Algoritmos Funcionais

CAPÍTULO 1

FUNDAMENTAÇÃO INICIAL

Este capítulo descreve de forma introdutória detalhes sobre os paradigmas computacionais mais comuns existentes, destacando-se imperativo, declarativo, concorrente e paralelo. São indicados aspectos históricos que norteiam o surgimento do paradigma declarativo funcional e os precipícios que esse paradigma se fundamenta. É apresentada uma rápida revisão de conceitos matemáticos fundamentais à programação funcional como álgebra, aritmética, equações, inequações, conjuntos e funções.

1.1 PARADIGMAS COMPUTACIONAIS

O desenvolvimento de aplicações computacionais nos primórdios da era da computação e mais precisamente por volta da década de 1940 era, em sua maioria, realizado de maneira intuitiva e sem grandes preocupações técnicas. O desenvolvimento de software era produzido no próprio ambiente para uso local com o objetivo de solucionar problemas internos, usado, normalmente, pela própria equipe de desenvolvimento.

Com o passar dos anos e a mudança do eixo de uso das aplicações computacionais do ambiente interno para o ambiente externo

o desenvolvimento de aplicações computacionais tornou-se mais complexo exigindo a adoção de sofisticadas técnicas de desenvolvimento e estilos de programação diferenciados. Assim, surgiram basicamente quatro grupos ou categorias de macroparadigmas computacionais: *imperativo*, *declarativo*, *concorrente* e *paralelo*. Um paradigma computacional caracteriza-se em ser um método organizado e formal de realização de tarefas dentro de certo programa que não está relacionado diretamente com a linguagem em si, mas com a forma de resolver um problema do ponto de vista computacional. A forma de classificação (imperativo, declarativo concorrente ou paralelo) afeta a forma de como um programa pode ser desenvolvido. Isso significa dizer que um paradigma de programação não está para as linguagens de programação e sim o inverso: as linguagens de programação são escritas para atender a certo grau de um ou mais paradigmas computacionais.

O *paradigma imperativo* é subdividido nos modelos de desenvolvimento *procedimental* (modular ou estruturado, desenvolvido a partir de sub-rotinas, variáveis locais, variáveis globais e passagens de parâmetro), *orientado a objetos* (estruturado a partir de classes com campos membros de dados, heranças, objetos e sub-rotinas membro na forma de métodos), *orientado a eventos* (ações automáticas executadas a partir de alguma ocorrência operacional definida em um comando ou ação de uma linguagem de programação) e *programação genérica* (definição de estruturas que podem ser aplicadas a qualquer tipo de dado).

O *paradigma imperativo* é, normalmente, utilizado no desenvolvimento de aplicações comerciais e/ou industriais. Para seu uso, o profissional de programação necessita de conhecimentos matemáticos focados em ações aritméticas básicas, tais como: adição, subtração, multiplicação, divisão, potenciação e radiciação. Seu uso baseia-se na definição de programas que passam ordens a partir de um conjunto de instruções ordenadas, que especificam um método de resolução como uma receita com o objetivo de chegar a solução do problema. As linguagens imperativas têm foco no funcionamento

dos computadores em si, por ser uma abstração do funcionamento da arquitetura Von-Neumann (lê-se *fon nóiman*) caracterizada principalmente pelos componentes de *memória principal* (onde se armazena o conjunto de dados e código do programa) e *unidade central de processamento* (componente responsável por armazenar temporariamente dados, realizar operações aritméticas ou lógicas e acesso a memória principal).

O *paradigma declarativo* é subdividido nos modelos de desenvolvimento *lógico* (avaliação de predicados lógicos e regras de inferência), *funcional* (manipulação de funções de uma perspectiva eminentemente matemática), *reativo* (uso de fluxos de dados assíncronos ao estilo de grafos) e *descritivo* (uso de fluxo de dados com propagação constante de mudanças). O paradigma declarativo é, normalmente, utilizado no desenvolvimento de aplicações científicas, mas está ganhando espaço no desenvolvimento de aplicações comerciais e industriais. Para seu uso o profissional de programação necessita possuir conhecimentos matemáticos mais apurados além das noções aritméticas e algébricas básicas, como manipulação de conjuntos e funções. De maneira mais ampla, o conhecimento matemático mais adequado para uso desse paradigma vem do estudo da *matemática discreta*. Seu uso baseia-se na definição do que será realizado a partir de poucos elementos de dispersão, com os quais se busca uma solução mais abstrata do problema. O paradigma declarativo é focado na funcionalidade de expressões matemáticas, similar ao uso de calculadoras, baseado no que fazer sem se preocupar em como fazer, Bird & Wadler (1988, p. 1).

O *paradigma concorrente* é definido a partir da execução de duas ou mais tarefas que podem ser iniciadas e encerradas em intervalos de tempos diferentes, sobrepondo-se uma a outra não significando que tais tarefas são executadas ao mesmo tempo, podendo fazer uso em conjunto de operações a partir dos paradigmas *imperativo* e *declarativo*.

