Programação com Prolog

Utilizando linguagens *dedutivas* para problemas simples

Antonio Coelho - 9360691

Fatorial

Nosso primeiro objetivo é implementar em prolog uma “calculadora” de números fatoriais.

Dado um número X, o problema deve calcular X! - o resultado como mostrado na imagem ao lado

$$n! = \prod_{k=1}^{n} k \qquad \forall n \in \mathbb{N}$$

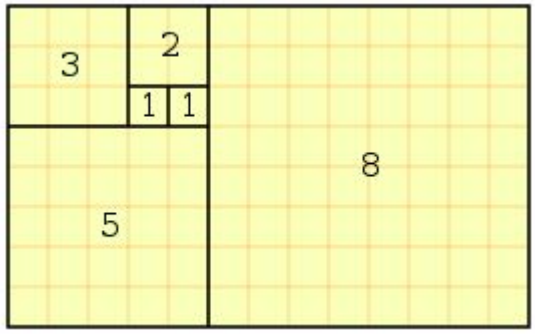
$$5! = 1 \times 2 \times 3 \times 4 \times 5 = 120$$

Fibonacci

Nosso segundo objetivo é implementar em prolog uma “calculadora” de números fibonacci.

Dado um número X, o problema deve calcular F(X-1) + F(X-2) - o resultado como mostrado na imagem ao lado

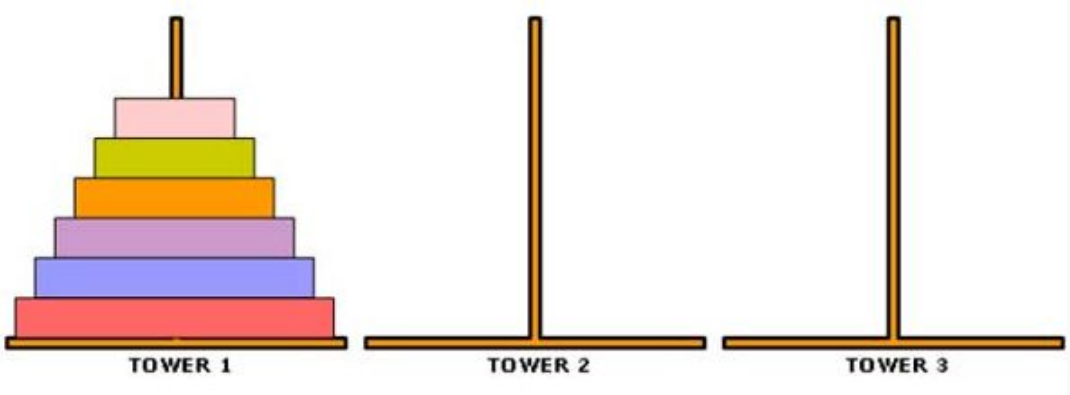
$$F_n = F_{n-1} + F_{n-2},$$



Torre de Hanoi

Nosso terceiro objetivo é implementar em prolog um “resolvedor” de torre de Hanoi.

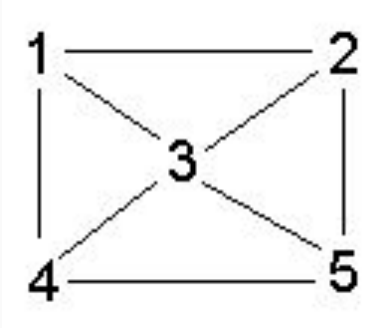
Dado um número N de discos, o problema é transportar todos os discos para torre 3 sem deixar um disco maior, sobre um disco menor

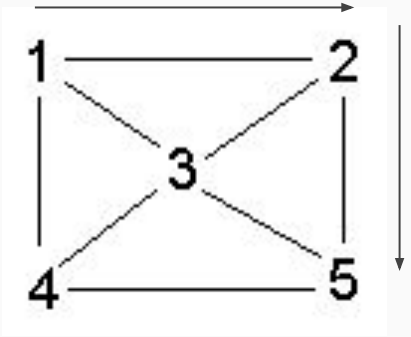


Caminho em Grafo

Nosso quarto objetivo é implementar em prolog um “resolvedor” de caminhos em grafos.

Dado um grafo G(V,E), devemos calcular o caminho para chegar de V1 até VN

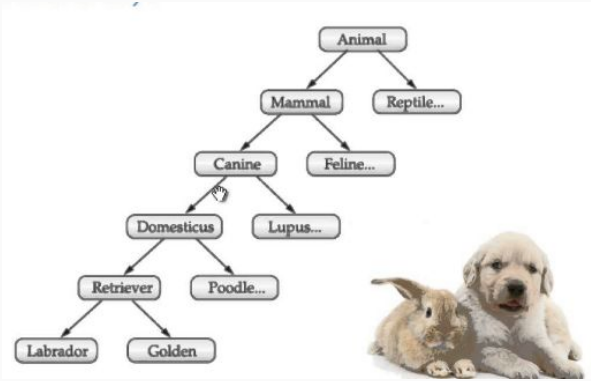




Sistema Especialista

Nosso quinto objetivo é implementar em prolog um pequeno sistema especialista para identificação de animais.

