Estrutura de Dados

João Dovicchi

Conteúdo

Prefácio

Este livro aborda alguns dos principais conceitos da área de programação e estrutura de dados. Podemos considerar os dados como conjuntos de elementos de determinados tipos que podem ser tratados como unidades informacionais. A informação, entretanto, depende da contextualização dos dados, isto é, como são selecionados, relacionados e organizados. Uma vez que a ciência da computação está fundamentada no processamento da informação, o estudo do conteúdo desta informação depende dos tipos de dados que são manipulados e sua organização. Assim, a disciplina *Estrutura de Dados* trata da manipulação, organização e utilização destes dados.

$$\textbf{Estrutura de Dados} \begin{cases} \text{Como manipular} \\ \text{Como organizar} \\ \text{Como utilizar} \end{cases}$$

Com a finalidade de compreender os dados e a maneira como eles são tipados, armazenados, ordenados etc., vamos utilizar, neste livro, alguns conceitos de programação procedural e funcional[1]. Para isso, escolhemos a linguagem C ANSI para apresentar os algoritmos conceituais no modelo procedural e a linguagem HASKELL no modelo funcional. A forma procedural (em C) apresenta uma visão imperativa dos dados e algoritmos, enquanto a forma funcional apresenta uma visão declarativa. As linguagens C e HASKELL foram escolhidas pela simplicidade, pela facilidade de compreensão e pelo poder de processamento de dados.

Como as linguagens de programação, em geral, são apenas formas semânticas de representação, elas facilitam a implementação dos algoritmos. Assim, quando necessário faremos algumas incursões na sintaxe de C e HASKELL , apenas o suficiente para compreender o processo de um algoritmo.

[1] Ver apêndice A

O conteúdo deste curso se encontra dividido em 4 partes. Na primeira parte serão estudados os conceitos fundamentais e introdutórios das entidades e abstrações que compõem a representação dos dados, além das expressões que formam os termos e sintaxe desta representação (capítulo 1). Na segunda parte, abordaremos as representações propriamente ditas, as operações com dados e a construção de funções que nos permitam fazer estas operações (capítulo 2). Na terceira parte nos dedicaremos ao estudo da manipulação de dados em listas, filas, pilhas e outras possibilidades de representação, tais como grafos e árvores (capítulos 3 e 4). Finalmente, na quarta parte vamos ver como as estruturas de dados se relacionam com a atividade de programação e implementação de algoritmos, ressaltando os conceitos de máquinas abstratas, recursividade, classificação, busca, ordenação e hashing (capítulos 5 e 6).

No apêndice A, encontra-se alguns conceitos básicos de programação funcional, particularmente da linguagem HASKELL . O apêndice B traz uma introdução ao cálculo-λ que é a fundamentação teórica das estruturas em Linguagem Funcional . Finalmente, o apêndice C traz alguns exercícios relacionados aos conceitos dos capítulos do livro.

Prof. Dr. João Dovicchi
Florianópolis, agosto de 2007.

Capítulo 1

Dados

Se partirmos do pressuposto de que a informação tem por objetivo a redução da incerteza sobre determinado fato, podemos afirmar que informação é uma mensagem resultante do processamento, manipulação e organização dos dados, com a finalidade de proporcionar o conhecimento de um fato a quem tiver acesso a ela. Assim, podemos definir **dados** como as unidades básicas de um conjunto de informações.

Em computação, dados são conjuntos de valores ou caracteres representados por dígitos binários (bits) e que se localizam em endereços de memória, arquivos ou qualquer outro meio digital. Eles podem ser representados por conjuntos de bits de tamanho variável, dependendo do seu tipo e tamanho, e podem sofrer vários tipos de operações. Neste capítulo, vamos entender como os dados são definidos (tipados) e de que maneira podem ser manipulados.

1.1 Tipos de Dados

Os dados podem ser tipados dependendo de vários fatores que possibilitam a sua caracterização. Eles devem ser definidos com relação a três propriedades principais:

1. **Conjunto de valores**, ou seja, que conjunto de valores uma variável ou campo de dados pode receber ou armazenar;

2. **Conjunto de operadores**, ou seja, que conjunto de operadores podem agir sobre os dados ou campo de dados; e

3. **Referência**, ou seja, de que maneira os dados podem ser localizados ou referenciados.

