Paradigmas de Linguagens de Programação

Linguagens de Programação Funcional

Introdução

Introdução

- A maioria das linguagens apresentadas até agora são **imperativas**, baseadas na **arquitetura de Von Neumann**.
- Essa arquitetura influenciou profundamente o design das linguagens:
  - O **modelo de memória compartilhada** e a **execução sequencial** são seus pilares.
- Apesar de eficiente, esse modelo é visto por alguns como **limitador** de estratégias alternativas de desenvolvimento.

Introdução

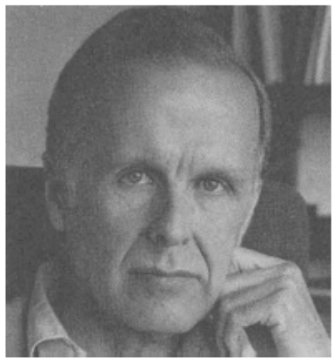
- O **paradigma funcional**, inspirado nas **funções matemáticas**, surgiu como **alternativa às linguagens imperativas**.
- Seu objetivo é eliminar a dependência da **mudança de estado** e do **uso explícito de variáveis**.
- Programas são definidos **por meio de funções e expressões**, não de comandos.
- O **controle da execução** é substituído pela **avaliação de expressões**.

Introdução

- **John Backus**, criador do **Fortran**, recebeu o **Prêmio Turing (1977)** e apresentou o artigo:
  - *“Can Programming Be Liberated from the von Neumann Style?”*
- Nesse trabalho, ele critica o modelo imperativo e propõe uma **nova forma de pensar programas como transformações de funções**.
- O artigo é um **marco teórico** que motivou o desenvolvimento e a pesquisa em **linguagens puramente funcionais**.

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



General permission to make fair use in teaching or research of all or part of this material is granted to individual readers and to nonprofit libraries acting for them provided that ACM's copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery. To otherwise reprint a figure, table, other substantial excerpt, or the entire work requires specific permission as does republication, or systematic or multiple reproduction.

Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

© 1978 ACM 0001-0782/78/0800-0613 $00.75

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

613

Communications of the ACM

August 1978
Volume 21
Number 8

Introdução

- Nos programas imperativos, o **estado** é representado por variáveis que mudam durante a execução.
- O programador precisa acompanhar essas mudanças, o que **dificulta o raciocínio sobre o programa**.
- Em programas funcionais puros, **não há variáveis mutáveis** — cada função devolve um novo valor sem alterar o anterior.
- Essa ausência de estado torna o comportamento do programa **mais previsível e matematicamente tratável**.

Introdução

- **Lisp** foi a **primeira linguagem funcional pura**, projetada no final da década de 1950.
- Baseada em **funções matemáticas** e **processamento simbólico**, Lisp introduziu conceitos como **funções de primeira classe** e **listas como estrutura central**.
- Rapidamente evoluiu com recursos que aumentaram sua **eficiência e aplicabilidade**, especialmente em **inteligência artificial**, **representação de conhecimento** e **aprendizado automático**.

Introdução

- **Scheme** é um **dialeto de Lisp** de **escopo estático**, usado amplamente no ensino de programação funcional.
- **Common Lisp** surgiu nos anos 1980 como uma **unificação dos dialetos** de Lisp existentes.
- Ambas consolidaram o Lisp como **base das linguagens funcionais modernas**.
- O desenvolvimento de linguagens como **ML**, **Haskell**, **OCaml** e **F#** ampliou o alcance do paradigma funcional.

Funções matemáticas

- Uma **função matemática** é um **mapeamento** entre dois conjuntos:
  - **Domínio:** conjunto de entrada
  - **Imagem:** conjunto de saída
- Cada elemento do domínio está associado a **exatamente um elemento da imagem**.
- A definição de uma função especifica:
  - O conjunto domínio
  - O conjunto imagem
  - A **regra de mapeamento** entre eles

$$x = 3 \longrightarrow [\ f(x) = x^2\ ] \longrightarrow 9$$

Funções matemáticas

- Funções são **aplicadas a elementos do domínio**, fornecidos como parâmetros.
- O **resultado** pertence ao conjunto imagem.
- O domínio pode ser formado por **um ou mais parâmetros**.

