Paradigma Funcional

"Haskell, Lisp e SQL são as únicas linguagens de programação nas quais gasto mais tempo pensando do que digitando."(Philip Greenspun, MIT)

História

- 1930 – matemáticos Alonzo Church e Stephen Kleene desenvolvem o **λ-cálculo** (teoria de funções que é simples mas poderosa, e providencial na fundamentação teórica de PF).
- 1950 – LISP (**LIS**t **P**rocessing, 1958, MIT), baseada em parte no λ-cálculo.
- 1960 – ISWIM, 1ª puramente funcional, fortemente baseada no λ-cálculo.

História

- Década de 70 – FP (*Funcional Programming*), que enfatiza *high-order functions*, e ML (Meta Linguagem), que introduz o polimorfismo de funções e o uso de inferência de tipos de dados.
- Década de 80 – Miranda é pioneira no uso da *lazy evaluation*, e as pesquisas em programação funcional resultaram na criação de mais de 12 linguagens.

Paradigma Funcional

- Objetivo: imitar as funções matemáticas ao máximo.
- Estilo de programação que enfatiza a **avaliação de expressões**. Uma expressão seria composta da aplicação de funções e argumentos.
- Não há o conceito de estado nem comandos como atribuição.

Fundamentos

- Linguagem puramente funcional (LPF) não usa variáveis ou instruções de atribuição – não possibilitando construções iterativas.
- Repetição é feita por recursão.
- Execução de uma função sempre produz o mesmo resultado dados os mesmos argumentos – **Transparência Referencial** (função sem efeito colateral = função "pura").

Fundamentos

- Programas consistem em definições de funções e especificação de aplicações de funções.
  - Funções são os elementos principais.
  - Funções podem receber funções como argumentos e podem devolver como resultado outra função.
  - Estruturas de dados podem conter funções como elementos.
- A execução consistem na avaliação das aplicações das funções.

Fundamentos

- Uma LPF deve prover:
  - Uma estrutura (ou conjunto) para representar dados (tradic. listas).
  - Um conjunto de funções primitivas para manipular objetos básicos de dados (LISP e ML oferecem funções p/tratamento e construção de listas).
  - Um conjunto de formas funcionais para construção de novas funções, mais complexas (composição de funções).
  - Um operador de aplicação de função.

Formas Funcionais

Uma **função de ordem superior** (*high order function*) é aquela que toma funções como parâmetros, produz uma função como resultado, ou ambos.

Ex.: **map** da API de Collections do Java, que aplica uma função recebida via parâmetro) a cada elemento da lista, retornando nova lista.

```
// cria a função cubo
def cubo(n: Int) = n * n * n
//cria uma lista de inteiros
val inteiros = List(1,2,3,4)
//aplica a função cubo a cada elemento da
//lista anterior, retornando uma nova lista
val cubos = inteiros.map(cubo)
// a nova lista cubos = List(1,8,27,64)
```

Formas Funcionais

1. **Tipo comum: composição de funções** – possui 2 parâmetros funcionais e produz uma função cujo valor é a 1ª função paramétrica real aplicada ao resultado da 2ª. Operador °.

- Exemplo: h ≡ f°g

  Sendo f(x) ≡ x+2 e g(x) ≡ 3*x

  Define-se h(x) ≡ f(g(x)), ou h(x) ≡ (3*x)+2

Formas Funcionais

2. **Tipo construção** – forma funcional que toma uma lista de funções como parâmetros e quando aplicada a um argumento, aplica cada parâmetro a este argumento e coleta resultados em uma lista ou sequência.

- Exemplo:
  - Sendo g(x) ≡ x*x, h(x) ≡ 2*x e i(x) ≡ x/2
  - Define-se [g,h,i](4)
  - Produzindo (16,8,2)

Formas Funcionais

3. **Tipo Apply-to-all** – forma funcional que toma uma única função como parâmetro e se aplicada a uma lista de argumentos, aplica seu parâmetro a cada um dos valores no argumento e coleta resultados em uma lista ou sequência.

- Exemplo:
  - Sendo h(x) ≡ x*x
  - Define-se a(h, (2,3,4))
  - Produzindo (4,9,16)

Fundamentos

- **Avaliação preguiçosa** (**lenta**) – técnica usada para atrasar a avaliação de uma expressão até que o seu valor seja necessário e que evita avaliações repetidas.
- Benefícios: capacidade de definir estruturas controle como funções regulares (melhor que usar primitivas internas); a capacidade de definir estruturas de dados potencialmente infinitas (implementação mais simples de alguns algoritmos) e aumento de desempenho (evitando cálculos desnecessários e condições de erro na avaliação de expressões compostas).

(*)Avaliação rápida – estratégia de avaliar todos os argumentos para uma função no momento da chamada.