Nos sistemas digitais de informação (informática), os dados podem ser de 3 tipos: numéricos, literais e lógicos. Os dados numéricos representam entidades aritméticas, podendo ser números inteiros, decimais ou racionais. Os dados literais representam caracteres, dígitos e símbolos de uma determinada tabela ou conjunto ou uma cadeia deles. Os dados lógicos representam os valores lógicos de verdadeiro ou falso.

Algumas características são implicitamente declaradas como, por exemplo, `int`, `char`, `float` etc.. Outras podem ser declaradas explicitamente, de forma abstrata. Este último tipo é denominado Tipo de Dado Abstrato (TDA). Um TDA consiste em duas partes: a definição de valores e a definição de operadores.

Os valores dos dados podem ser armazenados em variáveis ou constantes para ser referenciados posteriormente. No caso das variáveis, seus conteúdos podem ser alterados no decorrer do algoritmo, enquanto as constantes não. As variáveis, ao ser declaradas, fazem com que o compilador reserve o espaço adequado de memória para o seu tipo.

1.2 Tipo de dado abstrato

A programação orientada a objetos — *Object Oriented Programming* (OOP) — pode ser descrita como a programação de tipos abstratos de dados e suas relações. Independentemente da linguagem utilizada, um tipo abstrato de dado pode ser especificado por uma estrutura abstrata (*abstract structure*), definição de valores e de operadores. A definição tem 2 propriedades:

1. Exporta um **tipo**, definindo suas condições e seu domínio; e
2. Exporta as **operações**, que define o acesso ao tipo de estrutura de dados.

Para compreender como se define um tipo de dado, vamos tomar como exemplo a definição de um tipo de dado racional em C++ (`RATIONAL`). Um número racional é aquele que pode ser expresso como um quociente de dois inteiros. Podemos, por exemplo, definir as operações adição, multiplicação e igualdade sobre os números racionais a partir de dois inteiros, como na especificação deste tipo de dado abstrato (TDA) na listagem 1.1.

Observe que a definição do tipo `RATIONAL`, em C, não deixa claro como as operações funcionam, mas observe a mesma definição em uma linguagem funcional (veja listagem 1.2)

Listagem 1.1: Tipo Abstrato de Dado: Rational.

```
/* definicao de valor */
abstract typedef  RATIONAL;
condition RATIONAL[1] <> 0;

/* definicoes de operadores */
abstract RATIONAL makerational (a,b)
int a,b;
precondition b<>0;
postcondition makerational[0] == a;
              makerational[1] == b;

abstract RATIONAL add(a,b)  /* written a + b */
RATIONAL a,b;
postcondition add[1] == a[1] * b[1];
              add[0] == a[0] * b[1] + b[0] * a[1];

abstract RATIONAL mult(a,b) /*written a * b */
RATIONAL a,b;
postcondition mult[0] == a[0] * b[0];
              mult[1] == a[1] * b[1];

  abstract RATIONAL equal(a,b)  /*written a == b */
  RATIONAL a,b;
  postcondition equal == (a[0] * b[1] == b[0] * a[1]);
```

Listagem 1.2: Tipo Abstrato de Dado Racional em Haskell.

```
data (Integral a) => Ratio a = !a :% !a deriving (Eq)
type Rational = Ratio Integer

(%) ::  (Integral a) => a -> a -> Ratio a

reduce   0 = error "Ratio.% : zero denominator"
reduce x y = (x `quot` d) :% (y `quot` d)
       where d = gcd x y

instance (Integral a) => Num (Ratio a)
       where
         (x:%y) + (x':%y') = reduce (x*y' + x'*y) (y*y')
         (x:%y) * (x':%y') = reduce (x * x') (y * y')
```

Observe, na listagem 1.2, como é mais clara a definição de `Rational`. Assim, pode-se perceber que, dado $a + b$, onde

$$a = \frac{a_0}{a_1} \qquad \text{e} \qquad b = \frac{b_0}{b_1}$$

então

$$a + b = \frac{a_0 \times b_1 + a_0 \times a_1}{a_1 \times b_1}$$

As linguagens orientadas ao objeto utilizam o TDA para disponibilizar novos tipos de dados definidos pelo usuário (p. ex. C++, Java etc.), mas os TDAs podem ser definidos para outros tipos de estrutura.