$$x = 3 \longrightarrow [\ f(x) = x^2\ ] \longrightarrow 9$$

Funções matemáticas

- A ordem de avaliação de expressões é controlada por **recursão** e **expressões condicionais**, não por **sequência** ou **repetição iterativa**, comuns nas linguagens imperativas.
- Funções matemáticas **não possuem efeitos colaterais** e **não dependem de valores externos**.
- Elas **sempre mapeiam** um mesmo elemento do domínio no mesmo elemento da imagem.
- Em contraste, **subprogramas imperativos** podem depender de variáveis globais ou locais, tornando **difícil prever seus resultados** e efeitos colaterais.

Funções matemáticas

- Na matemática **não existe o conceito de variável que modela uma posição de memória**.
- Em linguagens imperativas, variáveis locais representam **posições mutáveis de memória**.
- Em funções matemáticas, a computação ocorre por meio da **avaliação de expressões**, e não por **atribuição de valores**.

Funções matemáticas

Funções simples

- Uma **função** é escrita com um nome, uma lista de parâmetros entre parênteses e uma **expressão de mapeamento**.
- Exemplo:

$$cube(x) \equiv x * x * x$$

- O símbolo **≡** significa “é definido como”.
- Os conjuntos **domínio** e **imagem** são os **números reais** neste exemplo.
- O parâmetro **x** representa um membro do conjunto domínio, fixado para um valor específico durante a **avaliação** da função.
- Essa fixação distingue os parâmetros das funções matemáticas das variáveis em linguagens imperativas.

Funções matemáticas

Funções simples

- Aplicar uma função significa **avaliar** sua expressão de mapeamento substituindo o parâmetro por um valor do domínio.
- Durante a avaliação, **o parâmetro é constante** (não pode ser alterado).
- Exemplo:

$$\text{cube}(2.0) \equiv 2.0 * 2.0 * 2.0 = 8$$

- O parâmetro **x** é vinculado a **2.0** e não há parâmetro desvinculado.
- Assim, cada ocorrência de **x** é substituída por **2.0** e avaliada como uma **constante**.

Funções matemáticas

Funções simples

- Os primeiros trabalhos teóricos sobre funções (Alonzo Church, 1941) introduziram a **notação lambda**.
- Uma **expressão lambda** define uma função **não nomeada**, especificando seus parâmetros e mapeamento.
- Exemplo:

$$\lambda(x)\ x * x * x$$

Funções matemáticas

Funções simples

- Church desenvolveu o **cálculo lambda**, um sistema formal para definir funções, aplicação de funções e recursão.
- Avaliar uma expressão lambda significa aplicar um valor ao parâmetro:

$$(\lambda(x)\ x * x * x)(2) \rightarrow 8$$

O **cálculo lambda** é a base teórica das linguagens de programação **funcionais** modernas.

Funções matemáticas

Formas funcionais

- Uma **função de ordem superior**, ou **forma funcional**, é aquela que:
  - Recebe uma ou mais **funções como parâmetros**;
  - Ou **retorna uma função** como resultado;
  - Ou faz **ambos**.
- Um tipo comum de forma funcional é a **composição funcional**, em que duas funções são combinadas para formar uma terceira.

Funções matemáticas

Formas funcionais

- A composição funcional combina duas funções, de modo que o resultado de uma é a entrada da outra.
- Representa-se por um círculo (∘) entre as funções:

$$\mathbf{h \equiv f \circ g}$$

$$f(x) \equiv x + 2$$

$$g(x) \equiv 3 * x$$

$$h(x) \equiv f(g(x)) \equiv (3 * x) + 2$$

Funções matemáticas

Formas funcionais

- **Aplicar-para-todos** (denotada por **α**) é uma forma funcional que:
- Recebe **uma única função** como parâmetro;
- Aplica essa função a **cada elemento** de uma lista ou sequência;
- Coleta os resultados em uma nova lista.
- Exemplo:

$$\mathbf{h}(x) \equiv x * x$$

$$\boldsymbol{\alpha}(h, (2,3,4)) \rightarrow (4, 9, 16)$$

- Em linguagens de programação, essa operação é conhecida como **função mapa (map)**.