O *paradigma paralelo* é definido a partir da execução simultânea de duas ou mais tarefas que devem ser iniciadas e encerradas ao

mesmo tempo, necessitando de processadores de múltiplas *cores* ou múltiplos processadores para que os processos ou *threads* sejam executados ao mesmo tempo, podendo fazer uso em conjunto dos paradigmas *imperativo* e *declarativo*.

A Figura 1.1 demonstra o mapeamento básico de uma estrutura de distribuição de paradigmas computacionais.

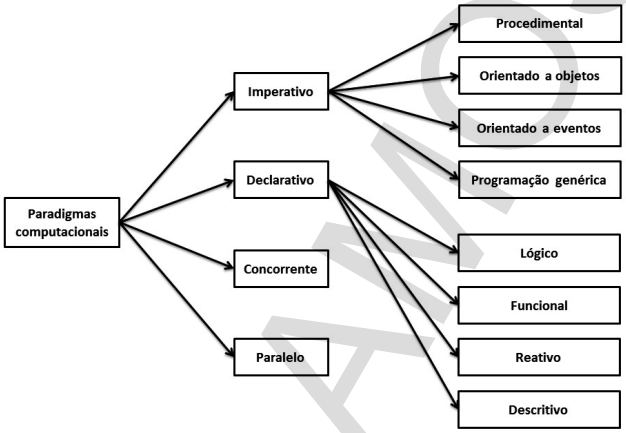

Figura 1.1 — Estrutura básica dos paradigmas computacionais.

Cada paradigma possui, de certa forma, as linguagens que o fazem mais conhecidos no meio em que são usados: *imperativo procedimental* (Pascal, C e C++), *imperativo orientado a objetos* (SmallTalk, Eiffel, C++, Java e C#), *imperativo orientado a eventos* (Visual Basic, Object Pascal, C# e Javascript relacionados ao uso de formulários), *imperativo genérico* (PolyP e C++ com *templates*), *declarativo lógico* (Lisp e Prolog), *declarativo funcional* (Lisp, Logo, Haskell, Erlang, Hope, OCaml e Elixir), *declarativo reativo* (Akka, Flapjax e Lucid), *declarativo descritivo* (HTML e SQL), *concorrente* (Ada, Erlang, Go, Rust e Elixir) e *paralelo* (CRAFT, MP Fortran, HPC++ escrita na linguagem C++ e Elixir).

Apesar de alguma preferência de uso dos paradigmas imperativo e declarativo (mais comuns) respectivamente para aplicações

comerciais e industriais e/ou científicas, nada impede de se fazer uso desses modelos de forma combinada para qualquer tipo de aplicação. Diversas linguagens de programação operam a partir de múltiplos paradigmas, dando suporte à programação declarativa, imperativa, paralela e concorrente como, por exemplo, as linguagens Rust, Lua, Python, C++, entre outras.

1.2 ASPECTOS HISTÓRICOS

A partir de uma visão conceitual sobre os paradigmas computacionais é interessante entender como tudo isso ocorreu desde a década de 1940 quando do lançamento do primeiro computador eletrônico ENIAC (`Electronic Numerical Integrator and Computer`) no ano de 1946. O ENIAC começou a ser desenvolvido em 1943 com o objetivo de ser usado para cálculos de balística durante a II Guerra Mundial (1939–1945), o que acabou por não acontecer. Com o término da II Guerra Mundial, os Estados Unidos da América (EUA) iniciaram com a União das Repúblicas Socialistas Soviéticas (URSS) a chamada Guerra Fria que durou até o ano de 1991, quando a URSS deixou de existir ocorrendo a separação dos países membros. O período da Guerra Fria se caracterizou por rivalidades nas áreas: militar, econômica, política, ideológica e científica, o que trouxe, inclusive, grandes avanços na área de computação culminando no que se tem na atualidade.

Um dos grandes episódios durante a Guerra Fria foi o desenvolvimento da corrida espacial entre EUA e URSS na busca da supremacia na exploração espacial durante os anos de 1957 a 1975. Isso fez com que, em 1957, a URSS colocasse na órbita terrestre o primeiro satélite artificial chamado *`Sputnik 1,`* e em 1961 patrocinasse o primeiro voo orbital tripulado pelo cosmonauta Yuri Alekseievitch Gagarin com a nave *`Vostok I`*. Em resposta, os EUA, em 1969, patrocinaram a operação de primeira alunagem com a nave *`Apollo 11`* tripulada pelos astronautas Neil Armstrong, Edwin Aldrin e Michael Collins.