1.3 Tipos de dados em C

Em C, existem os tipos básicos `int, float, double e char`. Como já vimos uma variável declarada como `int` contém um inteiro; declarada como `float` contém um número real de ponto flutuante; e, se declarada como `double` conterá um número real de ponto flutuante e de dupla precisão.

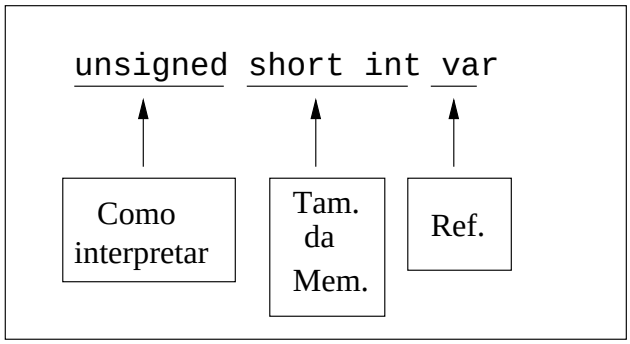


Figura 1.1: Representação de dados em C

A variável do tipo `int`, por exemplo, pode ser declarada junto com três tipos de qualificadores: `short, long` e `unsigned`. Os qualificadores `short` e `long` podem variar de uma máquina para outra mas, em geral têm as seguintes especificações:

`short int` – normalmente valores entre -32768 e 32767

`long int` – normalmente valores entre -2147483648 e 2147483647

`unsigned short int` – valores entre 0 e 65535

`unsigned long int` – valores entre 0 e 4294967295

A declaração de variáveis em C especifica dois atributos: a quantidade de memória a ser reservada para o armazenamento do tipo de dado; e como os dados armazenados devem ser interpretados (veja figura 1.1).

1.4 Operações

As operações que podem ser efetuadas sobre dados são de três tipos: Aritméticas, Lógicas e Relacionais. É evidente que tais operações se referem aos dados numéricos, enquanto que dados literais podem sofrer outros tipos de operações, tais como, concatenação, tamanho da cadeia, determinação de posição de um caractere na cadeia, comparação etc.. As operações podem ser:

- **Aritméticas**, por exemplo: Adição, Subtração, Multiplicação, Divisão etc.;

- **Lógicas**, por exemplo: NOT, AND, OR, XOR;

- **Relacionais**. por exemplo: Maior que (>), menor que (<), igual (==), diferente (!=) etc..

1.5 Referências

Para que se possa acessar ou recuperar um determinado dado da memória, é necessário que se tenha alguma referência a este dado. Normalmente, pode-se referenciar um dado pelo seu identificador ou campo. O nome de uma variável, por exemplo, pode ser o identificador de um determinado dado. Entretanto, conjunto de dados podem ser organizados em vetores ou matrizes, e podemos referenciá-los por meio de índices. Além disso, podemos referenciar os dados por seus endereços de memória acessando o conteúdo destes endereços. Esta última forma de referência é feita por meio de apontadores ou ponteiros.

Os tipos de referência se classificam em:

- **Direta:** pelo identificador ou campo;

- **Indexada:** pelo seu índice vetorial, matricial ou dimensional (coordenadas); e

- **Endereçada:** por apontadores ou ponteiros para endereços de memória.

