Aula 3: Recursividade

A recursividade é um princípio que nos permite obter a solução de um problema a partir da solução de uma instância menor dele mesmo. Para aplicar esse princípio, devemos assumir como hipótese que a solução da instância menor é conhecida.

Definição 01: Uma função é recursiva quando ela faz chamadas a sí mesma.

Definição 02: Em uma função recursiva há a necessidade da definição do passo base, que implica a condição de parada da recursividade.

Por exemplo, vamos construir uma função recursiva clássica para calcular o fatorial de N.

In [ ]:
```
fatorial(N):
    if N == 1 or N == 0:
        return 1
    return N * fatorial(N-1)
```

Em uma chamada recursiva, o interpretador faz o empilhamento das chamadas até chegar no passo base. Ao chegar no passo base, inicia-se o processo de desempilhamento. Vejamos um exemplo para o fatorial(5).

Outra versão de código recursivo, poderia ser a seguinte:

In [ ]:
```
fatorial(N, Acc=1):
    if N == 1 or N == 0:
        return Acc
    return fatorial(N-1, Acc*N)
```

Nesse exemplo, o valor calculado já está sendo armazenado no segundo parâmetro da função.

Exemplo: Vamos codificar o fatorial no Prolog:

Fatorial versão 01:

In [ ]:
```
fatorial(0, R) :- N =:= 0, R is 1.
fatorial(1, R) :- N =:= 1, R is 1.
fatorial(N, R) :- fatorial(N, N, R).

fatorial(N, Acc, R) :- N =:=1, R is Acc.
fatorial(N, Acc, R) :- N > 1,
                           Aux is N - 1,
                           Acc1 is Acc * Aux,
                           fatorial(Aux, Acc1, R).
```

Nessa primeira versão, devemos observar que:

1- A base de conhecimento é composta por cinco regras.

2- Três regras utilizam o predicado fatorial e duas regras utilizam o predicado fatorial_aux.

3- A primeira regra será verdadeira se N for igual a zero e R for 1.

Observação: No Prolog, o perador relacional de igualdade é =:=

Observação No Prolog, **is** é uma palavra reservada que pode ser utilizada para realizar uma atribuição numérica a uma variável.

4- A segunda regra será verdadeira se N for igual a um e R for 1.

5- O corpo da terceira regra é formado por uma cláusula da base de conhecimento, portanto, a terceira regra será verdadeira quando o seu corpo for verdadeiro.

6- A quarta regra será verdadeira se N for igual a um e R for Acc.

7- Na quinta regra é onde o fatorial, de fato, está sendo calculado. O corpo dessa regra possui quatro partes unidas por conjunção (E). A última parte do corpo da regra implica uma chamada recursiva, os valores passados na chamada recursiva

são previamente calculados nas partes anteriores que formam o corpo da regra. Conforme a primeira parte do corpo da regra, essa chamada recursiva irá empilhar, enquanto N > 1.

Nessa versão, vamos observar o seguinte:

Q1: Por que ao calcular o fatorial, além do valor de R, também é exibido o valor **false**?

Q2: O que acontece se retiramos N > 1 do corpo da quinta regra?

Fatorial versão 02:

In [ ]:
```
fatorial(N, R) :- N =:= 0, R is 1, !.
fatorial(N, R) :- N =:= 1, R is 1, !.
fatorial(N, R) :- fatorial_aux(N, N, R).

fatorial_aux(N, Acc, R) :- N =:=1, R is Acc, !.
fatorial_aux(N, Acc, R) :- Aux is N - 1,
                           Acc1 is Acc * Aux,
                           fatorial_aux(Aux, Acc1, R).
```

No Prolog, o símbolo de exclamação, denominado **cut**, informar que se a clásula for verdadeira, não há necessidade do algoritmo backtracking continuar a varredura na base de conhecimento.

Nessa versão 2, nós utilizamos o **cut** nas duas clásulas que determinam as regras do zero e um do fatorial, bem como na cláusla do passo base da recursão.

Ao utilizar o **cut** no passo base da recursão, não precisamos mais da relação N > 1 no corpo da quinta regra.

Vamos agora simplificar ainda mais as regras do zero, do um, e a do passo base.