O uso de computadores por empresas comerciais e industriais tem sua origem durante a década de 1950. Nesse período, o desenvolvimento de programas, ainda mesmo que precário, era produzido com linguagens de baixo nível (linguagens de máquina e assemblys). Durante 1954 é lançada pela equipe de cientistas da computação da empresa IBM (`International Business Machines`), chefiada por John Backus, a primeira linguagem de alto nível em estilo imperativo semiestruturado chamada FORTRAN (`FORmula TRANslation`) para aplicações matemáticas. Depois em 1958 (após o fim da II Guerra Mundial e início da Guerra Fria) é lançada a primeira linguagem funcional em estilo declarativo no MIT (`Massachusetts Institute of Technology`) por John McCarthy chamada Lisp.

A linguagem Lisp no seu desenvolvimento tem como influência diversos trabalhos produzidos por vários cientistas, destacando-se: Haskell Brooks Curry (teoria da lógica combinatória, 1927), Kurt Gödel (teoria das funções recursivas, 1931), Alan Turing (máquina de Turing, 1936), Alonzo Church (teoria do cálculo lambda, 1936), Stephen Cole Kleene (teoria da computabilidade: funções computáveis, 1938) e Emil Leon Post (máquina de estado finito, 1943), que por sua vez são a base de fundamentação do *`paradigma declarativo funcional`*. O desenvolvimento da linguagem Lisp decorreu das necessidades de alguns cientistas da computação estarem realizando pesquisas nas áreas de inteligência artificial, cálculo simbólico, comprovações de teoremas, sistemas baseados em regras e processamento de linguagem natural.

Apesar de ser o paradigma declarativo funcional o segundo paradigma mais antigo desenvolvido, acabou não sendo muito utilizado devido a um problema colateral de sua funcionalidade: o alto consumo de memória, tornando seu uso apenas viável mais recentemente. Esse problema decorreu do fato de, que no ano de 1957, início da corrida espacial, *`1 megabyte`* de memória custar algo em torno de US$400.000.000,00. Quando em 1975, com a popularização dos microprocessadores e o fim da Guerra Fria, *`1 megabyte`* de memória passou a custar algo em torno de US$50.000,00. Em

1985, *1 megabyte* de memória passou a ter um custo aproximado de US$1.500,00 chegando ao preço médio de US$10,00 em 2019.

A queda de preço no custo das memórias proporcionou computadores com maior capacidade de processamento tornando o uso de linguagens funcionais mais viáveis, o que justifica o aumento da popularidade e interesse pelo *paradigma declarativo funcional*.

Outro fato curioso em relação ao uso de linguagens funcionais é que devido ao seu apelo matemático além do uso em aplicações científicas, essas linguagens foram, por muitos anos, usadas no contexto acadêmico e mais recentemente estão sendo usadas em empreendimentos comerciais e industriais. Isso ocorre, não só devido ao baixo custo das memórias, mas também a capacidade dos microprocessadores mais recentes terem suporte as ações de processamento com paralelismo/concorrência e a necessidade de desenvolver programas mais dinâmicos com maior facilidade de manutenção, características oferecidas por linguagens funcionais. Mas há ainda, o fato de que diversas linguagens de programação mais tradicionais como C++, Java, C#, JavaScript, Ruby, Go, Python, entre outras vêm ao longo de seus desenvolvimentos adotando capacidades de operações funcionais, até então não existentes. Além das linguagens clássicas, diversas outras linguagens são lançadas com adoção direta do paradigma declarativo funcional como: Elixir, Clojure, Scala, Rust, Elm, entre outras.

1.3 PRINCÍPIOS DE PROGRAMAÇÃO FUNCIONAL

A *programação funcional* é uma atividade de ação lógica baseada no *paradigma declarativo funcional* que foca sua ação em resultados, ou seja, no que deve efetivamente ser computado e não no processo de como fazer. Esse estilo de programação trata programas como expressões e transformações que modelam fórmulas matemáticas, na forma de *expressões matemáticas*, categorizando os problemas computacionais de maneira diferente das linguagens imperativas como

é indicado por Ford (2014, p. 12). O paradigma declarativo funcional, em vez de usar instruções, usa expressões no sentido de produzir valores como respostas, trata-se de uma forma de identificar padrões e extrair funções que resolvam problemas computacionais.

As *expressões matemáticas* são o cerne da programação funcional tendo por característica a capacidade de descrever, sempre, um valor a partir de outros valores fornecidos para a operação, se assim existirem. Uma expressão matemática pode conter a definição de incógnitas que representam quantidades desconhecidas chamadas, em programação, de variáveis, nesse caso, imutáveis. Todo matemático entende que as incógnitas de uma expressão matemática não variam seus valores dentro do contexto de sua operação após serem definidos, pois devido a sua natureza denotam sempre a mesma quantidade dentro dos limites de ação de certa expressão matemática como é apontado por Bird & Wadler (1988, p. 4).