Cálculo Lambda (**λ-cálculo**)

- Pode ser visto como uma linguagem de programação abstrata em que funções podem ser combinadas para formar outras funções.
- Uma expressão lambda especifica os parâmetros e a definição de uma função, mas não o nome

Cálculo Lambda (**λ-cálculo**)

- O alfabeto do cálculo lambda é composto pelos símbolos:
  - um conjunto de constantes c1, c2,....cn
  - um conjunto de variáveis x1, x2,....xn
  - o símbolo de abstração λ
  - agrupadores (,) para delimitar escopo das abstrações

Cálculo Lambda (**λ-cálculo**)

- As expressões do cálculo lambda são de três tipos:
  - identificador único como x, ou constante como 3.
  - definição de função na forma (λx.**e**), onde **e** representa o corpo da função e **x** o parâmetro da função.
    - Ex.: função quadrado é λx.x*x.
  - aplicação da função na forma (**e1 e2**), onde função **e1** é aplicada à expressão **e2**.
    - Ex.: função quadrado aplicada ao valor 2 é ((λx.x*x) 2).

Funções lambda

- Formato:

[<nomeFunção>≡] λ <elementosDomínio>. <definição>

- Ex.: dobro ≡ λx. x+x
- Função com domínio composto (tupla)
  - `max: inteiro x inteiro → inteiro`
  - `max: λx,y. if x>y then x else y`
- Aplicação (formato lambda)
  - `max: <4,2>`
- Definição de uma função com aplicação em outra:
  - `abs ≡ λx. max:<x,-x>`

Funções lambda

- Como se ordenava uma lista no Java 7:

```
Collections.sort(cars,new Comparator<Car>(){
    @Override
    public int compare(Car car1, Car car2) {
        return car1.getId() - car2.getId();
    }
});
```

- Agora no Java 8:

```
Collections.sort(cars, (car1, car2) ->
car1.getId() - car2.getId());
```

LP Imperativa X Funções

- Caracterizada por três conceitos: variáveis, atribuições e seqüências	- Em matemática, variáveis são amarradas a valores e não trocam de valor
- Estado de um programa mantido por variáveis de programa	- Função matemática define um mapeamento de um valor do domínio para um valor da imagem
- Variáveis associadas com posições de memória (endereço + valor)	- Relaciona cada elemento do domínio com apenas um da imagem
- Acesso a variáveis direto ou indireto (através endereços)	
- Alteração das variáveis através de comandos de atribuição que introduzem dependência no programa	
- Resultado de um programa depende da ordem dos comandos de atribuição	- Valor de uma função não depende da ordem de execução
- Laços são usados para processar valores (percorrem seqüências de localização de memória)	- Valores processados através da aplicação de funções; Recursão é usada no lugar da iteração (laço) junto com expressões de condição
- Baseadas em estado e orientada a comandos	- Baseada em valor e aplicativa

Imperativo X Funcional

- O Paradigma Funcional pode ser visto como uma abstração de alto nível sobre a programação imperativa, sendo um conjunto de técnicas e conceitos criados para restringir ao máximo estados mutáveis e efeitos colaterais.
  - Isso facilita a criação de sistemas concorrentes e reativos – aplicações concorrentes que tirem o máximo proveito dos recursos de hardware são menos sujeitas a erros adotando a programação funcional.

Vantagens do Paradigma Funcional

- O alto nível de abstração faz com que os programas funcionais sejam mais pequenos, claros, rápidos.
- Interessantes pela sua simplicidade sintática.
- Facilidade de descrever problemas recursivos.
- Permite verificação formal de programas, em virtude da transparência referencial.
- Programas são pequenos e com alto poder de expressão, permitindo prototipação rápida.
- Maior facilidade na escrita de códigos concorrentes.

Desvantagens do Paradigma Funcional

- Os programas funcionais podem ser menos eficientes pois a maioria das linguagens é interpretada – ineficiência em comparação com linguagens de programação "tradicionais".
- Difícil prever os custos de execução (tempo/espaço).
- "O mundo não é funcional!".
- Alguns algoritmos são mais eficientes quando implementados de forma imperativa.
- Mecanismos primitivos de E/S e formatação.

Linguagens Funcionais

- Haskell (1990), derivada de Miranda – mais conhecida e próxima da "pureza" funcional.
- Clojure (2007), dialeto de LISP.
- Scheme (1975), Ruby (1995), Ocaml (1996), Scala (2001), F# (2005), Rust (2010) – multiparadigma
- Outras notáveis p/fins comerciais: Mathematica (matemática simbólica), R (estatística) e K (análise financeira).

Curiosidades

- Surgiram nas linguagens funcionais:
  - Garbage Collection
  - Funções Anônimas
  - Programação genérica, polimorfismo de tipo

(*) As funções anônimas ou funções lambda não tem nome e podem ser criadas/ executadas em qq momento da execução do programa, não precisando ser definidas, e apenas retornando o valor de uma expressão (sem precisar do return).