Dada perguntas e resposta de “SIM” e “NAO”, devemos descobrir qual a raça de um determinado animal



Agenda

1. Introdução

1. História do Prolog
2. Paradigma declarativo/lógica
3. Conceitos básicos lógica
4. Conceitos básicos Prolog
5. *Hello world* em Prolog

2. Montando o Fatorial

1. Utilizando regras recursivas
2. Lógica do Fatorial em Prolog

Agenda

3. **Montando o Fibonacci**

1. Lógica do Fibonacci

4. **Montando a Torre Hanoi**

1. Lógica do Hanoi em Prolog

5. **Montando Grafos**

1. Lógica do Grafos

6. **Montando Sistema especialista**

1. Lógica do Sistema especialista

Introdução

História do Prolog

Criada em 1972 por Alain Colmerauer e Robert Kowalski

1) Pertence ao paradigma de lógica: *declarativa*, utiliza B.C e fatos
2) Motor de inferência utiliza *cláusulas de Horn* e *regra resolução*

Alain Colmerauer

Linguagem Declarativa

- Descreve relação entre variáveis através de regras de inferência
- Deixa explícito o que deve ser executado | não “**como**” deve ser feito
- Toda a execução do “como” é implementada pelo compilador/interpretador da linguagem

Cláusula de Horn | Regra de Resolução

- Escritas no formato: $(P_1 \wedge P_2 \wedge \cdots P_k) \rightarrow Q$.
- Equivalente a: $\neg p \vee \neg q \vee \cdots \vee \neg t \vee u$

- Grande vantagem para motores de inferência (“horn + horn = horn”)

Cláusula de Horn | Regra de Resolução

- Através do princípio da resolução, as cláusulas de horn mantém consistência (bom para linguagens (1) lógicas e (2) declarativas)
- (1) O motor de inferência utiliza um padrão para resolução de *fatos* - torna o processo mais rápido
- (2) Define um padrão de escrita (sintaxe) - torna a “compilação” mais fácil e por consequência o processamento
- Utilização das cláusulas de Horn em Prolog: `u :- p, q, … , t`

Conceitos básicos Prolog

Átomos | **Números**: pedro, ‘exemplo’, 123, 3.80. São constantes de texto e número

Variáveis: X, _variavel123. São incógnitas, após descobertas, transformam em constantes

Fatos: professora(sara), país(Brasil, America). São predicados, definindo verdades que serão adicionadas a B.C

Se lê: Brasil é um país da América; Sara é professora

Conceitos básicos Prolog

Regras: filha(julinho) :- mae(roberta). São cláusulas que definem comportamentos verdadeiros que predicados podem assumir

Se lê: “filho julinho é verdadeiro se mãe for roberta

Disjunção (OU) e **conjunção** (E): Escritas respectivamente como “,” e “;”

Hello world em Prolog

1) Precisamos criar uma B.C que guardará nossos predicados e regras
2) Crie um arquivo base_conhecimento.pl
3) Adicione o texto abaixo. Não esqueça dos **pontos finais**

```
woman(maria).

woman(roberta).

sentenca_verdadeira.

eh_advogada(roberta).
```

Hello world em Prolog

Assim que abrir o console do Prolog, comece a executar queries:

```
?- mulher(maria).
true.

?-
```

```
?- mulher(carla).
false.

?-
```

```
?- eh_advogada(roberta).
true.

?- mulher(roberta).
true.

?- eh_advogada(X);mulher(X).
X = roberta
```

Montando o Fatorial

Utilizando regras recursivas

Em Prolog, regras recursivas precisam ser definidas por um **caso base**

E.g

%Define que quando recursão chegar a 0, seu valor será 1

recursao(0, 1).

Utilizando regras recursivas

```
%Caso base - soma de 1 é 1
soma_numeros(1, 1).

%Definimos a função recursiva - um número e o resultado de todos os outros anteriores somados
soma_numeros(NUMERO, RESULTADO) :-
  %Verificamos se o numero é maior que 0
  NUMERO > 0,
  %Definimos que o proximo numero, é o numero menos 1
  NUMERO1 is NUMERO-1,
  %Fazemos a recursao para somar os numeros anterior de NUMERO na variável RESULTADO_TEMP
  soma_numeros(NUMERO1, RESULTADO_TEMP),
  %Definimos o retorno. Nosso RESULTADO será a soma de NUMERO + RESULTADO_TEMP (da recursao)
  RESULTADO is NUMERO + RESULTADO_TEMP.
```

Utilizando regras recursivas

```
%Caso base - soma de 1 é 1
soma_numeros(1, 1).

%Definimos a função recursiva - um número e o resultado de todos os outros anteriores somados
soma_numeros(NUMERO, RESULTADO) :-
  %Verificamos se o numero é maior que 0
  NUMERO > 0,
  %Definimos que o proximo numero, é o numero menos 1
  NUMERO1 is NUMERO-1,
  %Fazemos a recursao para somar os numeros anterior de NUMERO na variável RESULTADO_TEMP
  soma_numeros(NUMERO1, RESULTADO_TEMP),
  %Definimos o retorno. Nosso RESULTADO será a soma de NUMERO + RESULTADO_TEMP (da recursao)
  RESULTADO is NUMERO + RESULTADO_TEMP.
```

