GHC.IO About Help

```
Welcome to GHC.IO!
Prelude> let fatorial a = product [1..a]
Prelude> fatorial 3
6
Prelude>
```


```
sumList [2,3,4,5]
= 2 + sumList [3,4,5]
= 2 + (3 + sumList [4,5])
= 2 + (3 + (4 + sumList [5]))
= 2 + (3 + (4 + (5 + sumList [])))
= 2 + (3 + (4 + (5 + 0)))
= 14
```

Programação Funcional

Introdução

Matheus Carvalho Viana

matheuscviana@ufsj.edu.br

Dcomp
Departamento de Ciência da Computação

UFSJ
Universidade Federal de São João del-Rei

Introdução

As linguagens de programação são criadas com base em um **paradigma de programação**.

O paradigma de programação determina a visão que o programador deve ter sobre a estruturação e a execução do programa.

A maioria das pessoas está acostumada com o ramo de paradigmas imperativos, no qual se encontram as linguagens procedurais e a maioria das orientadas a objetos.

Nessas linguagens, os programas são implementados como uma sequência de instruções para computador resolver o problema.

Introdução

As linguagens imperativas são similares entre si, pois todas se baseiam na arquitetura de von Neumann.

- Nessas linguagens, uma expressão é avaliada e o resultado armazenado em uma posição de memória (variável).
- As linguagens de montagem também trabalham com esse formato.
- Isso resulta em uma metodologia de nível mais baixo.

Alguns consideram que essa dependência é desnecessária para os processos de desenvolvimento de software.

Assim, surgiram outros paradigmas de programação:

- **Paradigma funcional**
- Paradigma lógico

Programação Funcional

Baseia-se no conceito matemático de função, em que para cada elemento do seu conjunto domínio (entrada) há apenas um elemento no seu conjunto contradomínio (saída).

Considera o programa como uma função matemática (ou um conjunto delas).

Os programas seguem o formato do **cálculo lambda**:

Um sistema formal que estuda funções recursivas computáveis.

Suas entidades podem ser utilizadas como argumentos e retornadas como valores de outras funções.

Exemplo:

```
sqsum(x, y) = x*x + y*y
```

*recebe um par de entradas, x e y, e retorna a soma de seus quadrados, x*x + y*y.*

Características

Não existe variável nem operação de atribuição

Nas linguagens funcionais, todas as estruturas de código (variáveis, operadores, condicionais, arrays, etc.) são, na verdade, funções, o que torna essas linguagens altamente ortogonais.

```
quicksort :: Ord a => [a] -> [a]
quicksort []    = []
quicksort (p:r) = quicksort menores ++ [p] ++ quicksort maiores
  where
     menores = [ y | y <- r, y < p ]
     maiores = [ y | y <- r, y >= p ]
```

Nesse exemplo em Haskell, p, r, menores e maiores não são variáveis, mas, sim, funções.

O símbolo = não é o operador de atribuição. É apenas um separador sintático que significa “é igual a”.

Características

Não existe estrutura iterativa

A única forma de realizar a repetição de instruções é por meio de recursão.

```
fatorial :: Int -> Int
fatorial 0 = 1
fatorial n = n * fatorial (n-1)

take :: [a] -> Int -> [a]
take [] _ -> []
take (h:t) n
  | n > 0 = n : take t (n-1)
  | otherwise = []
```

Paradigma Funcional

Não existe efeito colateral

Consiste no fato de um bloco de código alterar o valor de uma variável externa. É provocado por:

- Parâmetros de entrada com passagem por referência;
- Variáveis globais;
- Atributos de classes;
- Instruções de entrada e saída;

Aumenta a flexibilidade da linguagem, porém diminui a confiabilidade;

Nas linguagens funcionais, não existe a possibilidade das funções provocarem efeito colateral por dois motivos:

- Funções não podem acessar variáveis externas a elas;
- As variáveis, na verdade, são funções constantes;

A exceção é no caso de instruções de entrada e saída.

Características

Independência da ordem de avaliação

A ordem de avaliação de uma expressão determina a sequência em que cada operando é obtido/calculado.

Muitos compiladores/interpretadores podem alterar a ordem de avaliação das expressões para aumentar a eficiência do programa.

Seja a expressão **a + fun(a)**, a ordem de avaliação* interfere no resultado?

Não para o caso:

```
int fun(int x) {
    return x * x;
}
```

Sim para o caso:

```
int fun(int &x) {
    x = x + 1;
    return x * x;
}
```

Como não existe efeito colateral nas linguagens funcionais, as expressões podem ser avaliadas em qualquer ordem.

Características

Transparência referencial

Se uma pessoa fosse avaliar manualmente a expressão (3*ax* + *b*)(3*ax* + *c*), jamais iria avaliar a sub-expressão 3*ax* duas vezes.

Uma vez avaliada, o pessoa substituiria a sub-expressão 3*ax* pelo seu resultado em todos os casos onde ela aparecesse.

