Paradigmas de Programação

MCTA016-13 - Paradigmas de Programação

Emilio Francesquini
e.francesquini@ufabc.edu.br

2019.Q2

Centro de Matemática, Computação e Cognição
Universidade Federal do ABC

UFABC

Disclaimer

UFABC

- Estes slides foram preparados para o curso de {Paradigmas de Programação na UFABC}.
- Este material pode ser usado livremente desde que sejam mantidos, além deste aviso, os créditos aos autores e instituições.
- Conteúdo baseado no texto preparado, e gentilmente cedido, pelo Professor Fabrício Olivetti de França da UFABC.

Paradigmas de Programação

Definição: estilo de programação, a forma como você descreve a solução computacional de um problema.

UFABC

- Muitos cursos de Computação e Engenharia iniciam com paradigma imperativo e estruturado (vide Processamento da Informação e Programação Estruturada).
- Exemplo clássico da receita de bolo (que não é a melhor forma de descrever o conceito de algoritmo).

Alguns dos paradigmas mais populares:

- **Imperativo:** um programa é uma sequência de comandos que alteram o estado atual do sistema até atingir um estado final.
- **Estruturado:** programas com uma estrutura de fluxo de controle e uso de procedimento e funções.
- **Orientado a objeto:** organização através de objetos que contém dados, estados próprios e métodos que alteram ou recuperam os dados/estados. Os objetos comunicam entre si para compor a lógica do programa.

UFABC

- Declarativo: especifica o que você quer, mas sem detalhar como fazer.
- Funcional: programas são avaliações de funções matemáticas sem alterar estados e com dados imutáveis.
- Lógico: especifica-se um conjunto de fatos e regras, o interpretador infere respostas para perguntas sobre o programa.

UFABC

- Muitas das linguagens de programação são, na realidade, multi-paradigmas
  - Contêm características de diversos paradigmas
  - Contudo, na prática, elas acabam favorecendo um paradigma específico o que acaba lhes conferindo o título de linguagem ”funcional”ou ”imperativa”por exemplo

Paradigma Imperativo

UFABC

- Descrevo passo a passo o que deve ser feito.
- Infame `goto`.
- Evoluiu para **procedural** e **estruturado** com `if`, `while`, `for`, ...

Paradigma Imperativo

UFABC

```
    aprovados = {};
    i = 0;
inicio:
    n = length(alunos);
    if (i >= n) goto fim;
    a = alunos[i];
    if (a.nota < 5) goto proximo;
    nome = toUpper(a.nome);
    adiciona(aprovados, nome);
proximo:
    i = i + 1;
    goto inicio;
fim:
    return sort(aprovados);
```

Paradigma Imperativo

UFABC

- O fluxo de controle é explícito e completo, nada está escondido.
- Linguagem um pouco mais alto nível do que a linguagem de máquina.
- Ao seguir o passo a passo, você chega no resultado... mas pode ser difícil racionalizar qual será ele.
- Qualquer um pode usar suas variáveis globais e suas funções, para o seu uso intencional ou não...

Paradigma Estruturado

UFABC

- Estrutura o código imperativo em blocos lógicos.
- Esses blocos contém instruções imperativas que, quando escritas, foram feitas para cumprir um único objetivo.
- Elimina o uso de `goto`.
  - Ou deveria ter eliminado. Contra-exemplos: Java, C, ...

Paradigma Estruturado

UFABC

```
1  aprovados = {};
2  for (i = 0; i < length(alunos); i++) {
3      a = alunos[i];
4      if (a.nota >= 5) {
5          adiciona(aprovados, toUpper(a.nome));
6      }
7  }
8  return sort(aprovados);
```

Paradigma Estruturado

UFABC

- O programa é dividido em blocos lógicos, cada qual com uma função explícita (se for bom programador).
- Estimula o uso de variáveis locais pertencentes a cada bloco lógico.
- Não evita que certas informações sejam utilizadas fora do seu contexto.
- Usa mudança de estados, o que pode levar a *bugs*.