Criamos um caso base

Utilizando regras recursivas

```
%Caso base - soma de 1 é 1
soma_numeros(1, 1).

%Definimos a função recursiva - um número e o resultado de todos os outros anteriores somados
soma_numeros(NUMERO, RESULTADO) :-
  %Verificamos se o numero é maior que 0
  NUMERO > 0,
  %Definimos que o proximo numero, é o numero menos 1
  NUMERO1 is NUMERO-1,
  %Fazemos a recursao para somar os numeros anterior de NUMERO na variável RESULTADO_TEMP
  soma_numeros(NUMERO1, RESULTADO_TEMP),
  %Definimos o retorno. Nosso RESULTADO será a soma de NUMERO + RESULTADO_TEMP (da recursao)
  RESULTADO is NUMERO + RESULTADO_TEMP.
```

Definimos a função recursiva

Utilizando regras recursivas

```
%Caso base - soma de 1 é 1
soma_numeros(1, 1).

%Definimos a função recursiva - um número e o resultado de todos os outros anteriores somados
soma_numeros(NUMERO, RESULTADO) :-
  %Verificamos se o numero é maior que 0
  NUMERO > 0,
  %Definimos que o proximo numero, é o numero menos 1
  NUMERO1 is NUMERO-1,
  %Fazemos a recursao para somar os numeros anterior de NUMERO na variável RESULTADO_TEMP
  soma_numeros(NUMERO1, RESULTADO_TEMP),
  %Definimos o retorno. Nosso RESULTADO será a soma de NUMERO + RESULTADO_TEMP (da recursao)
  RESULTADO is NUMERO + RESULTADO_TEMP.
```

Fazemos validações

Utilizando regras recursivas

```
%Caso base - soma de 1 é 1
soma_numeros(1, 1).

%Definimos a função recursiva - um número e o resultado de todos os outros anteriores somados
soma_numeros(NUMERO, RESULTADO) :-
  %Verificamos se o numero é maior que 0
  NUMERO > 0,
  %Definimos que o proximo numero, é o numero menos 1
  NUMERO1 is NUMERO-1,
  %Fazemos a recursão para somar os numeros anterior de NUMERO na variável RESULTADO_TEMP
  soma_numeros(NUMERO1, RESULTADO_TEMP),
  %Definimos o retorno. Nosso RESULTADO será a soma de NUMERO + RESULTADO_TEMP (da recursao)
  RESULTADO is NUMERO + RESULTADO_TEMP.
```

Definimos novas variáveis

Utilizando regras recursivas

```
%Caso base - soma de 1 é 1
soma_numeros(1, 1).

%Definimos a função recursiva - um número e o resultado de todos os outros anteriores somados
soma_numeros(NUMERO, RESULTADO) :-
  %Verificamos se o numero é maior que 0
  NUMERO > 0,
  %Definimos que o proximo numero, é o numero menos 1
  NUMERO1 is NUMERO-1,
  %Fazemos a recursao para somar os numeros anterior de NUMERO na variável RESULTADO_TEMP
  soma_numeros(NUMERO1, RESULTADO_TEMP),
  %Definimos o retorno. Nosso RESULTADO será a soma de NUMERO + RESULTADO_TEMP (da recursao)
  RESULTADO is NUMERO + RESULTADO_TEMP.
```

Criamos a “recursão” em sí

Utilizando regras recursivas

```
%Caso base - soma de 1 é 1
soma_numeros(1, 1).

%Definimos a função recursiva - um número e o resultado de todos os outros anteriores somados
soma_numeros(NUMERO, RESULTADO) :-
  %Verificamos se o numero é maior que 0
  NUMERO > 0,
  %Definimos que o proximo numero, é o numero menos 1
  NUMERO1 is NUMERO-1,
  %Fazemos a recursao para somar os numeros anterior de NUMERO na variável RESULTADO_TEMP
  soma_numeros(NUMERO1, RESULTADO_TEMP),
  %Definimos o retorno. Nosso RESULTADO será a soma de NUMERO + RESULTADO_TEMP (da recursao)
  RESULTADO is NUMERO + RESULTADO_TEMP.
```

Definimos o retorno

Lógica do Fatorial em Prolog

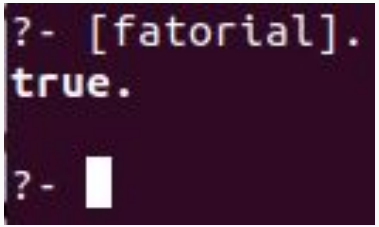
1. Precisamos criar um caso base
2. Definirmos a função de recursão
3. Quais validações precisamos fazer?
4. Como devemos definir novas variáveis?
5. Como deve ser nosso retorno?

Lógica do Fatorial em Prolog

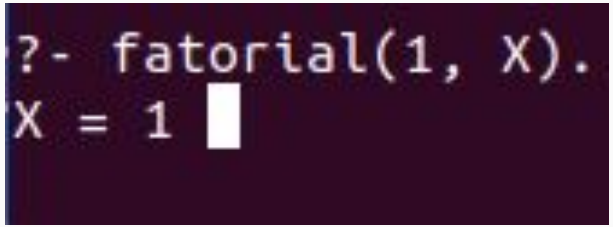
```
%Define fatorial de 0 como 1
fatorial(0,1).
%Podemos também definir fatorial de 1 como 1
fatorial(1,1).

%Cria metodo de fatorial | Recebe um numero e retorna o F (fatorial de N)
fatorial(N,F) :-
  %Caso N > 0 seja verdade
  N>0,
  %Entao N1 será definido como N-1 (que será nossa nova variável da recursão)
  N1 is N-1,
  %Entao roda o fatorial de N-1 e F-1 (F-1 representa nossa nova variável de resultado)
  fatorial(N1,F1),
  %Entao F será N * F-1 que sairá como resultado da linha acima
  F is N * F1.
```