Considerando a = 2, x = 3, b = 3 e c = -2:

3 * a * x
3 * 2 * 3
6 * 3
18

(18 + b) * (18 + c)
(18 + 3) * (18 + (-2))
21 * (18 + (-2))
21 * 16
336

A transparência referencial é o que permite a aplicação dessa prática. Ela existe se quaisquer duas expressões de mesmo valor puderem ser substituídas uma pela outra, em qualquer lugar, sem afetar o resultado.

Características

Realiza avaliação preguiçosa.

Nela, as avaliações das expressões são adiadas ao máximo, sendo realizadas somente quando o valor delas for estritamente necessário.

Por exemplo, considere a expressão fun(3 + 1):

- Em uma linguagem imperativa ou OO, a expressão 3+1 é avaliada primeiro e o seu resultado (4) seria passado como entrada para a função fun;

- Em uma linguagem funcional, a própria expressão 3+1 é passada como entrada para a função fun e o seu resultado só vai ser calculado quando for necessário dentro da função.

Características

Possui suporte a currying.

Processo de transformar uma função que recebe múltiplos argumentos em uma função que recebe apenas um único argumento que é executada sobre os demais argumentos.

- O primeiro parâmetro é aplicado a função e isso cria uma **função parcialmente aplicada**;
- Essa função parcial recebe o parâmetro seguinte da função original, gerando uma nova função parcial. Isso se repete até todos os parâmetros serem aplicados.

Por exemplo, considere a seguinte função:

```
mult :: Int -> Int -> Int -> Int
mult x y z = x * y * z
```

O que acontece quando executamos **`mult 3 5 9`**?

```
> mult 3 5 9
    > (mult 3) 5 9
        > (mult 15) 9
            > 135
```

Recursão em Cauda

Permite menor uso de memória durante o processo de empilhamento, o que a torna mais rápida que a recursão comum.

Em uma recursão comum, a cada chamada recursiva realizada, é necessário guardar a posição do código onde foi feita a chamada para que continue a partir dali assim que receber o resultado.

```
int fatorial(int n) {
  if (n <= 1) return n;
  else        return n * fatorial(n - 1);
}
```

Note que a multiplicação só pode ser resolvida depois da chamada recursiva. Isso provoca o empilhamento das funções chamadas.

Recursão em Cauda

Em uma recursão de cauda, não é necessário guardar a posição onde foi feita a chamada, visto que **a chamada recursiva é a última operação realizada pela função**.

```
int fatorial_cauda(int num) {
   return fatorial_aux(num, 1);
}

int fatorial_aux(int num, int parcial) {
   if (num == 1)
      return parcial;
   else
      return fatorial_aux((num - 1), (parcial * num));
}
```

Imperativa vs Funcional

Característica	Linguagens Imperativas	Linguagens Funcionais
Inspiração	Arquitetura Von Neumann	Cálculo Lambda
Enfoque	Sequência de passos	Descrição do problema
Variáveis	Definem o estado do programa	Não há
Sequência das instruções	Importante	Pouco relevante
Repetição	Laços e recursão	Recursão
Decisão	Estruturas condicionais	Estruturas condicionais

Código em C

```
int x = 10;
int y = 12;
int z = x + y;
```

Código em Haskell

```
z = x + y
x = 10
y = 12
```

Considerações

Muitos cientistas afirmaram que as linguagens funcionais são melhores do que as imperativas porque resultam em programas:

- mais legíveis;
- mais fáceis de usar;
- mais confiáveis;
- sem efeitos colaterais; e
- portanto, mais propensos a serem corretos.

Contudo, possuem desvantagens, como:

- serem menos eficientes;
- consumir muita memória por causa das recursões.

Considerações

Linguagens funcionais não são tão disseminadas como as imperativas e OO, mas são utilizadas em aplicações que envolvem paralelismo e transações atômicas.

Exemplos de linguagens funcionais:

- **LISP** foi desenvolvida para computação simbólica e aplicações de processamento de listas, comuns na área de inteligência artificial;
- **Haskell** é utilizada em alguns recursos do Linux, construção de compiladores e programação concorrente;
- **Closure** é uma linguagem funcional que roda na Máquina Virtual do Java e vem ganhando espaço na programação web e microsserviços. Usada pelo Nubank e outras *fintechs;*
- **Erlang** foi desenvolvida para suportar aplicações distribuídas e tolerantes a falhas para serem executadas em um ambiente de tempo real e ininterrupto. É utilizada em sistemas de telecomunicações, bancos, comércio eletrônico e mensagens instantâneas;
- **Scala** é orientada a objetos no sentido que todo valor é um objeto e funcional no sentido que toda função é um valor.

Considerações

Atualmente, diversas linguagens imperativas e/ou OO incorporam recursos de linguagens funcionais, como funções lambda, funções de alta ordem e listas dinâmicas.

Python

```
>>> soma = lambda x, y: x+y
>>> soma(1, 2)
3
>>> map(str, [2, 4, 6])
['2', '4', '6']
```

JavaScript

```
const lst1 = [1,2,3,4,5];
const lst2 = lst1.map(e => e*2);
const lst3 = lst1.filter(e => e%2==0);
```

Java

```
List<String> lst = Arrays.asList("1", "2", "3");
int soma = lst.stream().
   map(item -> Integer.parseInt(item)).reduce(0, (a,b) -> a+b);
```