Curiosidades

- LPF são utilizadas em várias empresas e projetos:
  - LISP: desenvolvimento do sistema de e-commerce via web, posteriormente vendido para o Yahoo por US$40 milhões, na época do *boom* da internet
  - Autocad possui codificação em LISP
  - SuperMario 64 usa LISP internamente
  - ERLANG (1986): LPF da Ericsson

Curiosidades

- Haskell - ferramentas para manipulação de programas PHP
- Erlang - parte do serviço de chat de facebook
- Scala (linguagem híbrida) - serviços de filas de mensagens no Twitter
- Scheme e LISP - ensino de programação em várias universidades
- ML - verificação de hardware e software na Microsoft e Intel

Exemplos de códigos

LISP

Função fatorial

```
(defun fact (n)
  (cond ((< n 0) nil
        ((= n 0) 1)
        (t (times n (fat(minus n 1))))
  )
)
```

Chamada exemplo: `(fact 6)`

Exemplos de códigos LISP

- N-ésimo termo de Fibonacci

```
(defun nfib(n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (plus (nfib(minus n 2))
                 nfib(minus n 1 ))))
  )
)
```

Chamada exemplo: `(nfib(5))`

Exemplos de códigos LISP

- Somar elementos de uma lista

```
(defun somael(x)
  (cond ((null x) 0)
        (t (plus (car x) somael(cdr x)))
  )
)
```

Chamada exemplo: `(somael([1,2,3,4]))`

Common Lisp OO - Shape, Circle, Rectangle

```
;; define the various classes
(defclass shape ()
  ((pos-x   :type real
     :accessor pos-x)
   (pos-y   :type real
     :accessor pos-y))
  (:documentation "A generic shape in a 2
  dimensional plane"))

(defclass rectangle (shape)
  ((width   :type real
         :accessor width)
   (height  :type real
         :accessor height))
  (:documentation "A rectangle"))

(defclass circle (shape)
  ((radius  :type real
     :accessor x-radius))
  (:documentation "An circle"))
```

Common Lisp OO - Shape, Circle, Rectangle

```
;; define the methods
(defmethod move-to ((thing shape) new-x new-y)
  "Move the shape THING to the new position (new-
  x, new-y)"
  (setf (pos-x thing) new-x
   (pos-y thing) new-y))

(defmethod shift-to ((thing shape) delta-x delta-y)
  "Move the shape THING by (delta-x, delta-y)"
  (incf (pos-x thing) delta-x)
  (incf (pos-y thing) delta-y))

(defmethod scale ((rec rectangle) factor)
  "Scale the rectangle REC by factor"
  (with-slots (width height) rec
   (setf width (* factor width)
         height (* factor height))))

(defmethod scale ((cir circle) factor)
  "Scale the circle CIR by factor"
  (setf (radius cir) (* (radius cir) factor)))
```

Exemplos de códigos

Haskell

- Sequência de Fibonacci:

```
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib(n - 1) + fib(n - 2)
```

Exemplos de códigos

Haskell - Quicksort

```
qsort [] = []
qsort (x:xs) = qsort xs1 ++ [x] ++ qsort xs2
        where xs1 = [x' | x'<-xs, x'<=x]
                    xs2 = [x' | x'<-xs, x'>x]
```

Exemplos de códigos Java

```
Thread thread = new Thread(new Runnable() {
  public void run() {
    System.out.println("In another thread");
  }
});
thread.start();
System.out.println("In main");
```

Java 1.0

```
Thread thread = new Thread(() ->
System.out.println("In another thread"));
```

Java 8.0

O construtor da classe `Thread` recebe uma expressão lambda no lugar da instância anônima do `Runnable`.

Exemplos de códigos Java

Versão *imperativa* simplificada do algoritmo conhecido como Crivo de Eratóstenes para encontrar números primos

```
public static List<Integer> gerarNumerosPrimosImperativo(int
quantidade) {
    List<Integer> primos = new ArrayList<>(quantidade);
    for(int i = 2; primos.size() < quantidade; i++) {
      boolean ehPrimo = true;
      for (final int primo: primos) {
        if (i % primo == 0) { ehPrimo = false; break;  }
      }
      if(ehPrimo) { primos.add(i); }
    }
    return primos;    }
```

Exemplos de códigos Java

Versão declarativa simplificada do algoritmo conhecido como Crivo de Eratóstenes para encontrar números primos

```
public static List<Integer> gerarNumerosPrimosDeclarativo(int quantidade) {
    List<Integer> primos = new ArrayList<>(quantidade);
    IntStream.iterate(2, i -> i + 1).
      filter(i -> {
        for (final int primo: primos) {
          if (i % primo == 0) { return false; }
        }
         return true;
      }).limit(quantidade).forEach(primos::add);
    return primos;   }
```

Exemplos de códigos

Scheme (multiparadigma)

```
(define hello-world
  (lambda ()
    (begin
        (write 'Hello-World)
        (newline)
        (hello-world))))
```

Exemplos de códigos

Phyton (multiparadigma)

```
transformar = lambda x, y: x**y
transformar(3, 4)

iguais = lambda x, y: "Sim" if x == y
else "Nao"
iguais(54, 55)
```