Em essência a programação funcional considera o uso e a avaliação de funções, computação simbólica e processamento de listas na forma de expressões matemáticas que evitam estados mutáveis de dados (funções puras) sem efeitos colaterais, pois o resultado de retorno de uma função será sempre o mesmo se a entrada também o for (transparência referencial), auxiliando o desenvolvimento de programas mais robustos, de fácil manutenção e fáceis de serem testados. Esse paradigma define a computação como uma série de expressões matemáticas baseadas no uso de funções e conjuntos. Os programas são escritos de forma concisa seguindo uma especificação matemática, uma vez que a estrutura principal de controle de um programa é uma função (ponto de vista matemático) e o foco é direcionado no que se deve fazer e não em como deve ser feito.

É um modelo que baseia a lógica de programação a partir de um conjunto de funções ou de expressões matemáticas. Assim sendo, a função matemática `f(x) = x + 1` diz que dado um argumento *X* em `f(x)`, seu domínio (conjunto de entrada), há sempre o retorno do valor sucessor de *X*, sendo este *X + 1*, seu contradomínio (conjunto de saída). Se fornecido para a função o argumento `5` o retorno será `6`, pois

`f(5) = 6`. Desta forma, um programa funcional segue uma estrutura declarativa e não imperativa, onde cada elemento de um conjunto domínio usado como entrada de uma função `f(x)` terá como saída um valor de elemento do conjunto contradomínio (imagem) `x + 1`.

A ação de aplicação lógica na programação funcional se caracteriza por ser a aplicação de métodos de raciocínio, ou seja, fornece regras e técnicas para definir se certo argumento é válido ou não. O raciocínio lógico usado na ciência matemática serve para verificar e validar teoremas e o raciocínio lógico usado na ciência da computação serve para verificar e testar se os programas estão ou não corretos.

No paradigma da programação funcional, uma função pode ser usada como argumento de outra função. Devido a essa característica, a programação funcional não é operada com variáveis de estado (variáveis que sofrem alterações de seus valores ao longo da execução de um programa a qualquer momento e por qualquer motivo) e sim com variáveis imutáveis. Isto posto, significa que se um valor é associado a uma variável no contexto funcional será este imutável até a conclusão do processamento do programa.

A programação funcional é realizada a partir da separação das estruturas de dados e das funções que operam sobre essas estruturas, sendo referenciadas como listas ao estilo máquinas de estado finito. As funções como elementos de primeira ordem podem receber outras funções como argumentos. Isso permite realizar o armazenamento de funções em listas e usá-las em seu processamento. Esse efeito é chamado *`callback`* (função passada a outra função como argumento).

As funções que recebem ou retornam como argumentos outras funções são chamadas de funções de ordem superior, funções de primeira ordem ou cidadãos de primeira classe (forma funcional), sendo esta uma maneira de se fazer a reutilização de código dentro do paradigma descritivo funcional.

A programação funcional não se utiliza de laços como se faz no paradigma imperativo. Para ações de repetições é usado o conceito de

recursividade, onde uma função por sua própria natureza de definição pode chamar a si mesma até que certa condição seja estabelecida.

No paradigma imperativo estruturado ou orientado a objetos, os programas são baseados no uso de variáveis mutáveis (variáveis de estado), atribuições de valores em variáveis que podem ser alteradas a qualquer instante e estruturas de blocos com execução de laços e a definição de sub-rotinas que operam aos estilos de procedimento e/ou função. Já no paradigma declarativo funcional os programas não se utilizam de atribuições em variáveis e sim de asserções, as variáveis são imutáveis (uma vez estabelecido um valor este não é alterado até a conclusão da função) e os programas possuem unicamente a definição de funções.

No sentido de demonstrar as diferenças básicas existentes entre os paradigmas imperativo e declarativo considere o uso de uma função chamada `quadrado(n)` que apresente o resultado do quadrado do valor indicado como parâmetro como exemplificado em Bird & Wadler (1988, pp. 2–5).

Primeiramente note os detalhes do estilo imperativo codificado a partir de um código expresso na forma PDL (`Program Design Language`) ao estilo *`português estruturado`*.

```
função quadrado (n : inteiro) : inteiro
início
  quadrado = n ^ 2
fim
```

Agora observe os detalhes de uma ação similar codificada ao estilo declarativo funcional a partir de um código expresso ao estilo *`português funcional`*.

```
quadrado (número) >> número
quadrado (n) << n ^ 2
```

No exemplo funcional, há na primeira linha do código a definição do cabeçalho da função `quadrado`, de seu argumento do tipo `numérico`

entre parênteses representando a entrada de valores na função e a definição do retorno de tipo `número` ao lado direito do símbolo de extração "`>>`". A segunda linha indica a ação efetiva a ser realizada pela função ao constituir a variável imutável local *N* como parâmetro operacional efetivo da função sendo a correspondência de padrão da função. Ao lado direito do símbolo "`<<`" (símbolo de asserção) é definido o resultado de retorno da função como o quadrado do valor definido para *N*. Uma vez definido o valor para a variável *N*, na chamada da função, esse valor não sofrerá nenhuma outra alteração. Observe que a versão funcional é mais compacta que a versão imperativa.

