3

Valores e Tipos

"Se una notte d'inverno un viaggiatore..."

—Italo Calvino

Conforme visto no capítulo anterior, programas são máquinas abstratas para transformação de dados, e as linguagens de programação são o recurso de que dispomos para construir programas. Os dados têm, portanto, papel central em qualquer programa, uma vez que são eles que alimentam um programa e/ou resultam dele. De uma certa maneira, podemos afirmar que os dados são a razão de existência dos programas. E, por sua vez, os programas são a razão de existência das linguagens de programação.

Dadas essas considerações, parece natural que um estudo dos princípios de linguagens de programação reserve posição especial para um estudo aprofundado da natureza dos dados. Neste capítulo e no próximo, iniciaremos esse estudo, que depois se complementará com diversas discussões ao longo de todos os capítulos seguintes.

Os dados são caracterizados por três aspectos básicos: *valores, tipos* e *variáveis*.

- *Valores* são representações simbólicas de conceitos. Se, por exemplo, o conceito representado é a temperatura que marca um termômetro, o número indicado pela coluna de mercúrio será um valor numérico .
- Os valores relevantes para a resolução de um problema são classificados segundo algum critério, e uma classe de valores recebe o nome de um *tipo*. Por exemplo, se todos os valores relevantes para resolver um problema são numéricos, os valores podem ser classificados segundo o seu tipo como naturais, inteiros e reais.
- Finalmente, como um programa geralmente é composto por mais de uma expressão, os valores dos dados precisam ser registrados – temporária ou perenemente – para passar de uma expressão para outra. Fazendo uma analogia entre nossas máquinas abstratas (os programas) e uma linha de produção industrial, cada unidade de trabalho na linha de produção é semelhante a uma expressão de um programa. Assim como o material semi-acabado precisa ser transportado de alguma maneira de uma unidade de trabalho para outra, os valores precisam ser repassados de uma expressão para outra dentro do programa. Os registros desses valores servem como repositório para passagem de valores, e eles são efetuados em *variáveis*.

No presente capítulo apresentaremos os conceitos essenciais relativos a tipos e valores. Mesmo antes de estudarmos como podemos armazenar e manipular valores para obtermos soluções para os nossos problemas, precisamos discutir a natureza dos dados que queremos tratar. Para isso, descrevemos os principais conceitos relacionados a valores e tipos. No capítulo 4 apresentaremos os conceitos necessários para construir variáveis.

3.1 POR QUE ESTUDAR TIPOS DE DADOS?

Valores são elementos que devem ser representados de alguma forma na linguagem de programação, para serem manipulados durante a execução de um programa. Do

ponto de vista da execução de um programa, os valores são elementos que podem ser avaliados, armazenados, atualizados e transmitidos durante a execução.

Os programas existem basicamente para manipular valores. É interessante podermos tratar valores semanticamente relacionados de maneira uniforme, de forma que valores "semelhantes" sejam tratados da mesma forma. Tipos são conjuntos de valores semanticamente relacionados. Quando uma operação em uma linguagem de programação é definida para um tipo como um todo, ela se aplica indistintamente a qualquer valor que pertença àquele tipo. Isso é muito mais eficiente do que definir operações para cada possível valor individualmente.

Exemplo 3.1 – Se considerarmos apenas valores, devemos definir operações sobre valores individuais: a soma dos valores 4 e 5, denotada por $4 + 5$, deveria ser definida como valor 9 (valor matemático correspondente). Da mesma forma, a soma de 4 e 6 deveria ser definida como valor 10, e assim por diante. Se os valores são individuais, o seu tratamento também será individualizado. □

Contudo, é desejável que a soma dos valores inteiros 4 e 5 seja calculada de forma semelhante à soma de quaisquer outros números inteiros nas linguagens de programação; por regras unificadas. Para isso, é necessário que elementos de natureza semelhante estejam agrupados em tipos de dados, e que definamos regras unificadas para as operações sobre tipos. Desta forma, grupos de valores são tratados, em vez de valores particulares. Esta é a razão pela qual tipos são definidos na maioria das linguagens de programação (no decorrer do livro abreviamos tipos de dados para tipos).

Os tipos devem representar os conjuntos de valores elementares que se deseja tratar nas linguagens de programação. Estes são representados por valores juntamente com as operações (pelas regras unificadas) para tratamento dos mesmos. Note que para tipos numéricos queremos, por exemplo, um tratamento aritmético, bem como realizar comparações entre valores. Isso significa que os tipos não são uma junção inadvertida de valores, mas de elementos que têm a mesma natureza e uma relação de ordem entre eles (o valor inteiro 1, por exemplo, é menor que o valor inteiro 2, e gostaríamos de ter tal idéia refletida nas linguagens).

Quando pensamos em soluções computacionais para problemas, precisamos, inicialmente, modelar o "mundo real" em elementos que possam ter seus valores representados. Desta forma, nos reportamos a valores conhecidos que designamos como representações para valores do problema do "mundo real". Grande parte dos problemas científicos, por exemplo, requerem representação de valores numéricos, os quais são na sua grande maioria números inteiros ou reais. O estudo sobre números inteiros e reais, que envolve sua representação e propriedades inerentes, antecedem o advento das linguagens de programação; eles já foram matematicamente estudados. De fato, os valores que usamos para modelar soluções de problemas científicos são inspirados nos valores matemáticos que conhecemos, e não na representação computacional dos mesmos.

Como nos reportamos a valores conhecidos para representar valores do problemas que queremos solucionar, podemos dizer que as linguagens de programação se inspiraram em valores e estruturas matemáticas conhecidas para determinar quais tipos de dados deveriam ser representados nas mesmas. No decorrer deste capítulo enunciamos os domínios e estruturas matemáticas que inspiraram a implementação dos principais tipos de dados nas linguagens de programação. A elucidação dos conceitos de tipos de dados matemáticos tem como objetivo principal mostrar a amplitude da representação de valores quando da definição de tipos nas linguagens de programação. Se conhecemos matematicamente quem são os tipos de dados que gostaríamos de representar, podemos identificar e avaliar a facilidade e expressividade de dados providas por quaisquer das linguagens de programação, independentemente de como aparecem em cada linguagem em particular.

Para efeito de estudo dos conceitos, os tipos são divididos em dois grandes grupos: os tipos *primitivos* e os tipos *compostos*, conforme detalharemos a seguir. Além destes, podemos definir tipos em termos deles próprios, os *tipos recursivos*. Na maioria dos exemplos mostrados neste capítulo, aparecem algumas variáveis, para as quais veremos os conceitos fundamentais apenas no próximo capítulo. Apesar de antecipar-mos a idéia de variáveis, a maioria dos leitores já tem tal idéia intuitiva proveniente da programação. O objetivo principal dos exemplos é facilitar o entendimento de tipos sob o ponto de vista prático das linguagens de programação.