Fatorial versão 03:

In [ ]:
```
fatorial(0, R) :- R is 1, !.
fatorial(1, R) :- R is 1, !.
fatorial(N, R) :- fatorial_aux(N, N, R).

fatorial_aux(1, Acc, R) :- R is Acc, !.
fatorial_aux(N, Acc, R) :- Aux is N - 1,
                           Acc1 is Acc * Aux,
                           fatorial_aux(Aux, Acc1, R).
```

Nesta versão, o objetivo foi eliminar a necessidade das relações de igualdade **=:=** nas três regras que as utilizavam. Essas relações estavam sendo utilizadas para garantir que a regra fosse verdadeira somente quando o valor da variável **N** fosse igual ao valor comparado. Ao utilizar o valor que estava sendo comparado com o termo da cabeça da regra, a regra só dará **match** se o valor passado for o especificado. Desta forma, implicitamente, estamos fazendo a comparação de igualdade.

Vamos agora simplificar ainda mais as regras do zero, do um, e a do passo base.

Fatorial versão 04:

In [ ]:
```
fatorial(0, 1) :- !.
fatorial(1, 1) :- !.
fatorial(N, R) :- fatorial_aux(N, N, R).

fatorial_aux(1, R, R) :- !.
fatorial_aux(N, Acc, R) :- Aux is N - 1,
                           Acc1 is Acc * Aux,
                           fatorial_aux(Aux, Acc1, R).
```

Nesta versão, o objetivo foi eliminar a necessidade da utilização do **is** para realizar a atribuição da variável **R**. Nas duas primeiras regras, o **is** estava sendo utilizado para determinar que o valor de **R** fosse gual a um. Para simplificar, determinamos o valor 1 como termo da cabeça da regra.

Na quarta regra o **is** também estava sendo para utilizado determinar o valor da variável **R**. Porém, nesse caso, a variável

R recebia o valor da variável **Acc**. Para simplificar e informar que o valor de **Acc** deve ser armazenado em **R**, especificamos **R** como segunda e terceira variável da cabeça.

Agora vamos renomear o predicado **fatorial_aux**

Fatorial versão 05:

In [ ]:
```
fatorial(0, 1) :- !.
fatorial(1, 1) :- !.
fatorial(N, R) :- fatorial(N, N, R).

fatorial(1, R, R) :- !.
fatorial(N, Acc, R) :- Aux is N - 1,
                       Acc1 is Acc * Aux,
                       fatorial(Aux, Acc1, R).
```

Nessa versão, apesar de parecer, não podemos dizar que essa base de conhecimento utiliza somente um predicado. Um predicado além de ser determinado pelo nome, também é determinado pela sua aridade, que corresponde ao número de termos. Portanto, nessa base de conhecimento, estamos utilizando dois predicados, quais sejam:

fatorial\2

fatorial\3

O predicado **fatorial\2** está sendo utilizado pelas três primeiras regras e o predicado **fatorial\3** pelas duas últimas regras.

Fatorial versão 06:

In [ ]:
```
fatorial(0, 1) :- !.
fatorial(1, 1) :- !.
fatorial(N, R) :- N1 is N-1,
                  fatorial(N1, R1),
                  R is R1 *  N.
```

Exemplo: Vamos analisar o exemplo do Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

Qual solução é melhor e por quê?

Solução A

In [ ]:
```
fibonacci(N):
    if N == 0: return 0
    if N == 1: return 1
    return fibonacci(N-1)+fibonacci(N-2)
```

Solução B

In [ ]:
```
fibonacci(N, X=0, Y=1):
    if N == 0: return 0
    if N == 1: return Y
    return fibonacci(N-1, Y, X+Y)
```

Vamos codificar o fibonacci no Prolog:

Versão 1

In [ ]:
```
fibonacci(0, 0) :- !.
fibonacci(1, 1) :- !.
fibonacci(N, R) :- fibonacci(N, 0, 1, R).
fibonacci(1, _, R, R) :- !.
fibonacci(N, X, Y, R) :- A is N-1,
                         B is Y,
                         C is X+Y,
                         fibonacci(A, B, C, R).
```

Nesse exemplo estamos utilizando uma variável anônima no segundo termo da regra correspondente ao passo base da recursão. No Prolog, uma variável anônima deve iniciar com underline. Ao utilizar esse tipo de variável, estamos informando que o valor passado para ela não será utilizado e nem considerado na verificação de match da cláusula.