Os programas do paradigma funcional possuem como característica avaliarem certa expressão matemática reduzindo-a ao seu equivalente mais simples. Por exemplo, a redução da expressão matemática para cálculo de quadrado a partir da função `quadrado(3 + 4)` pode internamente no computador, como apontam Bird & Wadler (1988, p. 5), ocorrer como:

```
quadrado (3 + 4) >> (3 + 4) ^ 2 [adição de 3 com 4 antes da exponenciação]
                 >> 7 ^ 2       [quadrado de 7]
                 >> 49          [resultado da exponenciação]
```

A ocorrência *`[adição de 3 com 4 antes da exponenciação]`* refere-se ao uso da adição indicada como parâmetro *`3 + 4`* para o argumento *N* da função `quadrado(n)`. A ocorrência *`[quadrado de 7]`* se refere à aplicação da regra *`7 ^ 2`* que resulta em `49` como retorno da operação da função.

Nem tudo são flores, a programação funcional possui detalhes negativos a serem considerados, destacando-se ser um paradigma menos conhecido e exigir grande mudança no modo de pensar. Por ser um modelo de mais alto nível usar ações de baixo nível (muitas vezes necessária) é impossível ou oneroso para o processamento da máquina. Outro ponto negativo é que as execuções de programas funcionais consomem, em média, mais memória e, podem em algumas linguagens, serem executados mais lentamente que

programas imperativos. Assim sendo, considere alguns pontos de desvantagem apontados por Bhadwal (2019):

- Valores imutáveis combinados com recursão podem levar a uma redução no desempenho;
- Em alguns casos escrever funções puras reduz a legibilidade do código;
- Embora seja fácil escrever funções puras, combiná-las com o restante do programa, bem como com as operações de entrada e saída poderá tornar a tarefa de uso mais difícil;
- Escrever programas com recursividade em vez de usar laços para o mesmo tipo de ação pode ser uma tarefa mais difícil.

Os detalhes aqui indicados não são necessariamente encontrados em todas as linguagens funcionais, uma vez que, como já exposto, a programação funcional é uma estratégia de desenvolvimento e não necessariamente uma linguagem em si. Dessa forma, mesmo em linguagens de programação sem suporte ao paradigma declarativo funcional é possível fazer uso de alguns desses detalhes.

1.4 LINGUAGENS FUNCIONAIS

O objetivo de uma linguagem funcional é imitar a implementação de funções matemáticas ao máximo possível, segundo Sebesta (2018, p. 636). Assim sendo, a partir do entendimento de que o paradigma declarativo funcional é um método de operação no desenvolvimento de programas de computadores e não é em si a linguagem de programação, cabe descrever sobre as ferramentas que atendem a esse paradigma. É importante ressaltar que a programação funcional é fundamentada basicamente em duas perspectivas: uso de funções puras e a eliminação de efeitos colaterais ocorridos com o uso de variáveis de estado, acrescentando-se a isso o fato de as linguagens funcionais fazerem uso de estruturas de dados baseadas em conjuntos (listas, mapas, dicionários, tuplas, conjuntos, entre outros).

Em relação à disponibilidade de linguagens funcionais, encontram-se as chamadas linguagens puras e impuras (ou híbridas) que operam com múltiplos paradigmas e que dão suporte ao modelo funcional. O problema da aprendizagem de qualquer linguagem de programação é o desenvolvimento do chamado *preconceito tecnológico* que é adquirido pelo estudante, muitas vezes dentro da sala de aula, ao achar que a linguagem que está aprendendo é a melhor opção que existe e que o paradigma em uso é melhor que os demais existentes. No contexto de desenvolvimento é necessário, na maior parte das vezes, ter a mente aberta para combinar paradigmas e tirar o máximo proveito deles, neutralizando os pontos negativos que possam existir entre esses paradigmas. Lembre-se de que na área de desenvolvimento de software se aplica sem exceção a célebre frase de Aristóteles: "A soma das partes torna-se maior que o todo."

As linguagens funcionais puras possuem como característica o uso de funções operacionalizadas a partir de variáveis imutáveis não gerando efeitos colaterais, não dependendo de variáveis globais ou locais, utilizando-se de recursividade para a execução de laços e realizando iteração em coleções de dados. Em contraposição linguagens funcionais impuras operam com entrada e saída de dados de forma explícita, utilizando-se de variáveis de estado, aceitando ações imperativas devido ao fato de serem híbridas e adotarem mais de um paradigma para a produção de programas. As operações de acesso à memória ocorrem automaticamente quando uma função é chamada.

Uma linguagem funcional típica deve dar suporte ao menos aos seguintes recursos:

- Funções de primeira classe (primeira ordem ou cidadãos de primeira classe);
- Funções de alta ordem (ordem maior, ordem superior ou forma funcional);
- Imutabilidade de dados;

- Funções puras;
- Recursividade;
- Manipulação de listas;
- Manipulação de tuplas.