Ponteiros permitem referenciar a posição de um objeto na memória, bem como o próprio objeto, ou seja, o conteúdo de sua posição. Assim, se uma variável `x` for declarada como um inteiro, `&x` se refere ao endereço de memória reservado para conter o valor de `x`. Veja, por exemplo o programa da listagem 1.3. Depois de compilado o programa deve retornar:

```
Valor de a: 5
Endereco de a: bfbff9cc
Conteudo de &a: 5
```

Listagem 1.3: Programa exemplo de referência.

```
/* Tipos de variaveis e enderecos */
#include <stdio.h>
#include <stdlib.h>

int main (int argv, char *argc[]){
  int a;
  a = 5;
  printf("Valor de a: %d\n", a);
  printf("Endereco de a: %x\n", &a);
  printf("Conteudo de &a: %d\n", *(&a));

  return 0;
}
```

1.6 Dados em C: Tipos e modificadores

A linguagem C trabalha com o conceito de *tipo de dado* com a finalidade de definir uma variável antes que ela seja usada. Em termos de estrutura, tal procedimento é utilizado para possibilitar a alocação de memória para a variável na área de dados do programa.

Quando se trata de definir variáveis, é importante ressaltar que tal definição pode se dar em dois escopos:

1. Definição da variável como declarativa com alocação de memória:

   ```
   int a;

   char c;

   struct per_rec person;
   ```

2. Definição como construtor de nome, parâmetro e tipo de retorno de uma função:

   ```
   long sqr(int num)

   {

        return(num*num);

   }
   ```

Em C existem, basicamente 6 tipos, no padrão ANSI da linguagem, que definem o tipo de variável a ser manipulada pelo programa naquele endereço de memória, a saber: `int, float, double, char, void` e `enum`[1].

1.6.1 O tipo `int`

O tipo `int` define uma variável do tipo "inteiro" que possue o tamanho de 16 bits (2 bytes) e pode armazenar valores de -32767 a 32767. Por exemplo:

```
{
     int Conta;
     Conta = 5;
}
```

1.6.2 O tipo `float`

O tipo `float` define uma variável do tipo "real" (com ponto flutuante) que tem o tamanho de 32 bits (4 bytes) e pode armazenar valores reais com seis dígitos de precisão. Por exemplo:

```
{
     float quilos;
     quilos = 8.123456;
}
```

1.6.3 O tipo `double`

O tipo `double` define uma variável real de maior precisão que tem o tamanho de 64 bits (8 bytes) e pode armazenar números reais com 10 dígitos de precisão. Por exemplo:

```
{
     double _Pi;
     _Pi = 3.1415926536;
}
```

Nota: Esta questão de precisão pode variar de compilador para compilador ou de processador para processador. Voce poderá testar a precisão de seu

[1] Compiladores mais recentes reconhecem o tipo `boolean`.

compilador/processador usando um programa que gere números e verificando a precisão. Por exemplo, o programa:

```
#include <stdio.h>
#include <math.h>

int main (){
     double rl;
     rl = 4*atan(1.0);
     fprintf(stdout, "%.10f\n", rl);
     return 0;
}
```

imprime o valor de π com 10 casas depois da vírgula. Voce pode alterar o parâmetro `%.10f` para valores maiores que 10 e verificar a precisão usando um valor bem preciso de π conhecido. O valor com 50 casas depois da vírgula é:

$$\pi = 3.14159265358979323846264338327950288419716939937508$$

1.6.4 O tipo `char`

O tipo `char` define uma variável do tipo "caractere" com o tamanho de 8 bits (1 byte), que armazenha o valor ASCII do caractere em questão. Por exemplo:

```
{
     char letra;
     letra = 'x';
}
```

Uma cadeia de caracteres é definida como um apontador para o endereço do primeiro caractere e que tem um tamanho definido de um *array*.

1.6.5 O tipo `enum`

O tipo `enum` esta relacionado à diretiva `#define` e permite a definição de uma lista de aliases que representa números inteiros, por exemplo, no lugar de codificar:

```
#define SEG 1
#define TER 2
```

```
#define QUA 3
#define QUI 4
#define SEX 5
#define SAB 6
#define DOM 7
#define FALSE 0
#define TRUE 1
```

é possível usar

```
enum semana { Seg=1, Ter, Qua, Qui, Sex Sab, Dom} dias;

enum boolean { FALSE = 0, TRUE };
```

1.7 Modificadores

Os tipos `int, float` e `double` podem ser redimensionados por modificadores `long` ou `short`; e `signed` ou `unsigned`. Tais modificadores definem a quantidade de bytes alocados na memória para as respectivas variáveis (veja tabela 1.1).

