs` com `Xs\Ys`, `Bs` têm que unificar com `Xs` – listas de diferença *compatíveis*
- Se a cauda de uma lista de diferença for não instanciada, essa lista de diferença é compatível com qualquer outra lista de diferença!

```
append_dl([a,b,c|Xs]\Xs,[1,2]\[],Ys)?
   Xs=[1,2], Ys=[a,b,c,1,2]\[]
```

Exemplos

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

```
flatten(Xs,Ys) ← flatten_dl(Xs,Ys\[ ]).

flatten_dl([X|Xs],Ys\Zs) ←
    flatten_dl(X,Ys\Ys1), flatten_dl(Xs,Ys1\Zs).
flatten_dl(X,[X|Xs]\Xs) ←
    constant(X), X≠[ ].
flatten_dl([ ],Xs\Xs).
```

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

```
reverse(Xs,Ys) ← reverse_dl(Xs,Ys\[ ]).

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

```
quicksort([X|Xs],Ys) ←
    partition(Xs,X,Littles,Bigs),
    quicksort(Littles,Ls),
    quicksort(Bigs,Bs),
    append(Ls,[X|Bs],Ys).
quicksort([ ],[ ]).
```

```
quicksort(Xs,Ys) ← quicksort_dl(Xs,Ys\[ ]).

quicksort_dl([X|Xs],Ys\Zs) ←
    partition(Xs,X,Littles,Bigs),
    quicksort_dl(Littles,Ys\[X|Ys1]),
    quicksort_dl(Bigs,Ys1\Zs).
quicksort_dl([ ],Xs\Xs).
```

Estruturas de Dados Incompletas

- Trabalhar com estruturas de dados incompletas facilita a adição de elementos

```
lookup(Key,[(Key,Value)|Dict],Value).
lookup(Key,[(Key1,Value1)|Dict],Value) ←
    Key ≠ Key1, lookup(Key,Dict,Value).
```

```
dict([(arnold,8881),(barry,4513),(cathy,5950)|Xs]).
dict(Dict), lookup(arnold,Dict,N)?
    N=8881
dict(Dict), lookup(david,Dict,1199)?
    Dict = [(arnold,8881),(barry,4513),(cathy,5950),(david,1199)|Xs1]
```

- Usando árvores binárias de pesquisa (ordenadas)

```
lookup(Key,dict(Key,X,Left,Right),Value) ←
    !, X = Value.
lookup(Key,dict(Key1,X,Left,Right),Value) ←
    Key < Key1, lookup(Key,Left,Value).
lookup(Key,dict(Key1,X,Left,Right),Value) ←
    Key > Key1, lookup(Key,Right,Value).
```

Todas as Soluções

- Obter todos os filhos de um pai
  - usando um acumulador:

```
children(X,Kids) :- children(X,[],Kids).
children(X,Cs,Kids) :-
    father(X,Kid), not member(Kid,Cs), !,
    children(X,[Kid|Cs],Kids).
children(X,Cs,Cs).
```

    - abordagem ineficiente: cada solução adicional para `father/2` recomeça no início da árvore de pesquisa
  - alternativa: ciclo baseado em falha combinado com *assert/retract*

```
children(X,Kids) :- assert(kids(X,[])), fail.
children(X,Kids) :-
    father(X,Kid),
    retract(kids(X,Cs)), assert(kids(X,[Kid|Cs])),
    fail.
children(X,Kids) :- retract(kids(X,Kids)).
```

    - percorre a árvore de pesquisa apenas uma vez!
    - mas a asserção/retracção de cláusulas são operações inerentemente pesadas

Findall

- `findall(Term,Goal,Bag)`
  - unifica `Bag` com a lista das instâncias de `Term` para as quais `Goal` é satisfeito
    - todos os `X` para os quais `call(Goal), X=Term?` é satisfeito
    - `Term` e `Goal` tipicamente partilham variáveis

```
children(X,Kids) ← findall(Kid,father(X,Kid),Kids).
```

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

```
children(terach,Xs)?
    Xs = [abraham,nachor,haran].

findall(F,father(F,K),Fs)?
    Fs = [terach,haran,terach,haran,terach,haran,abraham].

findall(F-K,father(F,K),Fs)?
    Fs = [terach-abraham, haran-lot, terach-nachor, haran-milcah,
          terach-haran, haran-yiscah, abraham-isaac].
```

Bagof/Setof

- `bagof(Term,Goal,Bag)`
  - idêntico ao `findall`, mas são encontradas soluções alternativas para as variáveis em `Goal`

