PARADIGMAS DE LING. DE PROGRAMAÇÃO (4)
PROGRAMAÇÃO LÓGICA

PROF. WALDEMAR BONVENTI JR. 2023

2 PROGRAMAÇÃO LÓGICA (DECLARATIVA)

- Surgiu como um paradigma distinto nos anos 70.
- Diferente dos outros paradigmas porque ela requer que o programador declare os objetivos da computação,
- em vez dos algoritmos detalhados por meio dos quais esses objetivos podem ser alcançados.
- Às vezes, é chamada programação baseada em regras.
- Dois domínios principais:
- Inteligência artificial
  - Prolog, MYCIN
- Acesso de informações em bancos de dados.
  - SQL

3 PROLOG (PROGRAMMING IN LOGIC)

- A lógica proposicional está relacionada às proposições, que são afirmações que podem ser Verdadeiro (V) ou Falso (F).
- Um exemplo é a proposição "São duas horas da manhã e tudo está em paz".
- REVER tabelas-verdade

- se p então q
- (ou de forma equivalente, p implica q)
- é escrita como p => q.
- Principais quantificadores são
- ∀ ("para todo") e ∃ ("existe").
- Para todo número primo x, existe um outro número primo y, tal que y é maior do que x.

4 ATRIBUTOS OU SENTENÇAS LÓGICAS

Tabela B.5 Propriedades dos Atributos

Propriedade	Significado	
Comutatividade	$p \vee q \Leftrightarrow q \vee p$	$p \wedge q \Leftrightarrow q \wedge p$
Associatividade	$(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$	$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$
Distributividade	$p \vee q \wedge r \Leftrightarrow (p \vee q) \wedge (p \vee r)$	$p \wedge (q \vee r) \Leftrightarrow p \wedge q \vee p \wedge r$
Idempotente	$p \vee p \Leftrightarrow p$	$p \wedge p \Leftrightarrow p$
Identidade	$p \vee \neg p \Leftrightarrow true$	$p \wedge \neg p \Leftrightarrow false$
deMorgan	$\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$
Implicação	$p \Rightarrow q \Leftrightarrow \neg p \vee q$	
Quantificação	$\neg \forall x\, p(x) \Leftrightarrow \exists x \neg p\,(x)$	$\neg \exists x\, p(x) \Leftrightarrow \forall x \neg p\,(x)$

5 PROLOG

- Prolog foi baseada em dois poderosos princípios descobertos por Robinson (1965) chamados resolução e unificação
- A Prolog surgiu em 1970, resultado do trabalho de Colmerauer, Rousseau e Kowalski

- Elementos de um programa
- Constante: átomo (a, zebra, 'Zé', ' . ')
- Variável: inicia com maiúscula: Zé, X, Y
- Estrutura: predicados em notação funcional
  - fala(Quem, russo) Quem fala russo?
  - amarelo(carro) Carro é amarelo
  - f(X,Y) função de X,Y

6 FATOS E REGRAS

- Um fato é um termo seguido de um ponto (.) e é similar a uma cláusula de Horn sem o lado direito;
- OBS.: uma variável não pode ser um fato.
- homem(pedro).
- homem(joao).
- mulher(maria).
- mulher(teresa).

- Uma regra é um termo seguido de :- e uma série de termos separados por vírgulas que termina em um ponto final.
- filho(Y,X) :- genitor(X,Y) .
- mae(X,Y) :- genitor(X,Y) , mulher (X) .
- irma(X,Y) :- genitor(Z ,X), genitor(Z ,Y), mulher (X).
- irma(X,Y) :- genitor(Z ,X), genitor(Z ,Y), mulher (X), not(X=Y).

7 HTTPS://SWISH.SWI-PROLOG.ORG/
CREATE A [PROGRAM]

- % Definir a relação pai
- pai(joao, maria).
- pai(joao, ana).
- % Definir a relação mãe
- mae(maria, josefina).
- mae(ana, julia).
- % Definir a relação filho
- filho(X,Y) :- pai(Y, X); mae(Y, X).