Tipo	Bytes	Bits	Valores		
short int	2	16	-32.768	→	+32.767
unsigned short int	2	16	0	→	+65.535
unsigned int	2	16	0	→	+65.535
int	2	16	-32.768	→	+32.767
long int	4	32	-2,147,483,648	→	+2,147,483,647
signed char	1	8	-128	→	+127
unsigned char	1	8	0	→	+255
float	4	32			
double	8	64			
long double	16	128			

Tabela 1.1: Tipos de dados e tamanho das variáveis

O uso do modificador `signed` com `int` é redundante, no entanto é permitido porque a declaração do tipo inteiro assume um número com sinal. O uso de

`signed char` pode ser feito em compiladores que não implementam este tipo por default, assim como o uso de `unsigned double` deve ser usado com cautela pois alguns compiladores não permitem esta modificação em números de ponto flutuante. Isto tudo deve ser levado em conta no caso da portabilidade do código.

Nos dias de hoje, o custo da memória é relativamente baixo, por isso poucos se preocupam com a quantidade utilizada, mas tudo isso é uma questão de conceito, uma vez que as linguagens de alto nível que estão aparecendo se incumbem de dimensionar a memória. A questão é: será que o programa resultante é otimizado?

1.8 Qualificadores de dados

A linguagem C define dois tipos de qualificadores `const` e `volatile` que podem melhorar o desempenho, principalmente em cálculos matemáticos. Os qualificadores podem ser usados para definir constantes que são usadas em qualquer parte do programa. Por exemplo:

```
const float pi=3.14159;
```

O qualificador `volatile` é usado para propósitos dinâmicos, muito usado em *device drivers*, para passar um dado por uma porta de *hardware*. Por exemplo:

```
#define TTYPORT 0x17755U

volatile char *port17 = (char)*TTYPORT;

*port17 = 'o';
*port17 = 'N';
```

na falta do qualificador `volatile`, o compilador pode interpretar a declaração `*port 17 = 'o';` como redundante e removê-la do código objeto. A declaração previne a otimização do código pelo compilador.

Capítulo 2

Representação de dados

Já vimos como podemos representar dados de forma direta, referenciando-os por meio de variáveis. Vamos, agora analisar a representação de dados que podem ser referenciados por índices. Para isso, devemos, partir de alguns conceitos fundamentais que envolvem vetores e matrizes.

Primeiramente, vamos introduzir o conceito de referência indexada que pode ser de vários tipos:

1. Vetorial;
2. Matricial; e
3. Multidimensional.

Consideremos, por exemplo, que tenhamos de manipular dados referentes a temperaturas médias de 3 cidades, no período de 4 meses. Estes dados podem ser organizados em uma tabela onde as colunas se referem às temperaturas médias em cada mês e as linhas se referem às temperaturas médias em cada cidade (veja tabela 2.1).

Cidade	**Janeiro**	**Fevereiro**	**Março**	**Abril**
Campinas	26	25	27	26
Campos do Jordão	22	23	21	20
Ribeirão Preto	31	29	30	29

Tabela 2.1: Dados de temperatura média de três cidades paulistas no período de janeiro a abril de 2003.

De forma semelhante à tabela, estes dados podem ser representados em uma matriz, onde cada coluna representa as médias mensais e onde cada linha representa os dados de uma cidade. Por exemplo:

$$\mathbf{A} = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \end{vmatrix} = \begin{vmatrix} 26 & 25 & 27 & 26 \\ 22 & 23 & 21 & 20 \\ 31 & 29 & 30 & 29 \end{vmatrix}$$

A representação de dados por arranjo (*array*) multidimensional é uma forma comum de referenciar dados de um conjunto coerente. A dimensão n de um array pode ser qualquer valor inteiro. Assim,

- Se $n = 1$, o array é unidimensional e é chamado de vetor;
- Se $n = 2$, o array é bidimensional e é chamado de matriz;
- Para $n = 3$, o array é tridimensional e pode ser representado por uma matriz de matrizes; e
- Para $n > 3$, o array é multidimensional e podemos representá-los apenas formalmente.