UFABC

- Encapsula os dados em classes cada qual contendo seus próprios estados e métodos, não compartilhados.
- Um objeto pode se comunicar com outro, não usa (ou evita) variáveis globais.
  - Linguagens como Smalltalk usam o termo *troca de mensagens* enquanto em outras como Java fala-se de *chamadas/invocação de métodos*
- Métodos são divididos em métodos de instância, que lidam com o estado e manipulação de um objeto específico e métodos de classe/estáticos que, tipicamente, tratam requisições que não são particulares a um objeto

```
1  class Aprovados {
2      private ArrayList aprovados;
3      public Aprovados () {
4          aprovados = new ArraLista();
5      }
6      public addAluno(aluno) {
7          if (aluno.nota >= 5) {
8              aprovados.add(aluno.nome.toUpper());
9          }
10     }
11     public getAprovados() {
12         return aprovados.sort();
13     }
14
15 }
```

UFABC

- Encapsula códigos imperativos para segurança e reuso.
- Evita o uso de variáveis globais, inibe uso indevido de trechos de códigos.
- Não são todos os problemas que podem ser facilmente ou naturalmente modelados como objetos.
- Composição de funções é feita através de herança, que pode *bagunçar* o fluxo lógico do código.
- Uso pesado de estados mutáveis.
- Difícil de paralelizar[1].

[1] Este é um ponto controverso já que a orientação a objetos surgiu como uma solução para concorrência (e não paralelização) em Simula. Neste contexto nos referimos às condições de corrida que são inerentes à POO mas que são inexistentes no paradigma funcional, por exemplo

Paradigma Declarativo

UFABC

- O fluxo lógico é implícito.
- Separação clara entre o que o programador deseja obter do como proceder para obter o que ele deseja
- Linguagens de alto nível que permite aos programadores dizer apenas o que desejam

UFABC

```sql
SELECT   UPPER(nome)
FROM     alunos
WHERE    nota >= 5
ORDER BY nome
```

Paradigma Declarativo

UFABC

- Utiliza uma linguagem específica para o domínio da aplicação (*Domain Specific Language* - DSL), atingindo um nível mais alto que outros paradigmas.
- Minimiza uso de estados, levando a menos bugs.
- Fazer algo não suportado nativamente na linguagem pode levar a códigos complexos ou uso de linguagens de outros paradigmas.
- Pode ter um custo extra no desempenho.

UFABC

- Especifica-se apenas fatos e regras de inferência.
- O objetivo (retorno) é escrito em forma de pergunta.

UFABC

```
aprovado(X) :- nota(X,N), N>=5.

sort(
  findall(Alunos, aprovado(Alunos), Aprovados)
    )
```

UFABC

- O compilador constrói o programa para você, baseado em fatos lógicos.
- Provar a corretude do programa é simples.
- Algoritmos mais complexos podem ser difíceis de expressar dessa forma.
- Costuma ser mais lento para operações matemáticas.

UFABC

- Baseado no cálculo λ.
- Programas são composições de funções.
- **Não** utiliza estados.
- Declarativo.

UFABC

```haskell
sort [nome aluno | aluno <- alunos, nota aluno >= 5]
```

UFABC

- 🙂 Expressividade próxima de linguagens declarativas, mas sem limitações.
- 🙂 Não existe estado e mutabilidade, isso reduz a quantidade de *bugs*.
- 😕 Como fazer algo útil sem estados?
- 😕 A ausência de mutabilidade dificulta o gerenciamento de memória, intenso uso de *Garbage collector*.

Paradigma Funcional

Características

UFABC

- Funções puras
- Recursão
- Avaliação Preguiçosa

Efeito colateral e estados

UFABC

Efeito colateral ocorre quando uma função altera algum estado global do sistema:

- Alterar uma variável global
- Ler entrada de dados
- Imprimir algo na tela

Funções Puras

UFABC

Funções puras são funções que não apresentam efeito colateral.

Ao executar a mesma função com a mesma entrada sempre terei a mesma resposta.

Se não temos efeito colateral...

- ...e o resultado de uma expressão pura não for utilizado $\rightarrow$ não precisa ser computado.
- ...o programa como um todo pode ser reorganizado e otimizado.
- ...é possível computar expressões em qualquer ordem (ou até em paralelo).