O recurso *funções de primeira classe* ocorre quando uma função pode ser atribuída a uma variável, passada como valor a outra função, tratando a função passada como se fosse um dado convencional como um valor ou retornada de outras funções.

O recurso *funções de alta ordem* ocorre quando uma função pode receber uma função como argumento ou retornar uma função como seu valor.

O recurso *imutabilidade de dados* ocorre quando certa variável recebe uma atribuição de valor e após esta ocorrência não permite mais a alteração desse valor.

O recurso *funções puras* ocorre quando uma função recebe um argumento de entrada, retornando sempre o mesmo valor de saída sem ocorrência de efeitos colaterais.

O recurso *recursividade* ocorre quando uma função efetua sucessivas chamadas, controladas, a si mesma até chegar ao resultado esperado sem o uso de laços.

O recurso *manipulação de listas* ocorre a partir do uso de um conjunto de dados de mesmo tipo (dados homogêneos) delimitados entre colchetes. As linguagens que possuem essa estrutura de dados têm à disposição um rol de operações existentes para o gerenciamento desses dados. Uma lista é uma estrutura de dados mutável, podendo ter elementos acrescidos ou retirados a qualquer momento.

O recurso *manipulação de tuplas* ocorre a partir do uso de um conjunto de dados de tipos diferentes (dados heterogêneos) delimitados entre parênteses. As linguagens que possuem essa estrutura

de dados têm à disposição um pequeno rol de operações existentes para o gerenciamento desses dados. Uma tupla é uma estrutura de dados imutável não podendo dela serem retirados ou adicionados outros dados.

O conjunto de linguagens funcionais disponíveis para uso é muito grande, desde linguagens tradicionais até linguagens com lançamentos mais recentes. Segue breve lista de linguagens funcionais puras e impuras: linguagens puras: *Agda* (2007), *Elm* (2012), *Haskell* (1990), *ML* (1980), *Miranda* (1985), *Lambdascript* (2017) e *SASL* (1972); linguagens impuras: *Erlang* (1986), *Elixir* (2011), *F#* (2005), *Lisp* (1958), *Logo* (1967), *Python* (1990) e *Rust* (2010).

Resumidamente, pode-se dizer que, do ponto de vista funcional, um programa é um conjunto de definições na forma de funções, que por sua vez são formadas por associações de um rótulo de identificação e um valor que representam em si certa função.

1.5 ÁLGEBRA E ARITMÉTICA

A programação funcional tem como princípio a aplicação dos conceitos de funções de um ponto de vista matemático. Por sua vez tem também relação com a teoria de conjuntos, os quais dependem de conhecimentos relacionados à aplicação de álgebra e aritmética.

A *aritmética* é um ramo da ciência matemática que lida com elementos numéricos e as possíveis operações entre esses elementos: adição (+), subtração (-), multiplicação (.), divisão (:), potenciação e radiciação. As operações aritméticas são representadas por constantes numéricas com auxílio de *operadores matemáticos* (chamados na ciência da computação de *operadores aritméticos*), os quais estabelecem as relações numéricas entre si, por exemplo, $3 + 2 = 5$; $8 : 4 = 2$; $6 \cdot 4 = 24$ ou $7 - 6 = 1$ (como representantes das operações matemáticas básicas), entre outras possibilidades e considerando-se em sua extensão as operações para os cálculos de potenciação $3^2 = 9$ e radiciação $\sqrt{9} = 3$.

A *álgebra*, como ramo da ciência matemática, tem por finalidade generalizar as ações aritméticas a partir da configuração de expressões matemáticas na forma de *equações* que representam operações de igualdade (=), ou seja, equivalência a partir de um ou mais valores desconhecidos na forma de *incógnitas* (letras que representam números desconhecidos) chamados de *variáveis* na ciência da computação como, por exemplo, as operações `x = a + b`; `x = a : b`; `x = a · b` ou `x = a - b`. Além das equações, a álgebra generaliza ações baseadas em *inequações* que representam desigualdades (≠), ou seja, uma relação de comparação a partir dos símbolos: maior que (>), menor que (<), maior ou igual a (≥) e menor igual a (≤), indicando pelo menos um valor desconhecido na forma de incógnita como, por exemplo, as operações `x > a + 2`; `x < a` ou `x ≥ a - b`. Na ciência da computação os símbolos que estabelecem inequações são chamados de *operadores relacionais*.

Os valores representados aritmeticamente ou algebricamente são elementos que pertencem a certo conjunto numérico, podendo ser conjunto: *natural*, *inteiro*, *racional*, *irracional*, *real*, *imaginário* ou *complexo*.

Como qualquer valor numérico pode ser representado por incógnitas, é possível a partir das operações de adição (e sua oposição como subtração) e multiplicação (e sua oposição como divisão) estabelecerem relações das propriedades: *associativa*, *comutativa* e *distributiva*, considerando-se a existência de valores neutros e opostos.