Lógica do Fatorial em Prolog

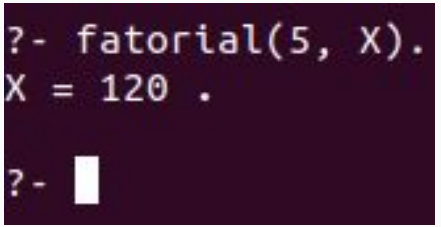
Lemos a B.C



Testamos o caso base



Testamos F(5)



Montando o Fibonacci

Lógica do Fibonacci

```
%Define caso base de fibonacci de 0 = 1 e previne que ocorra o backtracking
fib(0, 1) :- !.

%Define caso base de fibonacci de 1 = 1 e previne que ocorra o backtracking
fib(1, 1) :- !.

%Define fibonacci de N como Resultado
fib(N, Resultado) :-

  %Definimos N1 como N-1
  N1 is N - 1,

  %Realizamos o mesmo procedimento pro N2
  N2 is N - 2,

  %Calculamos\ a recursão do fibonacci de N1 para o resultado n-1
  fib(N1, ResultadoN1),

  %Calculamos a recursão do fibonacci de N2 para o results2 n-2
  fib(N2, ResultadoN2),

  %Retornamos o resultado como a recursão da esquerda (resultado1) com a recursão da direita (resultado2)
  Resultado is ResultadoN1 + ResultadoN2.
```

Lógica do Fibonacci

```
%Define caso base de fibonacci de 0 = 1 e previne que ocorra o backtracking
fib(0, 1) :- !.

%Define caso base de fibonacci de 1 = 1 e previne que ocorra o backtracking
fib(1, 1) :- !.

%Define fibonacci de N como Resultado
fib(N, Resultado) :-

  %Definimos N1 como N-1
  N1 is N - 1,

  %Realizamos o mesmo procedimento pro N2
  N2 is N - 2,

  %Calculamos\ a recursão do fibonacci de N1 para o resultado n-1
  fib(N1, ResultadoN1),

  %Calculamos a recursão do fibonacci de N2 para o results2 n-2
  fib(N2, ResultadoN2),

  %Retornamos o resultado como a recursão da esquerda (resultado1) com a recursão da direita (resultado2)
  Resultado is ResultadoN1 + ResultadoN2.
```

Definimos caso base.
Utilizamos “!” para impedir o backtracking de continuar ocorrendo

Lógica do Fibonacci

```
%Define caso base de fibonacci de 0 = 1 e previne que ocorra o backtracking
fib(0, 1) :- !.

%Define caso base de fibonacci de 1 = 1 e previne que ocorra o backtracking
fib(1, 1) :- !.

%Define fibonacci de N como Resultado
fib(N, Resultado) :-

  %Definimos N1 como N-1
  N1 is N - 1,

  %Realizamos o mesmo procedimento pro N2
  N2 is N - 2,

  %Calculamos\ a recursão do fibonacci de N1 para o resultado n-1
  fib(N1, ResultadoN1),

  %Calculamos a recursão do fibonacci de N2 para o results2 n-2
  fib(N2, ResultadoN2),

  %Retornamos o resultado como a recursão da esquerda (resultado1) com a recursão da direita (resultado2)
  Resultado is ResultadoN1 + ResultadoN2
```

Corpo do método recursivo

Lógica do Fibonacci

```
%Define caso base de fibonacci de 0 = 1 e previne que ocorra o backtracking
fib(0, 1) :- !.

%Define caso base de fibonacci de 1 = 1 e previne que ocorra o backtracking
fib(1, 1) :- !.

%Define fibonacci de N como Resultado
fib(N, Resultado) :-

  %Definimos N1 como N-1
  N1 is N - 1,

  %Realizamos o mesmo procedimento pro N2
  N2 is N - 2,

  %Calculamos\ a recursão do fibonacci de N1 para o resultado n-1
  fib(N1, ResultadoN1),

  %Calculamos a recursão do fibonacci de N2 para o results2 n-2
  fib(N2, ResultadoN2),

  %Retornamos o resultado como a recursão da esquerda (resultado1) com a recursão da direita (resultado2)
  Resultado is ResultadoN1 + ResultadoN2.
```

Definimos nossos dois próximos números

Lógica do Fibonacci

```
%Define caso base de fibonacci de 0 = 1 e previne que ocorra o backtracking
fib(0, 1) :- !.

%Define caso base de fibonacci de 1 = 1 e previne que ocorra o backtracking
fib(1, 1) :- !.

%Define fibonacci de N como Resultado
fib(N, Resultado) :-

  %Definimos N1 como N-1
  N1 is N - 1,

  %Realizamos o mesmo procedimento pro N2
  N2 is N - 2,

  %Calculamos\ a recursão do fibonacci de N1 para o resultado n-1
  fib(N1, ResultadoN1),

  %Calculamos a recursão do fibonacci de N2 para o results2 n-2
  fib(N2, ResultadoN2),

  %Retornamos o resultado como a recursão da esquerda (resultado1) com a recursão da direita (resultado2)
  Resultado is ResultadoN1 + ResultadoN2.
```