Funções Puras

UFABC

```
1  double dobra(double x) {
2      return 2 * x;
3  }
```

Funções Impuras

UFABC

```
1  double i = 0;
2
3  double dobraMaisI(double x) {
4      i += 1;
5      return 2 * x + i;
6  }
```

Pergunta

UFABC

Classifique as seguintes funções em puras ou impuras:

- `strlen`
- `printf`
- `memcpy`
- `getc`

Resposta

UFABC

- `strlen`: pura
- `printf`: impura
- `memcpy`: pura
- `getc`: impura

Funções Impuras

UFABC

```c
double media (int *valores, int n) {
    double soma = 0;
    int i;
    for (i = 0; i < n; i++)
        soma_valor(&soma, valores[i]);
    return soma / n;
}

void soma_valor (double *soma, int valor) {
    soma += valor;
}
```

Funções impuras

- Funções impuras são virais!
  - Se sua função chama uma função impura, então sua função é impura

Pergunta

Se sua função só chama funções puras, então ela é pura?

Pergunta 2

Um programa que contenha apenas funções puras é útil?

Programação sem bugs

UFABC

- A ausência de estados permite evitar muitos erros de implementação.
- O lema da linguagem Haskell: ”se compilou, o código está correto!”(e não só pela pureza).

Iterações vs Recursões

UFABC

- Em linguagens funcionais os laços iterativos são implementados via recursão
  - Geralmente leva a um código enxuto e declarativo.

Iterações vs Recursões

UFABC

```
1  int gcd (int m, int n) {
2    int r = m % n;
3    while (r != 0) {
4         m = n;
5         n = r;
6         r = m%n;
7    }
8    return m;
9  }
```

Iterações vs Recursões

UFABC

```haskell
mdc 0 b = b
mdc a 0 = a
mdc a b = mdc b (a `rem` b)
```

Avaliação Preguiçosa

UFABC

- Algumas linguagens funcionais implementam o conceito de avaliação preguiçosa.
- Quando uma expressão é gerada, ela gera uma promessa de execução ou *thunk*.
- Se e quando for necessário, o *thunk* é avaliado.

Avaliação Estrita

UFABC

```c
int main () {
    int x = 2;
    printf("%d\n", f(x * x, 4 * x + 3));
    return 0;
}

int f(int x, int y) {
    return 2 * x;
}
```

Avaliação Estrita

UFABC

```c
int main () {
    int x = 2;
    printf("%d\n", f(2 * 2, 4 * 2 + 3));
    return 0;
}

int f(int x, int y) {
    return 2 * x;
}
```

Avaliação Estrita

UFABC

```c
int main () {
    int x = 2;
    printf("%d\n", f(4, 4 * 2 + 3));
    return 0;
}

int f(int x, int y) {
    return 2 * x;
}
```

Avaliação Estrita

UFABC

```c
int main () {
    int x = 2;
    printf("%d\n", f(4, 11));
    return 0;
}

int f(int x, int y) {
    return 2 * x;
}
```

Avaliação Estrita

UFABC

```c
int main () {
    int x = 2;
    printf("%d\n", 8);
    return 0;
}

int f(int x, int y) {
    return 2 * x;
}
```

Avaliação Preguiçosa

UFABC

```haskell
f x y = 2 * x

main = do
  let z = 2
  print (f (z * z) (4 * z + 3))
```

UFABC

```haskell
f x y = 2 * x

main = do
  let z = 2
  print (2 * (z * z))
```

Avaliação Preguiçosa

UFABC

```haskell
f x y = 2 * x

main = do
  let z = 2
  print (8)
```

A expressão `4 * z + 3` nunca foi avaliada!

UFABC

Isso permite a criação de listas infinitas:

```
1 [2 * i | i <-[0..]]
```

- Linguagem puramente funcional (não é multi-paradigma)
- Somente aceita funções puras (como tratar entrada e saída de dados?)
- Declarativa
- Avaliação Preguiçosa
- Dados imutáveis
- Tipagem estática e forte