3.2 TIPOS PRIMITIVOS

Intuitivamente, os tipos primitivos são aqueles cujos valores são "atômicos", ou seja, não podem ser desmembrados em valores mais simples. Mais tecnicamente, isso significa que a implementação de um tipo primitivo em um computador é feita necessariamente "fora" da linguagem de programação. Em outras palavras, como um tipo primitivo é um elemento "atômico" na linguagem de programação, não há como descrever sua constituição utilizando os elementos da própria linguagem.

A implementação dos tipos de dados primitivos numéricos varia entre linguagens e depende do ambiente computacional no qual a linguagem está sendo implementada. Tipicamente, um tipo inteiro é implementado como um conjunto de bits de tamanho fixo. Efetivamente, portanto, os números inteiros que "cabem" no tipo inteiro de uma linguagem de programação em geral pertencem a um intervalo inteiro simétrico em torno do zero.

Por exemplo, na linguagem C [12, 50, 55], o tipo `int` permite a representação de números inteiros dentro do intervalo $[-32.768, 32.767]$. Esse intervalo contém precisamente 65.536 valores, ou seja 2^{16} valores, que são precisamente a quantidade de sequências distintas que conseguimos construir com 16 bits. Isso tudo porque os projetistas da linguagem C decidiram que cada valor do tipo `int` ocuparia 16 bits na memória de um computador.

A implementação dos números reais segue um raciocínio similar, porém, reservando alguns bits para representar um expoente de notação científica e os bits restantes para representar a "mantissa" do valor. Dessa forma, o conceito matemático de um valor real, quando implementado em um computador através das linguagens de programação, é aproximado pelo valor representável mais próximo.

Por exemplo, na linguagem C, o tipo `float` reserva 32 bits para cada valor. Os valores que pertencem a esse tipo variam de $1{,}17549435 \times 10^{-38}$ a $3{,}40282347 \times 10^{38}$, com intervalos cada vez maiores entre os números à medida que o expoente se torna maior. Para não confundir essa representação aproximada com os números reais da matemática, é comum nos referirmos a esse tipo de dados como *números de ponto flutuante*.

Cada valor precisa de uma representação tanto nas linguagens de programação quanto no hardware. Os conjuntos de números inteiros e dos números reais dos domínios matemáticos são infinitos, e, portanto, não podem ter todos os seus valores representados em um hardware (o qual possui recursos finitos). Desta forma, os tipos matemáticos inteiro e real são parcialmente representados pelas linguagens de programação.

A escolha dos tipos primitivos para uma linguagem de programação está intimamente relacionada com os propósitos para os quais a linguagem foi criada (domínios de aplicação vislumbrados). Fortran [11, 10], por exemplo, é uma linguagem de programação que foi criada primordialmente para resolver problemas científicos. Os tipos primitivos nas primeiras implementações dessa linguagem estavam relacionados com problemas numéricos. O foco era em números reais, variações de precisão, números inteiros e números complexos. Em contrapartida, linguagens criadas para processamento de dados comerciais, tal como Cobol [9], possuem cadeias de caracteres como tipo primitivo. Algumas linguagens mais especializadas, como APL e MATLAB – voltadas para a resolução de problemas matemáticos relacionados com a álgebra de matrizes – têm *matrizes* como um tipo de dados primitivo.

3.2.1 Numéricos

Alguns tipos numéricos, primordiais para o tratamento de valores tanto em linguagens dedicadas à solução de problemas numéricos quanto ao processamento de dados comerciais, aparecem na maioria das linguagens atuais. Os tipos inteiro e real, por exemplo, aparecem com nomenclatura diferente nas várias linguagens de programação atuais.

Como cada um dos valores de um dado tipo precisa de uma representação no hardware, os números reais não podem ser representados por completo, são, portanto, representados respeitando um grau de precisão imposto pelo hardware. Por isso, É comum, nas linguagens de programação, referir-se a números de ponto-flutuante

para a representação de números reais de forma que não haja confusão entre o tipo matemático real e o tipo de dados real representado nos computadores.

Exemplo 3.2 – A linguagem Pascal [72], por exemplo, embute uma representação de números inteiros e de ponto-flutuante que são denotados por `Integer` e `Real` respectivamente. Por exemplo,

```
var i: Integer;
    r: Real;
```

`i` é uma variável que pode armazenar um valor numérico do tipo inteiro, enquanto `r` pode armazenar um valor numérico do tipo ponto-flutuante. □

Exemplo 3.3 – Na linguagem C, estes mesmos tipos são descritos pelas palavras reservadas `int` e `float` (em C existem alternativas para representar números inteiros e números de ponto flutuante, utilizando quantidades diferentes de bits. Essas alternativas serão subconjuntos ou superconjuntos dos tipos apresentados no exemplo, dependendo se elas utilizam respectivamente uma quantidade menor ou maior de bits para cada valor representado):

```
int i;
float r;
```

□

Apesar de representados por nomenclatura própria para cada uma das linguagens, estes elementos denotam os mesmos tipos de dados: inteiro e ponto-flutuante.

3.2.2 Não-Numéricos

Nem todos os problemas a serem resolvidos computacionalmente possuem natureza exclusivamente numérica, e por isso tipos que representem valores não numéricos devem também ser providos pelas linguagens. Os valores mais comuns são os valores booleanos (0 e 1), agrupados no **tipo booleano**, e os caracteres que representam símbolos padrão, agrupados no **tipo caracteres**.

Na linguagem Pascal, por exemplo, o tipo booleano é denotado por `Boolean`, enquanto o tipo caractere é denotado por `Character`.

Exemplo 3.4 – Uma variável `b` booleana e uma `c` caractere são representadas por:

```
var b: Boolean;
    c: Character;
```

□

Na linguagem C, os valores booleanos não são tipos predefinidos da linguagem, mas o tipo caractere é denotado por `char`

Exemplo 3.5 – Uma variável caractere `c` é representada em C por:

```
char  c;
```

□

Outro tipo que armazena um valor atômico são os apontadores, usados especialmente para indicar o valor de memória correspondente a uma dada variável. Quando o apontador não está servindo para referenciar qualquer variável este é assinalado com o valor indefinido `nil`, por exemplo. O conjunto de valores deste tipo corresponde ao valor `nil` juntamente a uma faixa de valores que consiste em endereços de memória.

Os apontadores foram projetados vislumbrando dois usos distintos: para se ter acesso a endereçamento direto; e para fornecer um método de gerenciamento de armazenamento dinâmico (o espaço de memória é reservado apenas quando vai ser usado efetivamente).

Exemplo 3.6 – Na linguagem Pascal, um apontador para um valor inteiro pode ser definido por:

```
var intP : ^ Integer;
```

□

Exemplo 3.7 – De forma análoga, na linguagem C, um apontador para um valor inteiro pode ser definido por:

```
int *intC;
```

□

Como o tipo apontador tem como um dos propósitos alocação dinâmica de memória, mecanismos tanto para alocar quanto desalocar espaços de memória são providos para o tipo apontador em ambas as linguagens.