Fazemos a chamada recursiva idêntica ao Fatorial

Lógica do Fibonacci

```
%Define caso base de fibonacci de 0 = 1 e previne que ocorra o backtracking
fib(0, 1) :- !.

%Define caso base de fibonacci de 1 = 1 e previne que ocorra o backtracking
fib(1, 1) :- !.

%Define fibonacci de N como Resultado
fib(N, Resultado) :-

  %Definimos N1 como N-1
  N1 is N - 1,

  %Realizamos o mesmo procedimento pro N2
  N2 is N - 2,

  %Calculamos\ a recursão do fibonacci de N1 para o resultado n-1
  fib(N1, ResultadoN1),

  %Calculamos a recursão do fibonacci de N2 para o resultado n-2
  fib(N2, ResultadoN2),

  %Retornamos o resultado como a recursão da esquerda (resultado1) com a recursão da direita (resultado2)
  Resultado is ResultadoN1 + ResultadoN2.
```

Retornamos a soma das recursões

Montando o Torre Hanoi

Lógica Torre de Hanoi

```
%Caso base: Para uma peça (N=1) movimentaremos X para Y. O _ pode ser considera um caracter que não deve ser utilizado
move(1,X,Y,_).

%Definimos N como numero de discos, e X, Y e Z como nossas posições disponíveis
move(N,X,Y,Z) :-
  %Verificamos um N > 1
  N>1,
  %Definimos a recursão como M sendo N-1
  M is N-1,
  %Chamamos metodo recursivo de Hanoi para N-1 para lado esquerdo Z <-> Y
  move(M,X,Z,Y),
  %Chamamos o mesmo metodo de movimentacao porem para Y <-> Z
  move(M,Z,Y,X).
```

Lógica Torre de Hanoi

```
%Caso base: Para uma peça (N=1) movimentaremos X para Y. O _ pode ser considera um caracter que não deve ser utilizado
move(1,X,Y,_).
```

```
%Definimos N como numero de discos, e X, Y e Z como nossas posições disponíveis
move(N,X,Y,Z) :-
  %Verificamos um N > 1
  N>1,
  %Definimos a recursão como M sendo N-1
  M is N-1,
  %Chamamos metodo recursivo de Hanoi para N-1 para lado esquerdo Z <-> Y
  move(M,X,Z,Y),
  %Chamamos o mesmo metodo de movimentacao porem para Y <-> Z
  move(M,Z,Y,X).
```

Definimos o caso base da movimentação de 1 disco Que simplesmente faz a troca de X para Y

Lógica Torre de Hanoi

```
%Caso base: Para uma peça (N=1) movimentaremos X para Y. O _ pode ser considera um caracter que não deve ser utilizado
move(1,X,Y,_).

%Definimos N como numero de discos, e X, Y e Z como nossas posições disponíveis
move(N,X,Y,Z) :-
  %Verificamos um N > 1
  N>1,
  %Definimos a recursão como M sendo N-1
  M is N-1,
  %Chamamos metodo recursivo de Hanoi para N-1 para lado esquerdo Z <-> Y
  move(M,X,Z,Y),
  %Chamamos o mesmo metodo de movimentacao porem para Y <-> Z
  move(M,Z,Y,X).
```

Corpo do método recursivo

Lógica Torre de Hanoi

```
%Caso base: Para uma peça (N=1) movimentaremos X para Y. O _ pode ser considera um caracter que não deve ser utilizado
move(1,X,Y,_).

%Definimos N como numero de discos, e X, Y e Z como nossas posições disponíveis
move(N,X,Y,Z) :-
  %Verificamos um N > 1
  N>1,
  %Definimos a recursão como M sendo N-1
  M is N-1,
  %Chamamos metodo recursivo de Hanoi para N-1 para lado esquerdo Z <-> Y
  move(M,X,Z,Y),
  %Chamamos o mesmo metodo de movimentacao porem para Y <-> Z
  move(M,Z,Y,X).
```

Verificamos N e criamos variáveis pro próximo loop de recursão

Lógica Torre de Hanoi

```
%Caso base: Para uma peça (N=1) movimentaremos X para Y. O _ pode ser considera um caracter que não deve ser utilizado
move(1,X,Y,_).

%Definimos N como numero de discos, e X, Y e Z como nossas posições disponíveis
move(N,X,Y,Z) :-
  %Verificamos um N > 1
  N>1,
  %Definimos a recursão como M sendo N-1
  M is N-1,
  %Chamamos metodo recursivo de Hanoi para N-1 para lado esquerdo Z <-> Y
  move(M,X,Z,Y),
  %Chamamos o mesmo metodo de movimentacao porem para Y <-> Z
  move(M,Z,Y,X).
```