```
bagof(F,father(F,K),Fs)?
    K = abraham,
    Fs = [terach] ;
    K = haran,
    Fs = [terach] ;
    K = isaac,
    Fs = [abraham] ;
    K = lot,
    Fs = [haran] ;
    K = milcah,
    Fs = [haran] ;
    K = nachor,
    Fs = [terach] ;
    K = yiscah,
    Fs = [haran].
```

```
bagof(K,father(F,K),Fs)?
    F = abraham,
    Fs = [isaac] ;
    F = haran,
    Fs = [lot, milcah, yiscah] ;
    F = terach,
    Fs = [abraham, nachor, haran].

bagof(K,F^father(F,K),Fs)?
    Fs = [abraham, lot, nachor, milcah,
          haran, yiscah, isaac].
```

- `setof(Term,Goal,Bag)`
  - soluções ordenadas, sem duplicados (conjunto)

Predicados de Segunda Ordem

- Lógica de primeira ordem permite quantificação sobre indivíduos
- Lógica de segunda ordem permite quantificação sobre predicados

```
has_property([X|Xs],P) ← P(X), has_property(Xs,P).
has_property([ ],P).
```

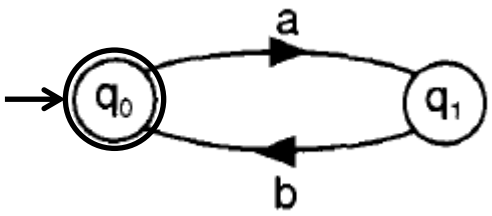
```
G =.. [P,X], G
```

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

```
G =.. [P,X,Y], G
```

Interpretador para (N)DFA

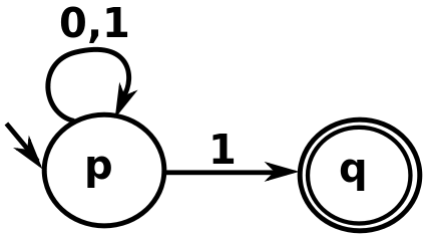
- $NDFA = \langle Q, \Sigma, \delta, I, F \rangle$




```
initial(q0).
final(q0).
delta(q0,a,q1).
delta(q1,b,q0).
```

- Interpretador:

```
accept(Xs) ← initial(Q), accept(Xs,Q).
accept([X|Xs],Q) ← delta(Q,X,Q1), accept(Xs,Q1).
accept([ ],Q) ← final(Q).
```



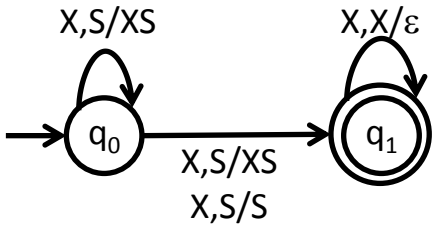

```
initial(p).
final(q).
delta(p,0,p).
delta(p,1,p).
delta(p,1,q).
```

Interpretador para NPDA

- $NPDA = \langle Q, \Sigma, G, \delta, I, Z, F \rangle$

- Palíndromos:

```
initial(q0).    final(q1).

delta(q0,X,S,q0,[X|S]).
delta(q0,X,S,q1,[X|S]).
delta(q0,X,S,q1,S).
delta(q1,X,[X|S],q1,S).
```



- Interpretador:

```
accept(Xs) ← initial(Q), accept(Xs,Q,[ ]).

accept([X|Xs],Q,S) ← delta(Q,X,S,Q1,S1), accept(Xs,Q1,S1).
accept([ ],Q,[ ]) ← final(Q).
```

Gramáticas sem Contexto

- $a^*b^*c^*$

```
s → a, b, c.

a → [a], a.
a → [ ].
b → [b], b.
b → [ ].
c → [c], c.
c → [ ].
```

```
s(Xs) :- append(As,BsCs,Xs), append(Bs,Cs,BsCs),
         a(As), b(Bs), c(Cs).

a(Xs) :- Xs=[a|As], a(As).
a([]).
b(Xs) :- Xs=[b|Bs], b(Bs).
b([]).
c(Xs) :- Xs=[c|Cs], c(Cs).
c([]).
```

- Com listas de diferença:

```
s(As\Xs) :- a(As\Bs), b(Bs\Cs), c(Cs\Xs).