- Query (consulta)
- ?- pai(joao, maria).
- ?- pai(joao, X).
- ?- mae(Quem, julia).

8 CONTINUANDO COM FAMÍLIA

- Baseado no arquivo familia2.pl
- Defina os homens e mulheres do slide anterior, usando exemplos do slide 6.
- Defina as regras do slide 6 (à direita)
- Faça as consultas que mostrem quem são irmãos
- Ex. filha(maria, Quem).
- irma(julia, X).

- Defina avou e avoh
- Defina primos
- OBS:

  ; OU

  , E

  not

  :- se

9 REGRAS RECURSIVAS

- Recursão infinita, deve ser evitada:
- descendente (Z,X) :- genitor (X,Y) , descendente (Z,Y) .
- As duas regras em conjunto abaixo resolvem a recursão com a definição elementar da primeira regra:
- descendente (Z,X) :- genitor (X, Z) .
- descendente (Z,X) :- genitor (X,Y), descendente (Z,Y) .
- Redefina avou e avoh usando recursão
- Bisavou e bisavoh

```
/* Programa Prolog sobre relações familiares. */
genitor( pam, bob). % fato
genitor( tom, bob). % fato
genitor( tom, liz). % fato
genitor( bob, ann). % fato
genitor( bob, pat). % fato
genitor( pat, jim). % fato
mulher(pam). % fato
mulher(liz). % fato
mulher(pat). % fato
mulher(ann). % fato
homem(tom). % fato
homem(bob). % fato
homem(jim). % fato
prole(Y,X) :- genitor(X,Y). % regra
mae(X,Y) :- genitor(X,Y), mulher(X). % regra
avos(X,Z) :- genitor(X,Y), genitor(Y,Z). % regra
irma(X,Y) :- genitor(Z,X), genitor(Z,Y), mulher(X). % regra
descendente(X,Z) :- genitor(X,Z). % regra
descendente(X,Z) :- genitor(X,Y), descendente(Y,Z). % regra recursiva
```

11 MAIS RECURSÃO

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

```
?- fatorial(0,1).
?- fatorial(3,6).
?- fatorial(5,F).
```

12 TORRES DE HANOI

```
move(1,X,Y,_) :-
    write('Move top disk from '),
    write(X),
    write(' to '),
    write(Y),
    nl.
move(N,X,Y,Z) :-
    N>1,
    M is N-1,
    move(M,X,Z,Y),
    move(1,X,Y,_),
    move(M,Z,Y,X).
```

?- move(3,left,right,center).

13 LISTAS

- Os elementos contidos em uma lista devem ser separados por vírgulas, e precisam estar entre colchetes.
- Por exemplo, uma lista pode conter os nomes dos indivíduos do exemplo da seção anterior, esta lista seria definida como:
- [pam, liz, pat, ann, tom, bob, jim]

- Pode-se especificar que um elemento de uma lista é também uma lista, assim, pode-se representar listas como:
- Hobbies1 = [tenis, musica].
- Hobbies2 = [sky, comida].
- Lista = [ann, Hobbies1, tom, Hobbies2].

14 LISTAS

- [mia, vincent, jules, yolanda]
- [mia, robber(honey_bunny), X, 2, mia]
- [ ]
- [mia, [vincent, jules],
  [butch, girlfriend(butch)]]
- [[ ], dead(zed), [2, [b, chopper]], [ ], Z,
  [2, [b, chopper]]]

- Existem dois tipos de listas,
  as listas vazias e as não vazias.
- Uma lista vazia é representada por [ ].
- Listas não vazias podem ser
  divididas em duas partes, são elas:
  - cabeça - corresponde
    ao primeiro elemento da lista;
  - cauda - corresponde
    aos elementos restantes da lista.