Definimos uma solução em “largura”
Movemos de Z <-> Y e de X <-> Z

Aumenta nosso espaço de busca

Montando o Grafo

Lógica do Grafo

```
%Definimos os nós do grafo
edge(1,2). edge(1,4). edge(1,3). edge(2,3). edge(2,5). edge(3,4). edge(3,5). edge(4,5).

%Definimos vertices vizinhos caso estes tenha uma "aresta" em comum
connected(X,Y) :- edge(X,Y) ; edge(Y,X).

%Metodo recursivo para calcular o caminho entre A e B
path(A,B,Path) :-
  %O caminho de A até B | Guardamos todos as arestas de A e uma lista ([]) e Q como nossa fila de testes
  travel(A,B,[A],Q),
  %Apos calcularmos a distancia de A até B, invertemos a fila Q em Path
  reverse(Q,Path).

%Definimos travel de A, B
%Criamos o P que representa os nos restante até B. Também teremos a lista B | P que separada a cauda da lista de B
travel(A,B,P,[B|P]) :-
  %Retornamos caso existe uma conexão entre A e B
  connected(A,B).

%Mesmo metodo travel porem definido como nó visitado e o caminho atual
travel(A,B,Visited,Path) :-
  %Verificamos se existe conexão entre A e C
  connected(A,C),
  %Verificamos se C é diferente de B | simbolo \== representa diferença
  C \== B,
  %Vemos se C não foi visitado ainda, através da sintaxe do "member" e do NOT (\+)
  \+member(C,Visited),
  %Por ultimo, caso nao tenha sido visitado, calculamos a viagem de C até B tirando C da lista de visitados e mantando o path
  travel(C,B,[C|Visited],Path).
```

Lógica do Grafo

```
%Definimos os nós do grafo
edge(1,2). edge(1,4). edge(1,3). edge(2,3). edge(2,5). edge(3,4). edge(3,5). edge(4,5).

%Definimos vertices vizinhos caso estes tenha uma "aresta" em comum
connected(X,Y) :- edge(X,Y) ; edge(Y,X).

%Metodo recursivo para calcular o caminho entre A e B
path(A,B,Path) :-
  %O caminho de A até B | Guardamos todos as arestas de A e uma lista ([]) e Q como nossa fila de testes
  travel(A,B,[A],Q),
  %Apos calcularmos a distancia de A até B, invertemos a fila Q em Path
  reverse(Q,Path).
```

Criamos o grafo a partir das arestas

Lógica do Grafo

```
%Definimos os nós do grafo
edge(1,2). edge(1,4). edge(1,3). edge(2,3). edge(2,5). edge(3,4). edge(3,5). edge(4,5).

%Definimos vertices vizinhos caso estes tenha uma "aresta" em comum
connected(X,Y) :- edge(X,Y) ; edge(Y,X).

%Metodo recursivo para calcular o caminho entre A e B
path(A,B,Path) :-
  %O caminho de A até B | Guardamos todos as arestas de A e uma lista ([]) e Q como nossa fila de testes
  travel(A,B,[A],Q),
  %Apos calcularmos a distancia de A até B, invertemos a fila Q em Path
  reverse(Q,Path).
```

Definimos o que significa dois vértices estarem conectados

Lógica do Grafo

```
%Definimos os nós do grafo
edge(1,2). edge(1,4). edge(1,3). edge(2,3). edge(2,5). edge(3,4). edge(3,5). edge(4,5).

%Definimos vertices vizinhos caso estes tenha uma "aresta" em comum
connected(X,Y) :- edge(X,Y) ; edge(Y,X).

%Metodo recursivo para calcular o caminho entre A e B
path(A,B,Path) :-
  %O caminho de A até B | Guardamos todos as arestas de A e uma lista ([]) e Q como nossa fila de testes
  travel(A,B,[A],Q),
  %Apos calcularmos a distancia de A até B, invertemos a fila Q em Path
  reverse(Q,Path).
```

Definimos um caminho como uma viagem de A -> B, usando todos os vértices de A em fila e um Path Q

Lógica do Grafo

```
%Definimos travel de A, B
%Criamos o P que representa os nos restante até B. Também teremos a lista B | P que separada a cauda da lista de B
travel(A,B,P,[B|P]) :-
  %Retornamos caso existe uma conexão entre A e B
  connected(A,B).

%Mesmo metodo travel porem definido como nó visitado e o caminho atual
travel(A,B,Visited,Path) :-
  %Verificamos se existe conexão entre A e C
  connected(A,C),
  %Verificamos se C é diferente de B | simbolo \== representa diferença
  C \== B,
  %Vemos se C não foi visitado ainda, através da sintaxe do "member" e do NOT (\+)
  \+member(C,Visited),
  %Por ultimo, caso nao tenha sido visitado, calculamos a viagem de C até B tirando C da lista de visitados e mantando o path
  travel(C,B,[C|Visited],Path).
```

Definimos a viagem de encontrar vértices conectados entre A e B. Esse sendo nosso caso base

Lógica do Grafo

```
%Definimos travel de A, B
%Criamos o P que representa os nos restante até B. Também teremos a lista B | P que separada a cauda da lista de B
travel(A,B,P,[B|P]) :-
  %Retornamos caso existe uma conexão entre A e B
  connected(A,B).

%Mesmo metodo travel porem definido como nó visitado e o caminho atual
travel(A,B,Visited,Path) :-
  %Verificamos se existe conexão entre A e C
  connected(A,C),
  %Verificamos se C é diferente de B | simbolo \== representa diferença
  C \== B,
  %Vemos se C não foi visitado ainda, através da sintaxe do "member" e do NOT (\+)
  \+member(C,Visited),
  %Por ultimo, caso nao tenha sido visitado, calculamos a viagem de C até B tirando C da lista de visitados e mantando o path
  travel(C,B,[C|Visited],Path).
```

Com exceção do caso base, precisamos verificar de A-> C, se C não é B, se C ainda não foi visitado e por último visitamos C