Versão 2

In [ ]:
```
fibonacci(0, 0) :- !.
fibonacci(1, 1) :- !.
fibonacci(N, F) :- N1 is N-1,
                   N2 is N-2,
                   fibonacci(N1, F1),
                   fibonacci(N2, F2),
                   F is F1 + F2.
```

Exemplo: Escreva um programa que faça a leitura de senhas até que seja digitada a senha correta. Para cada leitura de senha errada, exiba a mensagem "Senha inválida!". Quando a senha correta for digitada, exiba a mensagem "Acesso liberado!" e encerre o programa. Considere que a senha válida é qwter.

In [ ]:
```
senha :- read(X), senha(X).

senha(qwter) :- write('Acesso liberado'), !.
senha(_) :- write('Senha inválida'), senha.
```

Nesse exemplo, estamos realizando a intração com o usuário a partir da leitura e escrita. No Prolog, **read** é utilizado para ler as informações digitadas pelo usuário e **write** para escrever as informações na tela.

Exercício

Dada a base de conhecimento que representa uma árvore binária:

In [ ]:
```
raiz(15).
no(10, 15, 20).
no(5, 10, 9).
no(17, 20, 27).
no(nil, 5, 7).
no(25,27, nil).
```

1- Utilize o predicado **no_interno/1** para construir uma regra que retorne True se o nó passado como argumento da consulta for um nó interno.

2- Utilize o predicado **no_folha/1** para construir uma regra que retorne True se o nó passado como argumento da consulta for um nó folha. Se um determinado nó não possuir um dos filhos, a ausência é representada pelo átomo nil.

3- Utilize o predicado **filhos/1** para construir uma regra que informe os filhos da esquerda e da direita de um determinado nó. Para informar os filhos, utilize o predicado **write/1**. Utilize o seguinte padrão de mensagem:

Filho da esquerda e direita: Nó esquero e Nó direito.

Somente filho da esquerda: Nó esquero.

Somente filho da direita: Nó direito.

Não possui filhos.

4- Utilize os predicados **pre_ordem/0**, **pre_ordem/1**, **visita/1** e **write/1** para construir regras que realizem a leitura da árvore em pré-ordem. O predicado de consulta é o pre_ordem/0. Para a árvore de base de conhecimento, a saída esperada é: 15 - 10 - 5 - 7 - 9 - 20 - 17 - 27 - 25 -

5- Utilize os predicados **in_ordem/0**, **in_ordem/1**, **visita/1** e **write/1** para construir regras que realizem a leitura da árvore em ordem. O predicado de consulta é o in_ordem/0. Para a árvore de base de conhecimento, a saída esperada é: 5 - 7 - 10 - 9 - 15 - 17 - 20 - 25 - 27 -

6- Utilize os predicados **pos_ordem/0**, **pos_ordem/1**, **visita/1** e **write/1** para construir regras que realizem a leitura da árvore em pós-ordem. O predicado de consulta é o pos_ordem/0. Para

a árvore de base de conhecimento, a saída esperada é: 7 - 5 - 9 - 10 - 17 - 25 - 27 - 20 - 15 -

7- Utilize os predicados **nivel/2**, **nivel/3**, para calcular o nível de um determinado nó. O predicado de consulta é o nivel/2, tal que o primeiro termo corresponde ao nó e o segundo termo à variável de resposta não instanciada. Se necessário, você pode criar mais regras com novos predicados.

8- Utilize o predicado **grau/2** para calcular o grau de um determinado nó da árvore. Ao realizar a consulta, o primeiro termo corresponde ao nó e o segundo termo à variável de resposta não instanciada.

9- Utilize o predicado **irmao/2** para informar o irmão de um determinado nó da árvore. Ao realizar a consulta, o primeiro termo corresponde ao nó e o segundo termo à variável de resposta não instanciada.

10- Utilize os predicados **altura/1** e **altura/2** para informar a altura da árvore. O predicado de consulta é o altura/1 e o seu termo corresponde à variável de resposta não instanciada.

11- Utilize os predicados **mais_a_esquerda/1** e **mais_a_esquerda/2** para informar o nó mais a esquerda da árvore. O predicado de consulta é o mais_a_esquerda/1 e o seu termo corresponde à variável de resposta não instanciada.