15 LISTAS

- Por exemplo, para a lista
- [pam, liz, pat, ann, tom, bob, jim]
- pam é a cabeça, enquanto [liz, pat, ann, tom, bob, jim] é a cauda.
- Observe que a cauda é uma nova lista, que por sua vez também possui cabeça e cauda.
- Assim, pode-se dizer que o último elemento de uma lista possui uma cauda vazia (uma lista vazia).

- É possível separar as partes de uma lista utilizando uma barra vertical, assim, pode-se escrever
- Lista = [cabeça | cauda].
- Com isso, é possível determinar as seguintes listas:
- [a | b, c] = [a, b, c]

16 UNIFICAÇÃO EM LISTAS

- Faça as seguintes consultas (*queries*)
- [mesa] = [X|Y]
- [a,b,c,d] = [X,Y|Z]
- [[ana,Y] | Z] = [[X,foi] , [ao,cinema]]
- [ano, bissexto] =[X,Y | Z]
- [ano, bissexto] = [X,Y,Z]

- https://swish.swi-prolog.org/

17 PERTINÊNCIA A UMA LISTA

- Sempre que um programa recursivo é definido, deve-se procurar pelas condições limites (ou condições de parada) e pelo caso(s) recursivo(s):
- pertence(Elemento,[Elemento|Cauda]).
  pertence(Elemento,[Cabeca|Cauda]) :-
  pertence(Elemento,Cauda).
- Após a definição do programa, é possível interrogá-lo.
- Por exemplo, ?- pertence(a,[a,b,c]).

- O último elemento de uma lista que tenha somente um elemento é o próprio elemento
- ultimo(Elemento, [Elemento]).
- O último elemento de uma lista que tenha mais de um elemento é o último elemento da cauda
- ultimo(Elemento, [Cabeca|Cauda]) :-
  ultimo(Elemento,Cauda).
- Programa completo:
- ultimo(Elemento, [Elemento]).
- ultimo(Elemento, [Cabeca|Cauda]) :-
  ultimo(Elemento,Cauda).

Prolog: listas (rodrigorgs.github.io)

18 PERTINÊNCIA A UMA LISTA

- Essa sintaxe é útil em consultas quando queremos decompor uma lista em cabeça e cauda.
- `?- [Head|Tail] = [mia, vincent, jules, yolanda].`
- `Head = mia`
- `Tail = [vincent,jules,yolanda]`
- `yes`
- Equivalente: `[X|Y] = [mia, vincent, jules, yolanda].`

- Obter os dois primeiros elementos da lista
- `?- [X,Y | W] = [[], dead(zed), [2, [b, chopper]], [], Z].`
- Obter o segundo e o quarto elementos
- `?- [_,X,_,Y|_] = [[], dead(zed), [2, [b, chopper]], [], Z].`
- Exemplo mais avançado:
- `?- [_,_,[_|X]|_] =`
- `[[], dead(zed), [2, [b, chopper]], [], Z, [2, [b, chopper]]].`

Prolog: listas (rodrigorgs.github.io)

19 VARIÁVEL ANÔNIMA

- Quando uma variável aparece em uma única cláusula, não é necessário utilizar um nome para ela
- Utiliza-se a variável anônima, que é escrita com um simples caracter '_'.
- Exemplo
- temfilho(X) :- progenitor(X,F).
- Para definir temfilho, não é necessário o nome do filho(a)
- temfilho(X) :- progenitor(X,_).

- a variável anônima, _ (somente _), tem a seguinte peculiaridade: cada ocorrência dela representa uma variável distinta. Exemplo de consulta:
- *%% Quais são os professores (não importa a disciplina e a turma)?*
- *turma(_, _, P).*
- *%% Note que o primeiro e o segundo argumentos de turma são variáveis diferentes, que podem assumir valores diferentes. A consulta equivale a*
- *turma(_X, _Y, P).*