Lógica do Grafo

```
%Definimos travel de A, B
%Criamos o P que representa os nos restante até B. Também teremos a lista B | P que separada a cauda da lista de B
travel(A,B,P,[B|P]) :-
  %Retornamos caso existe uma conexão entre A e B
  connected(A,B).

%Mesmo metodo travel porem definido como nó visitado e o caminho atual
travel(A,B,Visited,Path) :-
  %Verificamos se existe conexão entre A e C
  connected(A,C),
  %Verificamos se C é diferente de B | simbolo \== representa diferença
  C \== B,
  %Vemos se C não foi visitado ainda, através da sintaxe do "member" e do NOT (\+)
  \+member(C,Visited),
  %Por ultimo, caso nao tenha sido visitado, calculamos a viagem de C até B tirando C da lista de visitados e mantando o path
  travel(C,B,[C|Visited],Path).
```

Verificamos se A e C se conectam

Lógica do Grafo

```
%Definimos travel de A, B
%Criamos o P que representa os nos restante até B. Também teremos a lista B | P que separada a cauda da lista de B
travel(A,B,P,[B|P]) :-
  %Retornamos caso existe uma conexão entre A e B
  connected(A,B).

%Mesmo metodo travel porem definido como nó visitado e o caminho atual
travel(A,B,Visited,Path) :-
  %Verificamos se existe conexão entre A e C
  connected(A,C),
  %Verificamos se C é diferente de B | simbolo \== representa diferença
  C \== B,
  %Vemos se C não foi visitado ainda, através da sintaxe do "member" e do NOT (\+)
  \+member(C,Visited),
  %Por ultimo, caso nao tenha sido visitado, calculamos a viagem de C até B tirando C da lista de visitados e mantando o path
  travel(C,B,[C|Visited],Path).
```

Verificamos se C não é B

Lógica do Grafo

```
%Definimos travel de A, B
%Criamos o P que representa os nos restante até B. Também teremos a lista B | P que separada a cauda da lista de B
travel(A,B,P,[B|P]) :-
  %Retornamos caso existe uma conexão entre A e B
  connected(A,B).

%Mesmo metodo travel porem definido como nó visitado e o caminho atual
travel(A,B,Visited,Path) :-
  %Verificamos se existe conexão entre A e C
  connected(A,C),
  %Verificamos se C é diferente de B | simbolo \== representa diferença
  C \== B,
  %Vemos se C não foi visitado ainda, através da sintaxe do "member" e do NOT (\+)
  \+member(C,Visited),
  %Por ultimo, caso nao tenha sido visitado, calculamos a viagem de C até B tirando C da lista de visitados e mantando o path
  travel(C,B,[C|Visited],Path).
```

Vemos se C não está listado na fila de nós visitados

Lógica do Grafo

```
%Definimos travel de A, B
%Criamos o P que representa os nos restante até B. Também teremos a lista B | P que separada a cauda da lista de B
travel(A,B,P,[B|P]) :-
  %Retornamos caso existe uma conexão entre A e B
  connected(A,B).

%Mesmo metodo travel porem definido como nó visitado e o caminho atual
travel(A,B,Visited,Path) :-
  %Verificamos se existe conexão entre A e C
  connected(A,C),
  %Verificamos se C é diferente de B | simbolo \== representa diferença
  C \== B,
  %Vemos se C não foi visitado ainda, através da sintaxe do "member" e do NOT (\+)
  \+member(C,Visited),
  %Por ultimo, caso nao tenha sido visitado, calculamos a viagem de C até B tirando C da lista de visitados e mantando o path
  travel(C,B,[C|Visited],Path).
```

Caso todos sejam verdadeiros, calculamos a viagem de C até B

Montando o Sistema especialista

Lógica do Sistema especialista

```
animal(cachorro)  :- is_true('tem pelo'), is_true('late').
animal(gato)  :- is_true('tem pelo'), is_true('mia').
animal(pato) :- is_true('tem penas'), is_true('quack').

raca(husky) :- animal(cachorro), is_true('grande').
raca(salsicha) :- animal(cachorro), \+is_true('grande').
raca(persa) :- animal(gato), is_true('pequeno'), \+is_true('peludo').
raca(mainecon) :- animal(gato), \+is_true('pequeno'), is_true('peludo').
raca(patosilvestre) :- animal(pato).

is_true(Q) :- format("~w?\n", [Q]), read(yes).
```

Lógica do Sistema especialista

```
animal(cachorro)  :- is_true('tem pelo'), is_true('late').
animal(gato)  :- is_true('tem pelo'), is_true('mia').
animal(pato) :- is_true('tem penas'), is_true('quack').

raca(husky) :- animal(cachorro), is_true('grande').
raca(salsicha) :- animal(cachorro), \+is_true('grande').
raca(persa) :- animal(gato), is_true('pequeno'), \+is_true('peludo').
raca(mainecon) :- animal(gato), \+is_true('pequeno'), is_true('peludo').
raca(patosilvestre) :- animal(pato).

is_true(Q) :- format("~w?\n", [Q]), read(yes).
```

Definimos as regras de um animal

Lógica do Sistema especialista

```
animal(cachorro)  :- is_true('tem pelo'), is_true('late').
animal(gato)  :- is_true('tem pelo'), is_true('mia').
animal(pato) :- is_true('tem penas'), is_true('quack').
```