3.2.3 Enumerados

Além dos tipos definidos e estudados matematicamente, alguns problemas requerem a construção de novos valores. Com este objetivo, algumas linguagens de programação fornecem mecanismos para construção de novos conjuntos de valores definidos pelos seus usuários, através da enumeração.

Em algumas linguagens de programação, tais como Pascal e C, conjuntos de dados primitivos podem também ser criados pela enumeração de seus elementos.

Exemplo 3.8 – Em Pascal, por exemplo, podemos criar um tipo (conjunto de valores) para representar os meses do ano:

```
type MesesP = (jan, fev, mar, abr, mai, jun,
               jul, ago, set, out, nov, dez);
```

□

Exemplo 3.9 – De forma análoga, podemos ter em C este novo tipo definido por:

```
enum MesesC {jan, fev, mar, abr, mai, jun,
             jul, ago, set, out, nov, dez};
```

□

Nos dois casos, os elementos que aparecem na enumeração são tratados como valores. Assim, uma vez definidos, eles podem ser usados dentro dos programas da mesma forma que quaisquer outros valores predefinidos. Além disso, eles mantêm a relação de ordem na qual foram definidos, o que permite comparação entre valores. É possível, por exemplo, atribuir o valor `jan` a uma variável definida com o tipo `MesesC`, ou ainda fazer comparações entre os valores enumerados (`jan < mar`) –

exemplos de uso de variáveis com tipos enumerados serão vistos no Capítulo 4. Para que se mantenha uma relação de ordem entre os valores da enumeração, a maioria implementa esses tipos associando valores inteiros a cada um dos valores enumerados. Na linguagem C, por exemplo, o primeiro valor da enumeração é 0, o segundo 1, etc. No caso do exemplo acima, o valor `jan` é associado ao inteiro 0, o valor `fev` é associado ao 1 e assim por diante. Como conseqüência disso, operações aritméticas também podem ser realizadas, mas esta é uma particularidade da linguagem não discutida neste capítulo.

Assim como os tipos enumerados acima, "novos" tipos podem ser criados a partir de tipos existentes. Subconjuntos de valores de alguns tipos de dados primitivos podem também ser utilizados para criar novos tipos em algumas linguagens. Pascal, por exemplo, permite a criação de novos tipos definidos a partir de inteiros:

Exemplo 3.10 – O subintervalo de valores inteiros de 1 a 31, inclusive, pode ser definido como um novo tipo:

```
type DiasP = 1..31;
```

e as operações aritméticas e relacionais definidas para inteiros estão também definidas para estes elementos, desde que os valores resultado estejam dentro do tipo definido. □

Os tipos criados como subintervalos de tipos existentes não são na realidade um novo tipo, mas apenas uma restrição do existente. Como tal, as operações definidas para o tipo estão automaticamente definidas para estes tipos restritos. Apesar de não serem novos tipos de fato, é conveniente termos estes subintervalos em vários problemas práticos. Nas linguagens Pascal e Modula-2 [73] podemos criar subintervalos de quaisquer tipos primitivos predefinidos, e a verificação das restrições podem ser realizadas em tempo de compilação, quando valores são atribuídos a variáveis, ou pelo menos em tempo de execução.

3.3 TIPOS COMPOSTOS

Como visto na Seção 3.2, a maioria das linguagens de programação oferecem tipos de dados conhecidos na matemática, tais como os números inteiros e reais, e ainda valores não numéricos. A característica que todos os tipos vistos têm em comum é o fato de que cada elemento representa um valor "atômico", e assim não podemos ter acesso a "partes" do valor. Contudo, grande parte dos problemas computacionais hoje requerem manipulação de dados que são constituídos de valores mais simples. Por exemplo, uma data qualquer do ano de 2002 deve conter pelo menos o dia e o mês, e em vários problemas práticos queremos ter acesso ao dia e ao mês separadamente.

De forma análoga aos tipos primitivos, as linguagens de programação provêem mecanismos para a criação de conjuntos de dados onde cada um dos seus elementos pode ser desmembrado em valores mais simples. Estes são os chamados **tipos de dados compostos**. Note que as linguagens visam prover diferentes formas de agruparmos valores que possam representar os dados que precisamos para as soluções dos "problemas reais". A escolha das possíveis formas de agrupar valores providas pelas linguagens de programação é também inspirada em estruturas matemáticas conhecidas (operações sobre conjuntos). Nas seções que seguem enunciamos os elementos matemáticos (produto cartesiano, união disjunta, mapeamentos e conjuntos potência) que inspiraram os agrupamentos de valores nas linguagens de programação, juntamente com exemplos ilustrativos nas linguagens C e Pascal.

3.3.1 Produto Cartesiano

Matematicamente, o produto cartesiano de dois conjuntos corresponde a todos os pares formados por valores destes conjuntos, de forma que o primeiro elemento do par pertence ao primeiro conjunto e o segundo elemento do par pertence ao segundo conjunto.

Exemplo 3.11 – O produto cartesiano dos conjuntos A e B definidos na Figura 3.1 tem como resultado o conjunto $A \times B$. O conjunto A contém os meses do primeiro trimestre do ano, enquanto o conjunto B contém os três primeiros dias do mês.

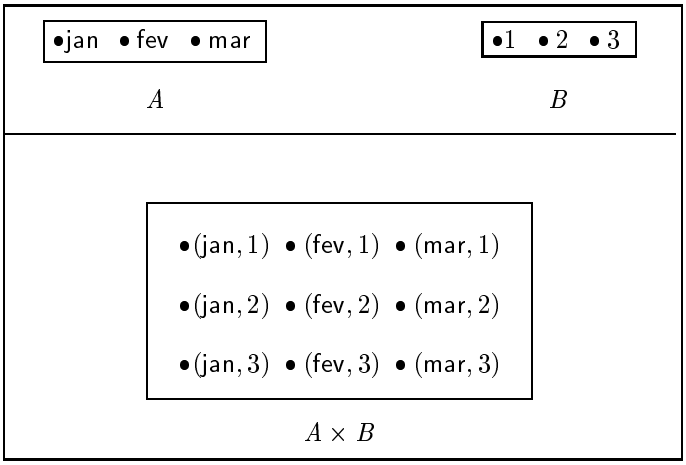


Figura 3.1 Produto Cartesiano

O produto cartesiano destes dois conjuntos representa todas as possíveis datas que podem ser formadas com os três primeiros dias dos meses do primeiro trimestre (denotadas pelo par (mês,dia)). □

Matematicamente, a definição do produto cartesiano é enunciada por:

Definição 3.1 *O produto cartesiano do conjunto S pelo conjunto T é definido por:*

$$C = S \times T = \{(x, y) \mid x \in S, y \in T\}$$

O número de elementos (cardinalidade) do conjunto C da Definição 3.1, denotado por $\#C$, corresponde à multiplicação da cardinalidade de S pela de T:

$$\#C = \#S * \#T$$

Note que se qualquer um dos conjuntos for infinito, o produto cartesiano resultante é também um conjunto infinito.