20 COMPARAÇÃO ENTRE TERMOS

Operadores Relacionais	
X = Y	X unifica com Y que é verdadeiro quando dois termos são o mesmo. Entretanto, se um dos termos é uma variável, o operador = causa a instanciação da variável porque o operador causa unificação
X \= Y	X não unifica com Y que é o complemento de X=Y
X == Y	X é literalmente igual a Y (igualdade literal), que é verdadeiro se os termos X e Y são idênticos, ou seja, eles têm a mesma estrutura e todos os componentes correspondentes são os mesmos, incluindo o nome das variáveis
X \== Y	X não é literalmente igual a Y que é o complemento de X==Y
X @< Y	X precede Y
X @> Y	Y precede X
X @=< Y	X precede ou é igual a Y
X @>= Y	Y precede ou é igual a X

21 COMPARAÇÃO ENTRE TERMOS

- Verificação de termos não idênticos:
- predicado \==
- Sintaxe: Termo1 \== Termo2
- Retorna sucesso se os termos Termo1 e Termo2 não são idênticos:
- ?- nome \== nome.
- fail.
- ?- X \== Y.
- true.

- Outras comparações:
- – Sintaxe: Termo1 <op> Termo2
- – <op> pode ser: @>=, @>, @=<, @<
- – Compara os termos sem calcular:
- ?- 2+1 < 1+3.
- true.
- ?- 2+1 @< 1+3.
- fail.
- ?- 2 @< 1+3.
- true

22 ARITMÉTICA EM PROLOG

- Faça as consultas
- ?- X is 1+3.
- X = 4
- ?- X is 4*3+10/2.
- X = 17
- ?- X is abs((15-30)//2).
- X = 7

is é atribuição

- Outros operadores:
- 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)

23 EXEMPLOS

- Somatório
- soma([ ],0).
- soma([Elem|Cauda], S) :- soma(Cauda,S1),
- S is S1 + Elem.
- Consulta:
- ?- soma([1,2,3,4,5,6], S).

- Contar o número de elementos de uma lista
- conta([ ],0).
- conta([_| Cauda], N) :-
- conta(Cauda, N1),
- N is N1 + 1.
- ?- conta([1,2,3,4,5,6],C)

CONTROLE DO "BACKTRACKING" OU RETROCESSO

- O backtracking, ou retrocesso, é um mecanismo existente no Prolog que procura fatos ou regras adicionais que satisfaçam um objetivo em uma questão com vários objetivos (subgoals) quando a instanciação corrente falhou.
- O controle do backtracking é importante pois é uma das maneiras de provocar repetição, sendo outra maneira o uso da recursão.
- Existem dois predicados fornecidos pelo Prolog para o controle do backtracking: a falha (fail) e o corte (cut).

25 CUT (!) / FAIL

- fail é a instrução que força uma falha.
- Ou seja, ao encontrar-se uma instrução fail, sempre é feito o backtracking.
- fail, sozinho, raramente é usado, e seu uso principal é a combinação cut/fail.
- irmao(A,A) :- ! , fail. % esta instrução força uma pessoa não ser irmão dela mesma
- irmao(A, B) :- filho(A, M), mae(M, B).
- irmao(A, B) :- filho(A, P), pai(P, B).

- Agora, a partir dessas duas linhas, podemos entender que essas duas afirmações são mutuamente exclusivas, portanto, quando uma é verdadeira, a outra deve ser falsa.
- Nesses casos, podemos usar o corte (cut).
- Também podemos definir um predicado em que usamos os dois casos usando disjunção (lógica OR).
- Assim, quando o primeiro satisfizer, ele não verificará o segundo, caso contrário, verificará a segunda declaração.

26 FALHA (FAIL)

- O predicado fail (chamado de falha em português sempre retorna uma falha na unificação.
- Com isso, força o Prolog a fazer o backtracking, repetindo os predicados anteriores ao predicado fail.
- Verifique o funcionamento do predicado fail. Retire o fail e verifique o resultado.
- Como fazer para que a regra "escreve_todos" resulte verdadeira, após terminar de escrever toda a lista? (i. e., como fazer para termos "True" em vez de "False" ao final das execução?)