Fundamentos das linguagens funcionais

- O objetivo do projeto de uma **linguagem de programação funcional** é **imitar as funções matemáticas** o máximo possível.
- Isso leva a uma **estratégia diferente** de solução de problemas em relação às linguagens imperativas.
- Em uma linguagem imperativa:
  - Uma expressão é avaliada e o resultado é **armazenado** em uma posição de memória (variável).
  - Isso caracteriza o uso de **sentenças de atribuição** e a dependência de **estado**.

Fundamentos das linguagens funcionais

- Em linguagens imperativas e de montagem, resultados intermediários de expressões são **armazenados em memória**.
- Exemplo:

$$(x + y) / (a - b)$$

  - Primeiro calcula-se (x + y) e armazena-se o valor.
  - Depois avalia-se (a - b) e então o quociente.
- Em linguagens de alto nível, o **compilador** gerencia esse armazenamento.
- Já em linguagens funcionais **puras**, não há variáveis nem atribuições
  - **não há estado**.

Fundamentos das linguagens funcionais

- Em linguagens imperativas, programar é dar instruções passo a passo, alterando o estado da memória.
- Exemplo:

```
int x = 2;
x = x + 3;
```

- Aqui, **x** representa uma posição na memória.
- Ao executar x = x + 3, o valor antigo é substituído
  - **o estado do programa muda**.

Fundamentos das linguagens funcionais

- Em linguagens funcionais puras, isso não acontece.
- Em vez de alterar valores, criamos **ligações imutáveis (bindings)**.
- Exemplo em Haskell:

```
x = 2
y = x + 3
```

- Aqui, **x** é definido como 2 e nunca será outra coisa.
- **y** é definido como o resultado de x + 3, que será sempre 5.
- Essas expressões são **declarações matemáticas**, não instruções de execução.

Fundamentos das linguagens funcionais

- Em linguagens funcionais, **uma ligação imutável (binding) é uma ligação fixa entre um nome e um valor**.
- Essa ligação é criada uma única vez e nunca muda.

```
a = 10
b = a * 2
c = b + 5
```

- Durante a execução, o compilador substitui cada nome pelo valor correspondente.
  - Assim, c é avaliado como 10 * 2 + 5 = 25.
- Não há etapas intermediárias nem reatribuição.

Fundamentos das linguagens funcionais

- Em linguagens imperativas, o **estado** é o conjunto de valores atuais das variáveis.
- Cada linha de código altera esse estado.

```
int total = 0;
for (int i = 1; i <= 3; i++) {
    total += i;
}
printf("%d", total); // 6
```

- O valor de **total** muda a cada iteração.

Fundamentos das linguagens funcionais

- Em linguagens funcionais, **não há estado mutável**.
- Cada nova operação cria **um novo valor**, sem alterar o anterior.
- Exemplo em Haskell:

```
total = sum [1, 2, 3]
```

- Aqui, sum apenas **mapeia a lista** para o resultado 6, não há variável sendo atualizada.
- O programa descreve *o que é* o resultado, não *como obtê-lo passo a passo*.

Fundamentos das linguagens funcionais

- Em linguagens funcionais puras, **nomes não podem ser redefinidos**.
- Cada nome é **uma ligação (binding) única**, que expressa uma **igualdade matemática permanente**.
- Por isso, o código abaixo **gera erro** em Haskell:

```
total = sum [1, 2, 3]
total = sum [1, 2, 5]
```

- O compilador retorna algo como:
  - Multiple declarations of ‘total’

Fundamentos das linguagens funcionais

- Isso ocorre porque **total já está definido**
  - não é uma variável, e sim uma **declaração de igualdade**.
- Se quiser outro valor, é preciso **criar outro nome ou escopo**:

```
total1 = sum [1, 2, 3]
total2 = sum [1, 2, 5]
```

- Essa regra reflete o princípio da **transparência referencial**:
  - uma vez definido, um nome sempre se refere ao mesmo valor.