Como os conjuntos A e B do Exemplo3.11 são finitos, cada um deles tem 3 elementos, podemos verificar que a cardinalidade de $A \times B$ é calculada por:

$$\#(A \times B) = \#A * \#B = 3 * 3 = 9$$

Como precisamos agrupar valores dessa forma para a solução de vários problemas, a maioria das linguagens de programação atuais provêem elementos predefinidos para a representação de produtos cartesianos.

Suponha que queiramos estender o Exemplo 3.11 para representar todas as datas do ano representadas pelo par (mês,dia). Assim, podemos obter tal representação pelo produto cartesiano de um conjunto contendo todos os meses do ano (de jan a dez), pelo conjunto contendo todos os possíveis dias de um mês (de 1 a 31). O conjunto resultado do produto cartesiano destes dois conjuntos é como segue:

$$\left[\begin{array}{c} (jan,1),(jan,2),\cdots,(jan,31), \\ (fev,1),(fev,2),\cdots,(fev,31), \\ \vdots \\ (dez,1),(dez,2),\cdots,(dez,31) \end{array}\right]$$

Na linguagem Pascal, os `records` que denotam o produto cartesiano de conjuntos, e o conjunto de valores acima pode ser definido por um novo tipo como segue.

Exemplo 3.12 – As datas do ano, denotadas pelo par (mês,dia), podem ser representadas em Pascal pelo produto cartesiano (`record`) de mês (`MesesP`, Exemplo 3.8), pelo dia (`DiasP`, Exemplo 3.10):

```
type DataP = record
               m: MesesP
               d: DiasP
             end;
```

□

As estruturas criadas em C (`struct`) formam também produtos cartesianos de seus subelementos.

Exemplo 3.13 – As datas representadas por dia e mês podem ser definidas em C pelo tipo:

```
enum DiasC {1,2,3,4,5,6,7,8,9,10,
            11,12,13,14,15,16,17,18,19,20,
```

```
            21,22,23,24,25,26,27,28.29,30,31};
struct DataC {
              MesesC m
              DiasC d
             };
```

onde `MesesC` são os meses do ano definido no Exemplo 3.9. Os valores representados por este novo tipo são os mesmos do exemplo correspondente em Pascal (Exemplo 3.12) e do conjunto definido acima. □

3.3.2 União Disjunta

Uma das operações sobre conjuntos mais usada é a união de conjuntos. A união de dois conjuntos tem como resultado um terceiro conjunto que contém todos os elementos de ambos os conjuntos. Um caso especial da união de conjuntos é a chamada **união disjunta**. Nesta, o conjunto resultante possui todos os elementos dos dois conjuntos originais, e além disso podemos distingüir o conjunto origem de cada elemento; se do primeiro ou do segundo conjunto.

Exemplo 3.14 – Usando os mesmos conjuntos A (meses do primeiro trimestre) e B (três primeiros dias do mês) do Exemplo 3.11, podemos agora definir um novo conjunto resultante da união disjunta dos conjuntos A e B. Usamos o termo *prim* para discriminar que o elemento é proveniente do primeiro conjunto (A, neste exemplo), e o termo *seg* para discriminar que o elemento é proveniente do segundo conjunto (B, neste exemplo). A união disjunta destes conjuntos tem como resultado o conjunto $A + B$.

Sob o ponto de vista prático, o conjunto resultado ($A + B$) denota um tipo que tanto pode assumir um valor que representa um mês do primeiro trimestre (para os valores que são discriminados por *prim*), quanto assumir um valor que representa um dos três primeiros dias do mês (para os valores que são discriminados por *seg*). □

Matematicamente, a união disjunta é definida por:

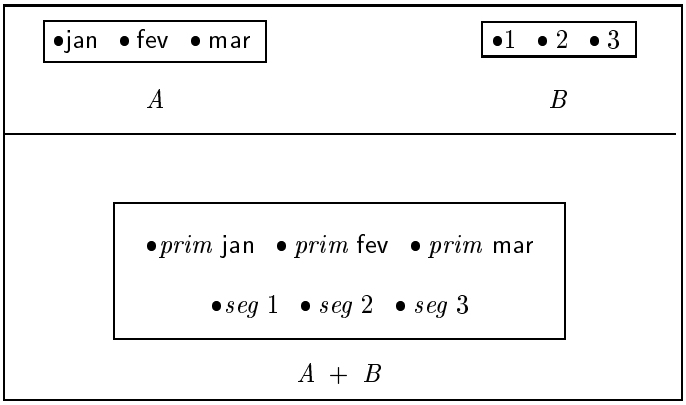


Figura 3.2 União Disjunta

Definição 3.2 *A união disjunta dos conjuntos S e T, denotada por $S + T$, é definida por:*

$$C = S + T = \{prim\ \ x \mid x \in S\} \cup \{seg\ \ y \mid y \in T\}$$

Note que o conjunto resultado da união disjunta possui todos os elementos dos conjuntos origem com as suas respectivas identificações da origem: *prim* identifica que o elemento é proveniente do primeiro conjunto, e *seg* do segundo conjunto. É importante salientar que as palavras escolhidas para discriminar o conjunto origem dos elementos é arbitrária; aqui usamos *prim* para denotar *primeiro* e *seg* para denotar *segundo*. No Exemplo 3.14, poderíamos usar termos mais significativos para os valores dos conjuntos: *mês* para os elementos do conjunto A e *dia* para os elementos do conjunto B, por exemplo.

Dado que o conjunto resultado possui todos os elementos dos dois conjuntos de forma discriminada, o número de elementos (cardinalidade) do conjunto C, denotado por $\#C$, corresponde à soma da cardinalidade de S com a de T.

$$\#C = \#S + \#T$$

Note que tanto os elementos quanto o número de elementos do conjunto resultado da união disjunta diferem dos correspondentes da união de conjuntos. Na união de

conjuntos, quando os dois conjuntos possuem elementos iguais apenas um elemento é considerado no conjunto resultado. Na união disjunta contudo, os dois elementos são considerados porque um discriminador é usado, e isso faz com que os elementos sejam diferentes. Se for realizada a união disjunta de dois conjuntos que possuem o número 1, por exemplo, o conjunto resultado possuirá os elementos $prim\ 1$ e $seg\ 1$, os quais são elementos distintos. Em conseqüencia disso, a cardinalidade do conjunto resultado será exatamente a soma das cardinalidades dos conjuntos originais, diferindo assim da cardinalidade da união de dois conjuntos.