Considerando as incógnitas (ou variáveis) *A*, *B* e *C* é possível estabelecer as relações de propriedades a partir das operações matemáticas básicas, onde é possível substituir o lado esquerdo pelo direito ou vice-versa. As relações de propriedades podem ser consideradas como igualdades algébricas (raciocínio equacional):

- Associativa = `(a + b) + c = a + (b + c)` e `(a · b) · c = a · (b · c)`
- Comutativa = `a + b = b + a` e `a · b = b · a`
- Elemento neutro = `a + 0 = 0 + a = a` e `a · 1 = 1 · a = a`

- Elemento oposto = `a + (-a) = 0` e `a · (1 / a) = 1`
- Propriedade distributiva = `a · (b + c) = a · b + a · c`

A partir da existência de valores algébricos e aritméticos, torna-se possível definir expressões matemáticas binárias (os operadores aritméticos são mecanismos binários de execução de cálculos) que operam sobre esses valores. Sendo este princípio, uma das bases operacionalizadas no paradigma descritivo funcional.

Se uma expressão for definida a partir de valores aritméticos, como `2 + 5 · 3`, está será uma *`expressão aritmética`* e se usado valores algébricos `a + b · c` ou valores aritméticos e valores algébricos `a + b · 3` ter-se-á uma *`expressão algébrica`*.

1.6 CONJUNTOS

Um conjunto se caracteriza por ser uma coleção de elementos de natureza homogênea ou heterogênea de diversas naturezas. Nesta obra, são considerados, por questões de praticidade, apenas o uso de conjuntos formados por valores numéricos. Um conjunto pode estar vazio `{}` ou representar um universo `U`, quando possui elementos que são relevantes a determinado contexto.

Basicamente todos os conceitos operacionalizados pela área da computação são fundamentados sobre conjuntos, sendo uma estrutura que permite agrupar elementos possibilitando a definição de estruturas mais complexas como aponta Menezes (2010, p. 25).

A representação matemática de conjuntos é feita a partir do uso de letras maiúsculas na definição de seus nomes de identificação, um símbolo de igualdade com a coleção de valores separados por vírgulas e delimitados entre chaves.

```
A = {1, 2, 3, 4, 5}
```

Na ciência da computação, a representação de um conjunto pode ser associado a uma variável (indicada tanto em letra maiúscula como em letra minúscula) podendo ser delimitado graficamente com os símbolos de chaves, parênteses ou colchetes, dependendo, é claro, da linguagem de programação em uso. Normalmente, se usam na definição computacional de conjuntos os símbolos de colchetes.

```
A = [1, 2, 3, 4, 5]
```

O tema conjuntos é estudado por um ramo matemático chamado *teoria dos conjuntos* e, neste sentido, na maioria das vezes, considera apenas os elementos de um conjunto que são relevantes ao seu contexto, o que pode fugir do contexto do universo da programação funcional.

A teoria de conjuntos determina que elementos e conjuntos possuam dois estados de relação: *pertinência* e *inclusão*. A *pertinência* se refere ao estado de certo elemento pertencer a determinado conjunto e a *inclusão* ocorre quando a relação é estabelecida entre dois conjuntos (relação binária). As operações binárias de inclusão entre os conjuntos *X* e *Y*, são:

- União (X ∪ Y)
- Intersecção (X ∩ Y)
- Complemento — diferença de conjuntos (∪ \ X ou X^c)
- Diferença simétrica (\ ou -)

A união entre dois ou mais conjuntos é a junção de todos os elementos pertencentes a cada conjunto ou de todos os conjuntos. A união dos conjuntos {1, 2, 3, 4} ∪ {3, 4, 5, 6} resulta no conjunto {1, 2, 3, 4, 5, 6}.

A intersecção entre dois ou mais conjuntos é composto por todos os elementos existentes em todos os conjuntos. A intersecção dos conjuntos {1, 2, 3, 4} ∩ {3, 4, 5, 6} resulta no conjunto {3, 4}.

O complemento de um conjunto *X* se refere aos elementos que não fazem parte do conjunto universo, ou seja, o complemento de *X* em relação a *U* (conjunto universo), sendo o resultado de *U - X*, onde o conjunto *X* é formado pelos elementos do conjunto *U* que não pertencem ao conjunto *X*. O complemento dos conjuntos {1, 2, 3, 4} \ {3, 4, 5, 6} resulta no conjunto {1, 2}, já o complemento dos conjuntos {3, 4, 5, 6} \ {1, 2, 3, 4} resulta no conjunto {5, 6}.

A diferença simétrica entre conjuntos é o resultado de todos os elementos que são membros de cada um dos conjuntos, mas não em todos eles. A diferença simétrica dos conjuntos {1, 2, 3, 4} \ {3, 4, 5, 6} resulta no conjunto {1, 2, 5, 6}. Esta operação equivale a (X ∪ Y) \ (X ∩ Y).