Fundamentos das linguagens funcionais

- Um programa puramente funcional:
  - **Não usa variáveis** nem sentenças de atribuição.
  - **Não possui estado**; cada função depende apenas de seus **parâmetros**.
  - Repetições são expressas por **recursão**, não por iteração.
- Os programas são **definições de funções** e **aplicações de funções**.
- A execução de uma função sempre produz **o mesmo resultado** para os mesmos parâmetros.
  - Esse princípio é chamado de **transparência referencial**.

Fundamentos das linguagens funcionais

- **Transparência referencial** significa que:
  - Uma função sempre devolve o **mesmo resultado** para as mesmas entradas.
  - Ela **não depende de variáveis externas** nem causa efeitos colaterais.
- Isso torna a **semântica** (significado) das linguagens funcionais **muito mais simples** do que a das imperativas.
- Também facilita o **teste e depuração**, pois cada função pode ser verificada isoladamente.

Fundamentos das linguagens funcionais

- Um **programa puramente funcional** não tem estado, nem na:
  - **Semântica operacional:** descreve *como* o programa executa passo a passo.
  - **Semântica denotacional:** descreve *o que* o programa significa, associando expressões a valores matemáticos.
- Assim, a execução é **determinística**:
  - mesmas entradas: mesmos resultados.
- Essa previsibilidade é uma das maiores vantagens do paradigma funcional.

Fundamentos das linguagens funcionais

- Uma linguagem funcional fornece:
  - Um **conjunto de funções primitivas**;
  - **Formas funcionais** (para compor funções complexas);
  - Uma **operação de aplicação de função**;
  - Estruturas para representar dados e parâmetros.
- Se bem projetada, precisa de **poucas funções primitivas**.

Fundamentos das linguagens funcionais

- A primeira linguagem funcional, **Lisp**, possuía uma sintaxe própria, muito diferente das linguagens imperativas.
- Linguagens posteriores (como **ML**, **Haskell**, **F#**) adotaram sintaxe mais próxima das imperativas.
- Embora **Haskell** seja considerada funcional pura, muitas linguagens funcionais modernas incluem recursos **imperativos**, como variáveis mutáveis e atribuições ocasionais.

Fundamentos das linguagens funcionais

- Conceitos das linguagens funcionais influenciaram diversas linguagens imperativas:
  - **Avaliação tardia** (lazy evaluation);
  - **Subprogramas anônimos** (funções lambda).
- Mesmo linguagens originalmente imperativas hoje adotam elementos funcionais.

Suporte funcional em linguagens imperativas

- Linguagens **imperativas** normalmente oferecem **suporte limitado** à programação funcional.
- As primeiras linguagens **não suportavam funções de ordem superior**, o que restringia a aplicação de conceitos funcionais.
- O **interesse crescente** em técnicas funcionais levou ao **suporte parcial** em linguagens imperativas, como **JavaScript, Python, Ruby, Java e C#**.

Suporte funcional em linguagens imperativas

- **Funções anônimas** (sem nome) são **expressões lambda**, hoje comuns em várias linguagens.
- Funcao anônima em JavaScript:

```
function        (parâmetros-formais) {
 corpo
}
```

Suporte funcional em linguagens imperativas

- **C#** define **expressões lambda** como instâncias de objetos representando funções.
- Sintaxe:

parâmetro(s) => expressão

- Se houver mais de um parâmetro, usa-se parênteses:

(a, b) => a + b

Suporte funcional em linguagens imperativas

- Em C#, **expressões lambda** podem ser passadas como **parâmetros reais** para métodos.
- Exemplo: método FindAll, que filtra elementos de um vetor

```
int[] numbers = {-3, 0, 4, 5, 1, 7, -3, -6, -9, 0, 3};
int[] positives = Array.FindAll(numbers, n => n > 0);
// Agora, positives é {4, 5, 1, 7, 3}
```

Suporte funcional em linguagens imperativas

- Expressões lambda em Python

```
lambda a, b : 2 * a - b
```

- O corpo da função vem **após os dois pontos**.