Para o Exemplo 3.14, podemos calcular:

$$\#(A\ +\ B) = \#A + \#B = 3 + 3 = 6$$

Na linguagem Pascal, as uniões disjuntas são providas por registros juntamente com uma variável que discrimina o tipo. Chamamos estas de **união discriminada**, para as quais uma variável é responsável por discriminar o tipo.

Exemplo 3.15 – É comum, em aplicações científicas, precisarmos de números que podem assumir valores inteiros (exatos), ou valores de ponto-flutuante (aproximado). Dessa forma, gostaríamos de ter um novo tipo que congregasse os valores inteiros, discriminados por $exato$, e valores de ponto-flutuante, discriminado por $aprox$:

$$\{..., exato\ \text{-}1, exato\ 0, exato\ 1, ...\}\ \cup$$

$$\{..., aprox\ \text{-}1.0...., aprox\ 0., ..., aprox\ 1.0, ...\}$$

O tipo `NumeroP` abaixo pode ser um valor inteiro ou de ponto-flutuante e representa o conjunto de valores definido acima.

```
type Precisao = {exato, aprox};
     NumeroP = record
                 case prec : Precisao of
                     exato : (ival : Integer);
                     aprox : (rval : Real)
               end;
```

A união disjunta discriminada é representada pelo `record` juntamente com o `case`, o qual possui a variável `prec` que terá um dos valores do tipo `Precisao` (`exato`

ou `aprox`). Dessa forma, o acesso às variáveis `ival` e `rval`, definidas no `record`, deve ser feito mediante a verificação do valor atual de `prec`. Esta variável discrimina se o valor a ser usado é um número inteiro (quando `prec` possui o valor `exato` só a variável `ival` deve ser usada) ou de ponto-flutuante (quando `prec` possui o valor `aprox`, só `rval` deve ser usada). □

Apesar de ser representado por um `record`, o tipo acima não constitui um produto cartesiano, mas uma união disjunta por causa do `case` junto com a variável `prec`. Para esse novo tipo definido, apenas um dos elementos do `record` pode ser usado por vez porque um espaço de memória é compartilhado por ambas as variáveis (ver Capítulo 4). Assim, quando `prec` possui o valor `exato`, o espaço de memória do `record` é utilizado para armazenar um valor inteiro (da variável `ival`). Em contrapartida, o mesmo espaço será utilizado para armazenar um valor do tipo ponto-flutuante quando `prec` possui o valor `aprox`.

A linguagem C também oferece uma construção própria para o tipo união disjunta, mas esta é uma **união livre**, sem elemento discriminador de tipo.

Exemplo 3.16 – O tipo `NumeroC`, com propósitios análogos ao `NumeroP` acima, pode ser definido em C como:

```
union NumeroC {
               int   ival;
               float rval;
              };
```

Diferentemente das `struct` que definem produtos cartesianos de seus subelementos, na `union` apenas um dos elementos é acessível por vez. Um espaço de memória é reservado para armazenar `ival` e `rval`, e assim apenas um deles pode ser usado, o último que teve um valor atribuído. □

Vale salientar que a união disjunta pode tanto ter um elemento que discrimine o subelemento a ser usado, como em Pascal, quanto não possuir o elemento discriminador. Mesmo sem elemento discriminador a `union` em C representa uma união disjunta por caracterizar um novo conjunto contendo os dois tipos, e apenas um deles pode ser utilizado por vez. Outro ponto a ser ressaltado é que se faça a correta distinção,

nas linguagens, entre a união disjunta e o produto cartesiano como mostrado nos exemplos acima.

3.3.3 Mapeamentos

O mapeamento de dois conjuntos resulta em um terceiro conjunto de pares de elementos, onde o primeiro elemento é originário do primeiro conjunto e o segundo é originário do segundo conjunto. Além disso, cada elemento do primeiro conjunto está associado, no máximo, a um elemento do segundo conjunto.

Exemplo 3.17 – Usando os mesmos conjuntos A e B dos exemplos anteriores, podemos criar diferentes mapeamentos. Poderíamos ter, por exemplo, os conjuntos C e D abaixo:

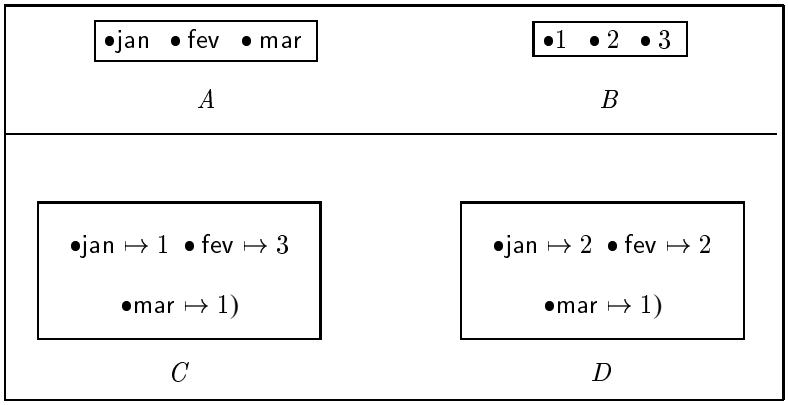


Figura 3.3 Mapeamento

intuitivamente, os conjuntos C e D representam possíveis valores quando queremos representar um dos três primeiros dias do mês para cada um dos meses do primeiro trimestre. Note que vários outros mapeamentos podem ser formados a partir dos conjuntos A e B. □

Matematicamente, os mapeamentos são definidos por:

Definição 3.3 *O mapeamento do conjuntos S para o conjunto T, denotado por $S \rightarrow T$, é definido por:*

$$m\colon S \rightarrow T = \{m \mid x \in S \ \Rightarrow \ m(x) \in T\}$$

Nos mapeamentos (ou funções), o primeiro conjunto (S) é denominado de **domínio** (ou **índices** no jargão das linguagens de programação), enquanto o segundo conjunto (T) é dito **contra-domínio** ou **imagem**. Na definição acima, m representa todos os possíveis mapeamentos que podem ser formados com o conjunto S como domínio e T como contra-domínio. Ainda considerando os conjuntos S e T quaiquer, como na Definição 3.3, o número de elementos (cardinalidade) do conjunto $S \rightarrow T$ é definido por:

$$\#(S \rightarrow T) = (\#T)^{\#S}$$

Para o Exemplo 3.17, podemos formar ao todo a seguinte quantidade de mapeamentos:

$$\#(A \rightarrow B) = (\#A)^{\#B} = 3^3 = 27$$

Mapeamentos de valores aparecem na maioria das linguagens de programação como *Arrays*. Como estas são construções predefinidas nas linguagens de programação, podemos tanto construir novos tipos utilizando estas estruturas, quanto definir variáveis.