a(Xs\Ys) :- Xs=[a|Xs1], a(Xs1\Ys).
a(Xs\Xs).
b(Xs\Ys) :- Xs=[b|Xs1], b(Xs1\Ys).
b(Xs\Xs).
c(Xs\Ys) :- Xs=[c|Xs1], c(Xs1\Ys).
c(Xs\Xs).
```

Meta-Interpretadores

- Um meta-interpretador de uma linguagem é um interpretador da linguagem escrito na própria linguagem
- Em Prolog, é particularmente fácil construir meta-interpretadores porque não há distinção entre programa e dados
- Interesse em desenvolver meta-interpretadores:
  - implementar diferentes estratégias de pesquisa da solução
  - incluir capacidades de explicação
  - incluir facilidades acrescidas de traçagem, teste e "debugging"

- Um meta-interpretador para Prolog puro (sem *cuts*):

```
solve(true).
solve((A,B)) ← solve(A), solve(B).
solve(A) ← clause(A,B), solve(B).
solve(A) ← builtin(A), A.
```

Meta-Interpretadores (2)

- Com traçagem:

```
solve_trace(Goal) ← solve_trace(Goal,0).

solve_trace(true,Depth) ← !.
solve_trace((A,B),Depth) ←
    !, solve_trace(A,Depth), solve_trace(B,Depth).
solve_trace(A,Depth) ←
    builtin(A), !, A, display(A,Depth), nl.
solve_trace(A,Depth) ←
    clause(A,B), display(A,Depth), nl, Depth1 is Depth + 1,
    solve_trace(B,Depth1).

display(A,Depth) ←
    Spacing is 3*Depth, put_spaces(Spacing), write(A).

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

- Geração da árvore de prova:

```
solve(true,true) ← !.
solve((A,B),(ProofA,ProofB)) ←
    !, solve(A,ProofA), solve(B,ProofB).
solve(A,(A←builtin)) ← builtin(A), !, A.
solve(A,(A←Proof)) ← clause(A,B), solve(B,Proof).
```

Definição de Operadores

- Nome (um átomo), tipo (classe e associatividade) e prioridade (inteiro entre 1 e 1200)

- Tipos de operadores:

Specifier	Class	Associativity
fx	prefix	non-associative
fy	prefix	right-associative
xfx	infix	non-associative
xfy	infix	right-associative
yfx	infix	left-associative
xf	postfix	non-associative
yf	postfix	left-associative

- Definir operadores:

```
:- op(Prioridade,Tipo,Nome).
```

Definição de Operadores (Exemplo)

```
:- op(900, xfy, opa).
:- op(800, yfx, opb).
```

- `1 opa 2 opb 3` ⇔ `(1 opa (2 opb 3))`
  - `opa` tem prioridade mais elevada do que `opb`
- `1 opa 2 opa 3` ⇔ `(1 opa (2 opa 3))`
  - `opa` é um operador associativo à direita

- Verificação:

```
X opa Y :- write(X:Y).
X opb Y :- write(X:Y).

| ?- 1 opa 2 opb 3.
     1 : (2 opb 3)
     yes
| ?- 1 opa 2 opa 3.
     1 : (2 opa 3)
     yes
| ?- 1 opb 2 opb 3.
     (1 opb 2) : 3
     yes
```

Operadores Predefinidos

Priority	Specifier	Operator(s)
1200	xfx	(:- -->)
1200	fx	:-
1100	xfy	;
1000	xfy	,
700	xfx	= \=
700	xfx	== \==
700	xfx	=..
700	xfx	is =:= =\= < <= > >=
500	yfx	+ -
400	yfx	* /
200	xfy	& ^
200	fy	-

Programa genérico para Jogos

```
play(Game) ←
    initialize(Game,Position,Player),
    display_game(Position,Player),
    play(Position,Player,Result).
```

```
play(Position,Player,Result) ←
    game_over(Position,Player,Result), !, announce(Result).
play(Position,Player,Result) ←
    choose_move(Position,Player,Move),
    move(Move,Position,Position1),
    display_game(Position1,Player),
    next_player(Player,Player1),
    !, play(Position1,Player1,Result).
```

Escolher a melhor jogada

```
choose_move(Position,computer,Move) ←
    findall(M,move(Position,M),Moves),
    evaluate_and_choose(Moves,Position,(nil,-1000),Move).


evaluate_and_choose([Move|Moves],Position,Record,BestMove) ←
    move(Move,Position,Position1),
    value(Position1,Value),
    update(Move,Value,Record,Record1),
    evaluate_and_choose(Moves,Position,Record1,BestMove).
evaluate_and_choose([ ],Position,(Move,Value),Move).

update(Move,Value,(Move1,Value1),(Move1,Value1)) ←
    Value ≤ Value1.
update(Move,Value,(Move1,Value1),(Move,Value)) ←
    Value > Value1.
```