Suporte funcional em linguagens imperativas

- Python também suporta **funções de ordem superior**
- **filter**() e **map**() aplicam uma função a todos os elementos de uma sequência.
- Exemplo:

```
map(lambda x: x ** 3, [2, 4, 6, 8])
```

- Resultado: [8, 64, 216, 512]

Suporte funcional em linguagens imperativas

- Python suporta **aplicações parciais**, que fixam parte dos parâmetros de uma função:

```
from operator import add
from functools import partial

add5 = partial(add, 5)

add5(15)
```

- add5(15) retorna 20

Suporte funcional em linguagens imperativas

- A incorporação de **recursos funcionais** em linguagens imperativas permite:
  - Melhor modularidade e clareza;
  - Maior reuso de código;
  - Estilo declarativo mais próximo da matemática.
- Exemplos modernos:
  - **Python:** lambda, map, filter, reduce, partial;
  - **JavaScript:** =>, map(), filter(), reduce();
  - **C#:** expressões lambda com Func<T>;
  - **Java 8+:** lambdas e Stream API;
  - **Ruby:** lambda e Proc.

Suporte funcional em linguagens imperativas

- A incorporação de **recursos funcionais** em linguagens imperativas permite:
  - Melhor modularidade e clareza;
  - Maior reuso de código;
  - Estilo declarativo mais próximo da matemática.
- Exemplos modernos:
  - **Python:** lambda, map, filter, reduce, partial;
  - **JavaScript:** =>, map(), filter(), reduce();
  - **C#:** expressões lambda com Func<T>;
  - **Java 8+:** lambdas e Stream API;
  - **Ruby:** lambda e Proc.

Comparação com linguagens imperativas

- A semântica funcional é mais simples que a imperativa, pois:
  - Não há variáveis nem efeitos colaterais a representar;
  - As funções são avaliadas de forma puramente matemática (transparência referencial).
- Algumas pesquisas (Wadler, 1998) sugerem que:
  - Programas funcionais podem ter **até 10% do tamanho** dos programas imperativos equivalentes;
  - Ou que suas soluções podem ter **cerca de 25% do tamanho** das versões imperativas.
- Essa concisão permite ganhos de **clareza e expressividade**, embora não signifique automaticamente maior produtividade.

Comparação com linguagens imperativas

- Linguagens imperativas foram projetadas para computadores com **arquitetura de von Neumann**, baseadas em **atribuições e variáveis mutáveis**.
- Já as linguagens funcionais **imitam funções matemáticas**, o que nem sempre se ajusta perfeitamente à arquitetura de hardware tradicional.
- Ainda assim, linguagens funcionais oferecem **vantagens de legibilidade e simplicidade**.

Comparação com linguagens imperativas

- Exemplo imperativo em C:
  - Considere uma função que computa a soma dos cubos dos primeiros n positivos inteiros

```
int sum_cubes(int  n){
  int sum = 0;
  for(int index = 1; index <= n; index++)
    sum += index * index * index;
  return sum;
}
```

- Exemplo típico de **controle iterativo** e uso de **variáveis mutáveis**.
- Embora eficiente, é mais **verboso** e **menos legível**.

Comparação com linguagens imperativas

- Exemplo funcional em Haskell:

```
sumCubes n = sum (map (^3) [1..n])
```

- Essa versão executa os seguintes passos:
  - Cria a lista [1..n];
  - Usa **map** para aplicar a função de cubo a cada elemento;
  - Soma os valores resultantes com sum.
- **Sem variáveis nem loops explícitos.**
- O código é **mais próximo da definição matemática** do problema.

Comparação com linguagens imperativas

- Programação concorrente é **mais difícil** em linguagens imperativas:
  - É preciso sincronizar tarefas que compartilham variáveis.
- Em linguagens funcionais:
  - As funções são **independentes e puras** (sem variáveis globais);
  - As chamadas são **avaliadas de forma isolada**, facilitando a **execução paralela**.
- Essa ausência de estado torna a programação funcional **naturalmente segura para concorrência**.

Comparação com linguagens imperativas

- A simplicidade sintática e semântica das linguagens funcionais:
  - Reduz erros e torna o raciocínio mais matemático e previsível;
  - Facilita o paralelismo e o reuso de funções.
- Em contrapartida:
  - Linguagens funcionais podem parecer **estranhas ou difíceis** para programadores acostumados com linguagens imperativas;
  - Entretanto, após o domínio inicial, tornam-se **altamente expressivas e poderosas**.

Paradigmas de Linguagens de Programação

Linguagens de Programação Funcional