Algumas linguagens permitem o mapeamento de elementos contanto que o domínio seja de valores inteiros ou ainda números naturais. Outras permitem que os conjuntos de valores índice sejam definidos pelo programador. Aqui seguem alguns exemplos de mapeamentos permitidos nas linguagens Pascal e C. Detalhes sobre os possíveis conjuntos domínio (índices) dos mapeamentos serão vistos no Capítulo 4.

Exemplo 3.18 – Em Pascal, um novo tipo pode ser criado como um mapeamento de inteiros no intervalo $[0 \,..\, 15]$ para números inteiros representáveis na linguagem:

```
var mapintP:  array [0..15] of Integer;
```

□

Exemplo 3.19 – De forma análoga, um tipo mapeamento de inteiros no intervalo $[0 .. 15]$ para inteiros pode ser criado na linguagem C:

```
int mapintC[16];
```

□

As variáveis `mapintP` e `mapintC` dos exemplos acima podem assumir quaisquer dos mapeamentos de elementos: $\{0, ..., 15\} \rightarrow \{..., -1, 0, 1, ...\}$. Em Pascal, o conjunto de índices é explicitamente definido, enquanto que em C apenas o número de elementos do mapeamento é definido, pois o índice é assumido do tipo inteiro de 0 a n (de forma a completar o número de elementos necessários).

O fato dos índices serem explicitamente declarados em Pascal, faz com que possamos ter índices que não sejam necessariamente números inteiros, como mostra o exemplo abaixo.

Exemplo 3.20 – Suponha que queiramos registrar a menor temperatura de cada mês do ano. Gostaríamos, por exemplo, de consultar a menor temperatura anotada para cada mês. Usando o tipo `MesesP` previamente definido (Exemplo 3.8), podemos criar:

```
...
 var menortemp: array[MesesP] of Real;
...
   if (menortemp[jan] < menortemp[fev]) ...
```

o índice utilizado é a representação dada por nós para os meses do ano. □

As funções existentes na grande maioria das linguagens de programação são também mapeamentos entre o domínio e o contra-domínio. Neste caso, os mapeamentos não são explicitamente construídos, como é o caso dos *arrays*, mas um algoritmo descreve a relação entre os elementos do domínio (os argumentos) e a imagem (o resultado correspondente computado pela função). Vale ressaltar que as funções definidas nas linguagens de programação não são exatamente uma função

matemática, mas uma representação para estas. Tais representações envolvem um algoritmo com suas características particulares, inclusive eficiência e abrangência, em vez de puros mapeamentos de valores, como no significado matemático.

3.3.4 Conjuntos Potência

Alguns problemas computacionais precisam de uma representação de conjuntos em vez de um único elemento do conjunto. Poderíamos, por exemplo, definir um conjunto de cores primárias para depois obtermos as várias cores a partir da mistura destas cores primárias. Para isso, não queremos que o nosso elemento seja exclusivamente uma das cores primárias, mas um conjunto das mesmas. Matematicamente, o conjunto formado por todos os possíveis subconjuntos de um dado conjunto é chamado de conjunto potência .

Exemplo 3.21 – Considerando ainda o conjunto A dos exemplos anteriores, podemos construir o conjunto potência , denotado por $\mathcal{P}(A)$:

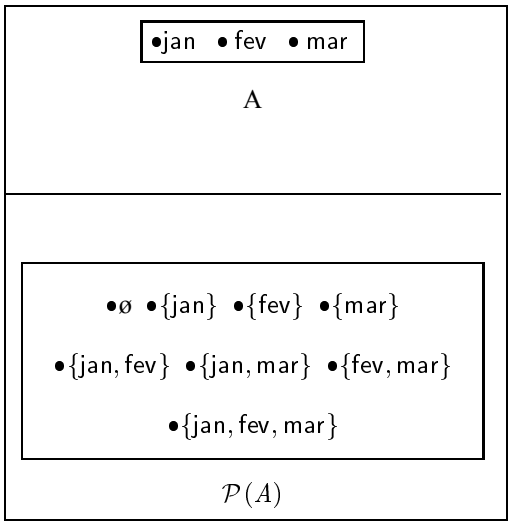


Figura 3.4 Conjuntos Potência

Intuitivamente, o conjunto $\mathcal{P}(A)$ representa todos os possíveis conjuntos que podemos formar com os meses do primeiro trimestre do ano. Este poderia ser usado, por

exemplo, em um problema para o qual gostaríamos de responder "em quais meses do primeiro trimestre de 2002 que tivemos temperaturas acima de 33^oC?". A resposta para esta pergunta poderia ser quaisquer dos valores representados em $\mathcal{P}(A)$. □

Matematicamente, o conjunto potência é definido por:

Definição 3.4 *O conjunto potência de um dado conjunto S, denotado por $\mathcal{P}(S)$, é definido como segue:*

$$\mathcal{P}(S) = \{s \mid s \subseteq S\}$$

Ou seja, todos os subconjuntos que podem ser formados com os elementos de S. O número de elementos (cardinalidade) do conjunto $\mathcal{P}(S)$ é dado por:

$$\#(\mathcal{P}(S)) = 2^{\#S}$$

Para o exemplo acima, como A possui 3 elementos, podemos formar 8 possíveis subconjuntos (como mostra a Figura 3.4), o resultado de

$$\#(\mathcal{P}(A)) = 2^{\#A} = 2^3 = 8$$

.

Apenas algumas linguagens de programação dão suporte à representação de conjuntos potência.

Exemplo 3.22 – Suponha que queiramos construir conjuntos de cores que podemos formar a partir das cores primárias, como enunciado anteriormente. Deveríamos portanto definir as cores primárias inicialmente e então construir novas cores a partir destas primárias. A linguagem Pascal possui um construtor de conjuntos que nos permite, por exemplo, construir conjuntos de cores primárias, as quais são definidas por uma enumeração (`Cores`):

```
type Cores = (vermelho, azul, amarelo);
     NovasCores = set of Cores
```

O conjunto de elementos do tipo `NovasCores` é constituído de:

```
{{}, {vermelho}, {azul}, {amarelo},
```

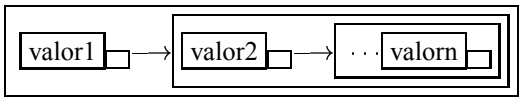
```
{vermelho,azul}, {vermelho,amarelo}, {azul,amarelo},
{vermelho,azul,amarelo}}
```

e qualquer variável deste tipo ou representa uma cor primária (quando o conjunto possui um único elemento) ou uma mistura de cores primárias. □

A linguagem C não possui construtores de conjuntos. Das linguagens imperativas atuais, Pascal e Modula-3 estão dentre as poucas que possuem manipulação de conjuntos (construtores juntamente com os respectivos operadores).