Por ser indicado o uso de conjuntos com valores numéricos, podem ser considerados os conjuntos de números reais (R), composto pelos conjuntos de números irracionais (I) e dos números racionais (Q). O conjunto de números racionais é composto pelo conjunto dos números inteiros (Z), que por sua vez é composto pelo conjunto de números naturais (N). Além desses conjuntos há o conjunto de valores imaginários (i) que conjugado com o conjunto de valores reais forma o conjunto de números complexos (C). Observe a Figura 1.2.

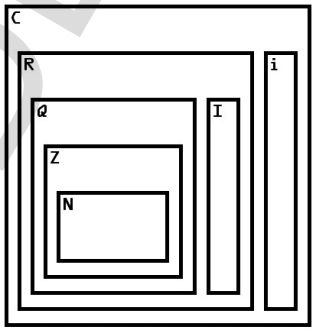

Figura 1.2 — Composição da estrutura de conjuntos.

O conjunto de números naturais (N = `natürlich`, natural no idioma alemão) serve para representar quantidades. Inicialmente o

valor zero não fazia parte deste conjunto, mas com a necessidade de representar quantidades nulas este valor foi acrescido.

O conjunto de números inteiros (`Z` = `zahl`, número inteiro no idioma alemão) surge da necessidade de representar perdas ou valores de dívidas. Este conjunto possui todos os números naturais e seus correspondentes negativos.

O conjunto de números racionais (`Q` = `quotient`, quociente nos idiomas alemão/inglês) surge da necessidade de representar partes de um todo. Este conjunto possui todos os números do conjunto `Z` com `N`.

O conjunto de números irracionais (`I` = `irrational`, irracional nos idiomas alemão/inglês) surge da obtenção de valores numéricos que não são representados a partir de frações de números inteiros e naturais, tendo infinitas casas decimais depois da vírgula (ou ponto na computação) não sendo dízima periódica.

O conjunto de números reais (`R` = `real`, real no idioma inglês) é formado por todos os números dos conjuntos racionais e irracionais. Este é um conjunto bastante importante, principalmente para a área de computação.

O conjunto de números imaginários (`i` = `imaginär`, imaginário no idioma alemão) contém todos os valores de raízes negativas cujo índice da raiz seja um valor numérico par, como o uso de bases negativas em raízes quadradas. Este conjunto não depende dos demais conjuntos.

O conjunto de números complexos (`C` = `complex`, complexo no idioma inglês) é formado a partir dos conjuntos dos números reais e imaginários. Um valor complexo é representado ao estilo `a + bi`, onde *A* é a parte real e *B* é a parte imaginária.

Nesta obra, para a representação genérica de conjuntos, do ponto de vista computacional, serão usados, por maior conveniência, valores numéricos separados por vírgulas e delimitados entre colchetes por ser este o estilo mais comum adotado nas linguagens funcionais.

1.7 FUNÇÕES

Função é um recurso usado na ciência matemática para estabelecer uma relação entre elementos de dois conjuntos. É uma forma de se estabelecer o mapeamento dos membros existentes entre um conjunto domínio com um conjunto imagem. Considerando os conjuntos *X* e *Y* de modo que o primeiro conjunto *X* possua alguma relação com o segundo conjunto *Y*, ter-se-á a definição de uma relação e, assim, a existência de uma função. Desta forma, a relação estabelecida entre os conjuntos será uma função de *X* em relação a *Y* se, e somente se, para todo elemento do conjunto *X* exista um único elemento no conjunto *Y* de modo que haja uma relação entre os elementos dos dois conjuntos.

Considerando que o conjunto *X* (domínio) seja formado pelos valores `1`, `2`, `3` e `4` e o conjunto *Y* (contradomínio) formado pelos valores `2`, `4`, `6`, `8` e `9`, tem-se no conjunto *Y*, como imagem, os valores `2`, `4`, `6` e `8` resultantes da função `f(x) = 2x`, onde *X* indicado em `f()` é um elemento pertencente ao conjunto *X* e `2x` é um elemento imagem do conjunto *Y*. A Figura 1.3 mostra a relação existente entre os conjuntos *X* e *Y*.

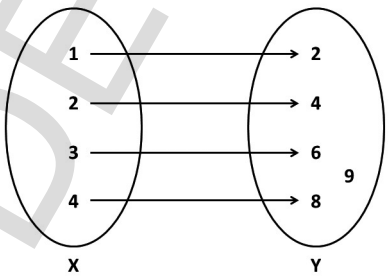

Figura 1.3 — Relação entre os conjuntos domínio X e contradomínio Y — Função injetora.

A partir da existência de uma função esta pode ser classificada como: injetora, sobrejetora ou bijetora.

Uma função é *injetora* quando cada elemento do domínio *X* associa-se a um único elemento do contradomínio *Y*, ou seja, quando para