Definimos as regras de raça

```
raca(husky) :- animal(cachorro), is_true('grande').
raca(salsicha) :- animal(cachorro), \+is_true('grande').
raca(persa) :- animal(gato), is_true('pequeno'), \+is_true('peludo').
raca(mainecon) :- animal(gato), \+is_true('pequeno'), is_true('peludo').
raca(patosilvestre) :- animal(pato).

is_true(Q) :- format("~w?\n", [Q]), read(yes).
```

Lógica do Sistema especialista

```
animal(cachorro)  :- is_true('tem pelo'), is_true('late').
animal(gato)  :- is_true('tem pelo'), is_true('mia').
animal(pato) :- is_true('tem penas'), is_true('quack').

raca(husky) :- animal(cachorro), is_true('grande').
raca(salsicha) :- animal(cachorro), \+is_true('grande').
raca(persa) :- animal(gato), is_true('pequeno'), \+is_true('peludo').
raca(mainecon) :- animal(gato), \+is_true('pequeno'), is_true('peludo').
raca(patosilvestre) :- animal(pato).

is_true(Q) :- format("~w?\n", [Q]), read(yes).
```

O sistema verifica pra cada regra, se este atende os critério na definição. Inclusive executa recursões

Lógica do Sistema especialista

```
animal(cachorro)  :- is_true('tem pelo'), is_true('late').
animal(gato)  :- is_true('tem pelo'), is_true('mia').
animal(pato) :- is_true('tem penas'), is_true('quack').

raca(husky) :- animal(cachorro), is_true('grande').
raca(salsicha) :- animal(cachorro), \+is_true('grande').
raca(persa) :- animal(gato), is_true('pequeno'), \+is_true('peludo').
raca(mainecon) :- animal(gato), \+is_
raca(patosilvestre) :- animal(pato).

is_true(Q) :- format("~w?\n", [Q]), read(yes).
```

Definimos que é verdadeiro sempre que escrevemos “yes” no teclado

Alguns resultados do sistema

1. Do lado esquerdo, definimos o labrador enquanto o sistema especialista verifica de uma a uma qual a melhor regra que define o que procuramos
2. Mesmo caso, porém exemplo distinto para gatos

```
?- animal(A).
tem pelo?
|    yes.
late?
|    yes.
tem raça?
|    yes.
tem pelo?
|    yes.
mia?
|    no.
tem penas?
|    no.
tem pelo?
|    yes.
late?
|    yes.
tem raça?
|    yes.

A = labrador.
```

```
?- raca(A).
tem pelo?
|    yes.
late?
|    no.
tem pelo?
|    yes.
late?
|    no.
tem pelo?
|    yes.
mia?
|    yes.
pequeno?
|    no.
tem pelo?
|    yes.
mia?
|    yes.
pequeno?
|    no.
peludo?
|    yes.

A = mainecon .
```

Conclusões & Referências

Conclusões

1. Apesar de não tão popular quanto outras linguagens, prolog oferece uma **base forte** para resolução de problemas através de um paradigma distinto
2. Bastante utilizada no ramo da IA na implementação de **sistemas de recomendação**
3. Permite entendermos “**o que**” antes de pensar no “como” - torna a resolução de problemas com foco no objetivo final

Referências bibliográficas

STERLING, L. and SHAPIRO, E. The Art of Prolog - Advanced Programming Techniques, MIT Press, 1986
http://www.swi-prolog.org/ - Site oficial da implementação SWI do Prolog
https://www.ime.usp.br/~slago/slago-prolog.pdf - Slides do professor Silvio Lago do IME
https://repositorium.sdum.uminho.pt/bitstream/1822/7780/1/exercicios-sbc.pdf - Estudos e reforços sobre Prolog prof Paulo Cortez Escola de Engenharia Universidade do Minho, Guimarães, Portugal
http://silveiraneto.net/2007/08/29/exemplos-de-prolog/ - Exercicio gerais de prolog
http://lpn.swi-prolog.org/lpnpage.php?pagetype=html&pageid=lpn-htmlch1 - Referencial oficial do SWI para estudos em Prolog
https://en.wikipedia.org/wiki/Prolog - Página oficial do prolog na wikipedia
https://pt.wikipedia.org/wiki/Fatorial - Pagina sobre fatoriais
https://pt.wikipedia.org/wiki/Sistema_especialista
https://pt.wikipedia.org/wiki/Alain_Colmerauer - Página sobre o criador da linguagem Prolog
https://rodrigorgs.github.io/aulas/mata56/aula03-prolog-recursao - Detalhes sobre recursões em Prolog
https://pt.wikipedia.org/wiki/Torre_de_Han%C3%B3i
https://pt.wikipedia.org/wiki/Sequ%C3%AAncia_de_Fibonacci
https://pt.wikipedia.org/wiki/Cl%C3%A1usula_de_Horn - Detalhes sobre as clausulas de Horn

Todos os código apresentados na aula estão disponiveis no GitHub utilizando licença GNU v.3
link: https://github.com/cobap/prolog

Appendix - Prints Prolog

Como iniciar o console

```
antonio@notebookDell:~/Área de Trabalho$ swipl
Welcome to SWI-Prolog (threaded, 64 bits, version 7.6.4)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).
```

Como ler uma base de conhecimento

```
?- [base_conhecimento].
true.
```