3.4 TIPOS RECURSIVOS

Em vários dos problemas que desejamos resolver computacionalmente nos deparamos com listas de elementos que não sabemos exatamente quantos elementos deveriam ter. Uma das alternativas que teríamos para representar dados com esta característica seria por mapeamentos (visto na Seção 3.3.3), pois estes preservam uma ordem dos elementos guardados. Contudo, para esta representação devemos determinar o número de elementos que desejamos guardar *a priori*. Outra forma de representação de listas é pelas células que contêm dois elementos: um valor e um segundo elemento que é um apontador para uma lista de elementos do mesmo tipo do valor.



No caso acima, a primeira célula da lista contém o valor1 e o segundo elemento é uma lista, a qual contém como primeira célula um elemento com o valor2, e assim por diante. Como cada um dos elementos contém uma lista como segundo elemento, dizemos que este tipo é recursivo, pois contém um elemento do seu próprio tipo. De uma forma geral, os tipos recursivos são compostos de valores que possuem o próprio tipo como valor; eles são definidos em termos deles próprios:

$$T = ...T... \tag{3.1}$$

De fato, uma lista deve ser vista como um tipo recursivo e não como um mapeamento. Um tipo lista de inteiros, por exemplo, pode ser definido pela sua unidade base, a lista vazia ($Unit$), juntamente com um inteiro seguido de uma lista de inteiros:

$$IntList = Unit + (Integer \times IntList) \tag{3.2}$$

Note que nesta definição $IntList$ é utilizado para definir a si próprio, ou seja é um tipo recursivo. Considerando a lista vazia como nil e $cons$ um construtor de listas (constrói uma lista dado um elemento i e uma lista l), podemos ainda re- definir a equação acima como segue:

$$IntList = \{nil\} \cup \{cons(i, l) \mid i \in Integer, l \in IntList\} \tag{3.3}$$

Em outras palavras, o tipo lista de inteiros é um conjunto formado pela união de uma unidade base de lista (nil), com todas a listas de inteiros com apenas 1 elemento, todas a listas de inteiros com 2 elementos, etc. A Equação 3.3 pode ainda ser desmembrada em:

$$\{nil\} \tag{3.4}$$

$$\cup\{cons(i, nil) \mid i \in Integer\} \tag{3.5}$$

$$\cup\{cons(i, cons(j, nil)) \mid i, j \in Integer\} \tag{3.6}$$

$$\ldots \tag{3.7}$$

Desta forma, obtemos um conjunto infinito de listas, calculado pela união do conjunto que contém a lista vazia (nil) com todas as listas de inteiros com um único elemento, mais todas as listas de inteiros com dois elementos e assim por diante. Um raciocínio similar pode ser usado para listas de quaisquer outros tipos.

O fato do tipo recursivo ser definido em termos dele próprio faz com que novos elementos sempre possam ser criados. Assim, a cardinalidade de tipos recursivos é infinita, mesmo que o conjunto de valores do tipo seja finito. Considerando o conjunto

A dos exemplos anteriores, quais listas podemos formar com os três elementos do conjunto?

Exemplo 3.23 – As listas que podem ser formadas com elementos do conjunto A são:

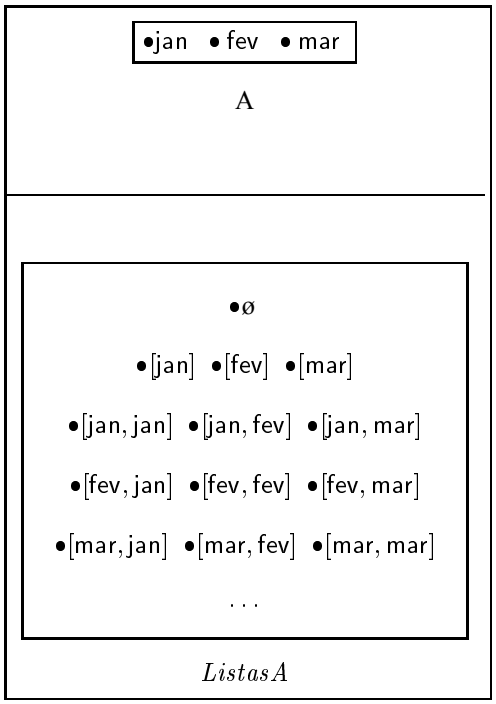


Figura 3.5 Tipo Recursivo

além das listas com um e dois elementos aqui mostradas, precisamos de todas as listas possíveis que podemos formar com três, quatro, etc, elementos. Isso resulta em um conjunto infinito de listas. □

As listas são tipos recursivos encontrados em algumas linguagens de programação funcionais. Na maioria das linguagens, estas são construídas ao invés de predefinidas; é comum criarmos estruturas de dados ou objetos que constroem o tipo lista.

Exemplo 3.24 – Na linguagem Pascal, por exemplo, uma lista de números inteiros pode ser definida por um `record` com o auxílio de apontadores:

```
type IntListP = ^ IntNode;
     IntNode  = record valor: Integer;
                       prox: IntListP
                end;
```

aqui, `valor` guarda o valor inteiro e `prox` é um apontador para uma lista de inteiros, caracterizando o tipo recursivo. Contudo, como um tipo em Pascal não pode ser definido usando o seu próprio nome, é necessário que se crie um elemento auxiliar para se fazer a referência cruzada. □

Exemplo 3.25 – Na linguagem C, uma lista de números inteiros pode ser também definida por apontadores como segue:

```
typedef struct NoIntlist* IntListC ;
struct  NoIntlist {
                   int valor;
                   IntListC *prox;
                  }
```

da mesma forma que no exemplo anterior, `prox` é um apontador para uma estrutura do tipo `IntListC`, um tipo recursivo. □

Diferente das linguagens imperativas, as linguagens funcionais, tais como SML [48, 46], Miranda [62] e Lisp [36], possuem listas como tipos predefinidos (`list`). Como este é um tipo predefinido na linguagem, as operações básicas, tais como: primeiro elemento da lista (`hd`), a lista sem o primeiro elemento (`l`) e tamanho da lista (`lenght`), estão também predefinidas na linguagem. Ter tais operações predefinidas na linguagem se faz necessário porque estas são as regras uniformes com as quais gostaríamos de tratar as listas, da mesma forma que temos as operações aritméticas para tratar o tipo números inteiros, por exemplo.

Exemplo 3.26 – Na linguagem SML, uma lista de inteiros e sua manipulação podem ser definidas como segue:

```
5::3::2::nil =  5::3::[2] =  5::[3,2] = [5,3,2]
```

`nil` representa a lista vazia (também representada por `[]`), e `::` é um construtor de lista que precisa de um elemento e uma lista. Algumas operações fundamentais sobre o tipo lista são também predefinidas na maioria das implementações da linguagem:

```
  [5,3,2] @ [7,8]  {concatenação de listas}
> [5,3,2,7,8] : int list
  rev [5,3,2];     {lista em ordem inversa}
> [2,3,5] : int list
  hd [5,3,2];      {primeiro elemento da lista}
> 5: int
  tl [5,3,2];      {descarta o primeiro elemento}
> [3,2] : int list
  length [5,3,2];  {o número de elementos da lista}
> 3 : int;
```

as duas primeiras operações (`@` e `rev`) fazem parte da linguagem, enquanto as outras podem ser implementadas a partir dos elementos base da linguagem, mas vêm implementadas na grande maioria dos processadores da linguagem. □

Além de se ter listas como tipo predefinido, as linguagens funcionais também fornecem mecanismos para a construção de novos tipos. Neste caso, novos construtores podem ser criados para indicar a formação de novos tipos como indica o exemplo seguinte.