Vamos estudar como é feita a representação de dados em array como tipo de dado abstrato, sua utilização e as operações com arrays de dados numéricos e literais.

2.1 Representação em array

Dados de um determinado conjunto podem ser representados como um tipo de dado abstrato (TDA), que respeita algumas convenções. Por exemplo, uma seqüência pode ser descrita como na listagem 2.1

Assim, pode-se definir um TDA para um vetor como na listagem 2.2.

Listagem 2.1: TDA de uma seqüência

```
/* Exemplo Genérico:
abstract typedef <tipo1, tipo2, ..., tipoN>
    [tipo_da_seq]; */

abstract typedef <<int>> intseq;
    /* seq. de inteiros de qualquer tamanho */

abstract typedef <integer,char,float> seq3;
    /* seq de tamanho 3 com um inteiro,
    um caractere e um float */

abstract typedef <<int,10>> intseq;
    /* seq de 10 inteiros */
```

Listagem 2.2: TDA de um vetor

```
abstract typedef <<tipo, tam>> ARRTYPE(tam,tipo);
condition tipo(tam) == int;
abstract tipo extract(a,i); /* written a[i] */
ARRTYPE(tam, tipo) a;
int i;
precondition 0 <= i < tam;
postcondition extract == a; /* written a[i] = tipo_el */
ARRTYPE(tam, tipo) a;
int i;
tipo tipo_el;
precondition 0 <= i < tam;
postcondition a[i] == tipo_el;
```

Listagem 2.3: Programa exemplo de uso de array

```
#include <stdio.h>
#include <stdlib.h>

#define NUMEROS 10

int main (int argv, char *argc[]){
  int num[NUMEROS];   /* vetor de numeros */
  int i, total;
  float media;

  total = 0;
  for(i=0; i<NUMEROS; i++){
    scanf("%d", &num[i]);
    total += num[i];
  }
  media = (float) total / (float) NUMEROS;
  printf("Media = %f\n", media);
  return 0;
}
```

Desta forma, pode-se usar a notação `a[i]`, onde a representa uma variável e i um índice. Na prática temos que a referencia o endereço de memória da primeira variável do vetor e i referencia o deslocamento dentro desta área reservada, que depende do tipo de variável declarada.

Vamos, tomar um exemplo de uso de um vetor para manipulação de dados. Suponhamos que temos que entrar 10 números inteiros quaisquer e devemos efetuar algumas operações sobre eles, por exemplo, calcular sua média (veja listagem 2.3).

Neste exemplo a variável `num[i]` representa um vetor do tamanho de 10 elementos (definido na constante `NUMEROS`).

Listagem 2.4: media2.h

```
/* Cabecalho para o programa media2.c */

#define NUMEROS 10

float med (int a[], int t);
```

Listagem 2.5: med.c

```
#include "media2.h"

float med(int a[], int tam){
  int i, soma;
  float result;
  soma = 0;

  for(i=0; i<tam; i++){
    soma += a[i];
  }
  result = (float) soma / (float) tam;
  return (result);
}
```

2.2 Usando vetores como parâmetros de funções

Programas em C podem ser modulares, ou seja, podemos especificar definições em cabeçalhos (`.h`) e podemos colocar funções em subprogramas. Vamos, por exemplo, modularizar nosso programa de médias. Podemos colocar a constante em um arquivo de cabeçalho e a função que calcula a média em um outro arquivo, deixando apenas o corpo principal para a função main. Assim, o cabeçalho poderia ser como na listagem 2.4.

A função que calcula a média (med.c) recebe um vetor como parâmetro e o tamanho deste vetor, conforme descrito no cabeçalho (listagem 2.5).

Listagem 2.6: media2.c

```
#include <stdio.h>
#include <stdlib.h>
#include "media2.h"

int main (int argv, char *argc[]){
  int num[NUMEROS];   /* vetor de numeros */
  int i;
  float media;

  for(i=0; i<NUMEROS; i++){
    scanf("%d", &num[i]);
  }
  media = med(num, NUMEROS);
  printf("Media = %f\n", media);
  return 0;
}
```

Finalmente, o programa principal (media2.c) faz a chamada à função, passando os parâmetros (listagem 2.6).