- `aluno(tipo_nome). escreve_todos.`
- `aluno(benedicte).`
- `aluno(alice).`
- `aluno(marcelo).`
- `aluno(andre).`
- `aluno(roberto).`
- `escreve_todos :- aluno(Nome), write(Nome), nl, fail.`

27 CORTE (CUT)

- O Prolog, na busca de uma solução, sempre faz automaticamente o backtracking após uma tentativa de unificação de um predicado que não resultou positiva.
- O corte é um predicado existente em todas as implementações do Prolog, usado para evitar o backtracking.
- Ao usar o corte, o Prolog "esquece" todos os marcadores de backtracking que foram feitos ao se tentar satisfazer o sub-goal atual e não mais procura por outras possíveis unificações caso ocorra uma falha.
- Com isso, o corte pode tornar um programa mais rápido, pois evita que o Prolog explore alternativas que, sabe-se de antemão, não irão contribuir para a solução do problema
- e ainda permite a economia de memória, pois elimina marcadores de backtracking.
- O símbolo do corte é uma exclamação "!".

28 CORTE (CUT)

- Apesar de ser usualmente comparado com o break existente em linguagens como Pascal e Linguagem C,
- o funcionamento do corte não é exatamente igual ao dele, existindo muitas diferenças entre os dois.
- Por isso, o aluno não deve pensar no corte simplesmente como um break.
- O corte pode ser visto como um diodo, que só permite a passagem da busca da esquerda para direita e não da direita para esquerda.
- Assim, as unificações feitas pelo Prolog à esquerda de um corte não podem ser desfeitas após a passagem da busca do Prolog para o lado direito do corte.

29 CORTE (CUT)

- Por exemplo,
- homem(joao).
- homem(jose).
- homem(pedro).
- mulher(maria).
- mulher(ana).
- mulher(paula).
- pessoas(P) :- homem(P), ! ; mulher(P), !.

- Experimente retirar cada corte (!) por vez e examine o resultado
- No caso
- pessoas(P) :- homem(P) ; mulher(P), ! .
- o Prolog tenta satisfazer mulher(P) e não realiza o backtracking para as demais respostas.

30 CORTES "VERDES"

- `f(X,0) :- X < 3,!.`
- `f(X,2) :- 3 <= X, X < 6,!.`
- `f(X,4) :- 6 <= X,!.`
- Este exemplo define uma função em partes:

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

- Use a questão
- ?- f(2,Resposta), Resposta < 2
- e verifique o resultado.
- Retire os cortes e verifique novamente.
- Com o uso do corte, eliminamos a passagem do programa por regras que sabemos de antemão que não serão úteis para resolver o problema.

31 CORTES "VERMELHOS"

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

- Use a questão
- ?- f(2,Resposta), Resposta < 2
- e verifique o resultado.
- Retire os cortes e verifique novamente.

32 CUT + FAIL

- Esse método lembra o repeat-until das linguagens procedimentais, porém é mais poderoso, pois o until não é apenas uma condição, mas um predicado que pode inclusive realizar operações.
-
```
aluno(benedito).
aluno(alice). aluno(marcelo).
aluno(andre). aluno(roberto).
```

-
```
escreve_ate(Nome_final) :-
aluno(Nome), write(Nome), nl,
controle_do_corte(Nome,
Nome_final), !.
```

- A regra abaixo falha provocando o backtracking, até que o nome corrente da busca seja igual ao final, dado pelo usuário. Assim, ela substitui o predicado fail neste programa.
-
```
controle_do_corte(Nome_2,
Nome_final_2) :-
      Nome_2 = Nome_final_2.
```

33 MÁXIMO DENTRE DOIS

- max(X,Y,X) :- X >= Y , ! . % se X maior
- max(X,Y,Y) :- X < Y . % se Y maior
- max_deles(X, Y, Max) :-
  - X >= Y , ! ,
  - Max = X;
  - Max = Y.

- Usando IF/THEN/ELSE
- if A then B else C é escrito como (A -> B ; C)
- max(X, Y, Max) :-
- (X =< Y
- -> Max = Y
- ; Max = X
-).