Exemplo 3.27 – As listas de inteiros podem ser definidas em SML pela criação de um tipos específico:

```
datatype IntlistS =   nil                        (1)
                    | cons of int * IntlistS     (2)
```

`nil` representa a lista vazia e `cons` é o construtor de lista. Nesta definição do tipo lista de inteiros, temos duas definições alternativas: em (1), uma lista de inteiros pode ser uma lista vazia; e em (2), é construído (`cons`) uma nova lista contendo um valor inteiro (`int`) combinado (`*`) com uma lista de inteiros (`IntlistS`). Note que a

definição deste tipo em SML corresponde ao tipo recursivo definido na Equação 3.2.
□

3.5 TIPOS CONSTRUÍDOS PELO PROGRAMADOR

Nas seções anteriores, vimos que a maioria das linguagens fornece mecanismos para a criação de tipos dados compostos, os quais são inspirados em elementos matemáticos que agregam valores. Com os *arrays* nas linguagens Pascal e C, por exemplo, podemos ter valores agregados como mapeamentos finitos. Assim, variáveis podem ser criadas para agregar valores em forma de mapeamentos. Da mesma forma os `records` da linguagem Pascal e `struct` da C, por exemplo, fornecem recursos para que tenhamos dados agregados como produtos cartesianos. Além disso, em vários dos exemplos mostrados foram dados nomes aos novos tipos de dados.

Exemplo 3.28 – As datas do ano, denotadas pelo par (mês,dia), criadas em Pascal e C, por exemplo, foram nomeadas como `DataP` e `DataC` respectivamente:

```
type DataP = record
               m: MesesP
               d: DiasP
             end;


struct DataC {
               MesesC m
               DiasC d
             };
```

□

A partir da definição desses novos tipos, variáveis podem ser criadas tanto usando os tipos predefinidos quanto os novos tipos. De uma forma geral, o programador pode definir tipos de dados nestas linguagens pela renomeação de tipos existentes ou agregação de tipos predefinidos ou criados na linguagem (via os construtores

disponíveis em cada linguagem). A declaração de tipos, como mostrado acima, permite que o programador tenha um mecanismo uniforme de acesso aos componentes agregados, e assim várias instâncias destes elementos podem ser criadas (variáveis declaradas como deste tipo).

Existem algumas vantagens de se ter novos tipos nomeados [4]: a facilidade de leitura dos programas, uma vez que os tipos são nomeados de forma significativa para a solução do problema; facilidade de modificação de tipos de variáveis, dado que basta modificar o tipo para modificar todas as variáveis correspondentes; facilidade de escrita dos tipos de variáveis, uma vez que o tipo é declarado de uma única vez, não havendo necessidade de repetir a mesma definição para cada variável e diminui o risco de se definir variáveis que deveriam ter o mesmo tipo de maneira diferente. Uma discussão sobre os tipos construídos e a verificação de tipos será vista no Capítulo 5.

Vale lembrar que estas vantagens aqui citadas são características desejáveis nas linguagens de programação no tocante à facilidade de definição de tipos e estruturas, facilidade de escrita e clareza (descritas no Capítulo 2).

3.6 LEITURA RECOMENDADA

A teoria dos tipos tem sido vastamente estudada. Um estudo sobre teoria de tipos para linguagens de programação pode ser encontrado em [61] e [52]. Um estudo sistemático sobre especificação de tipos abstratos pode ser encontrado em [35], e a partir deste, o leitor pode encontrar referências para várias abordagens da teoria dos tipos.

Um estudo sistemático sobre tipos compostos pode ser encontrado em [32], [33] e [60].

3.7 EXERCÍCIOS

1. Para cada um dos tipos abaixo (definidos em Pascal), mostre qual conjunto de valores eles representam usando a notação de tipos primitivos, produto cartesiano (X), união disjunta ($+$), mapeamentos ($\rightarrow$), conjuntos potência ($\mathcal{P}$).

```
type  Cores = (azul, vermelho, amarelo);
      Tonalidade = 1..10;
      Textura = 1..5;
      PontoColorido = record c: Cores;
                             to: Tonalidade;
                             tx: Textura
                      end;
      LinhaColorida = array [1..30] of PontoColorido;
      NovaCor = set of Cores;
      Noptr = ^NoListras;
      NoListras = record c: LinhaColorida;
                         next: Noptr
                  end;
```

2. Analise de forma sistemática quais os tipos de dados presentes nas linguagens que você conhece. Para cada uma das linguagens analisada defina:

   (a) Qual o conjunto de valores de cada tipo primitivo da linguagem?

   (b) Quais os tipos compostos existentes na linguagem? Eles cobrem todos os tipos compostos apresentados neste capítulo?

   (c) É permitido construir novos tipos na linguagem? Existem limitações para isso?

3. Analise os seguintes conjuntos de dados usando os tipos matemáticos apresentados neste capítulo (os tipos compostos):

   (a) Um banco de dados (BD) relacional, onde cada elemento do BD é dado por uma relação que pode ser uma tupla com dois elementos (um do tipo T e o outro do tipo S). Explore as alternativas para isso.

   (b) Um arquivo com acesso direto, onde dado um valor do tipo T temos acesso a um outro valor do tipo T'.

(c) Um arquivo com acesso seqüencial, onde cada elemento do aquivo é um valor do tipo T.

(d) Acesso ao resultado de uma pesquisa na internet por palavras-chaves.

4. Vimos na Seção 3.3.3 que funções e *arrays* podem representar a estrutura matemática de mapeamentos finitos. Essas duas representações podem ser usadas indistintamente para representar os mesmos mapeamentos?

   (a) Compare o conjunto de valores que podem ser representados por ambas as representações.

   (b) Compare quanto ao acesso aos elementos do mapeamento.

   (c) Compare quanto à atribuição de valores aos elementos do mapeamento.

5. Em várias das linguagens de programação atuais encontramos a representação de *strings*. Pesquise nas linguagens de programação que você conhece qual o tipo associado às *strings* em cada uma das linguagens (tipo primitivo, composto? qual?).

6. As listas são raramente tipos predefinidos nas linguagens de programação. Quando é o caso, devemos ter algumas operações juntamente predefinidas para caracterizá-las como tipo.

   (a) Quais operações são necessárias para o tipo lista?

   (b) Quais as vantagens de termos lista como um tipo predefinido em uma linguagem de programação?