Neste caso, para compilar o programa, deve-se passar ao compilador os códigos fonte a ser compilados para gerar o executável. Por exemplo:

```
gcc -O2 -Wall -ansi -pedantic -o <nome_do_executavel>
<arq1.c arq2.c ... arqN.c>
```

2.3 Operações com arrays

As operações sobre conjunto de dados representados em arrays podem ser de dois tipos: **operações sobre dados numéricos** e **operações sobre dados literais**. As operações sobre dados numéricos segue as regras da álgebra linear, uma vez que um array numérico é do tipo vetor, matriz ou multidimensional. As operações com dados literais são, via de regra, operações de manipulação

de caracteres ou cadeia de caracteres, tais como concatenação, segmentação, localização de um caractere na cadeia etc..

As operações sobre dados numéricos compreende:

- Multiplicação por escalares;
- Soma e subtração;
- Multiplicação entre arrays;
- Identidade;
- Inversibilidade; e
- Operações lógicas.

A multiplicação por escalares pode ser facilmente implementada, uma vez que a matriz resultante tem as mesmas dimensões da matriz de entrada (veja exemplo na listagem 2.7).

As operações sobre dados literais podem ser:

- Recuperação;
- Ordenação;
- Inserção;
- Eliminação;
- Substituição;
- Concatenação; e
- Segmentação.

Em C, existem várias funções para manipulação de caracteres e cadeias de caracteres (strings). Observe, no programa da listagem 2.8, como se pode atribuir este tipo de dado para variáveis. Se usarmos uma atribuição direta, do tipo:

```
a_linha = "Outra linha.";
```

Listagem 2.7: Multiplicação de matrizes: escmulmat.c

```
#include <stdio.h>
#include <stdlib.h>

#define LINHAS 4
#define COLUNAS 5

int main (int argv, char *argc[]){
  float Mat_A[LINHAS][COLUNAS]; /* matriz de entrada */
  float Mat_B[LINHAS][COLUNAS]; /* matriz resultante */
  float fator;      /* fator de multiplicacao */
  int i,j;

  fator = 1.0;
  printf("Entre o fator de multiplicacao: ");
  scanf("%f", &fator);

  for(i=0; i<LINHAS; i++){
    for(j=0; j<COLUNAS; j++){
      printf("linha %d, coluna %d: ", i, j);
      scanf("%f", &Mat_A[i][j]);
      Mat_B[i][j] = Mat_A[i][j] * fator;
    }
  }
  for(i=0; i<LINHAS; i++){
    for(j=0; j<COLUNAS; j++){
      printf("%2.2f  ", Mat_A[i][j]);
    }
    printf("\n");
  }

  printf("\n\n");
  for(i=0; i<LINHAS; i++){
    for(j=0; j<COLUNAS; j++){
      printf("%2.2f  ", Mat_B[i][j]);
    }
    printf("\n");
  }
  return 0;
}
```

Listagem 2.8: Exemplo de manipulação de cadeias de caracteres: prtstr.c

```
/* prtstr.c - Como atribuir cadeia de caracteres
   em variáveis */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argv, char *argc[]){
  char a_linha [80], mais_uma [80];
  char *strptr; /* Ponteiro para o conteudo de a_linha */

  /* Para atribuir uma cadeia de caracteres*/
  strcpy( a_linha, "Outra linha." );
  strcpy( mais_uma, a_linha );

  strptr = a_linha;

  /* Para adicionar mais caracteres  */
  strcat( a_linha, "bla-bla" );

  printf( "Uma linha de texto.\n");
  printf( "Variavel 'a_linha': %s\n", a_linha);
  printf( "Variable 'mais_uma': %s\n", mais_uma);
  printf( "Conteudo de 'a_linha': %s\n", strptr);
  return 0;

}
```

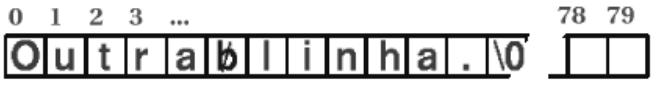


Figura 2.1: Representação de dados em C

O compilador acusará um erro de tipo. Assim, devemos usar a função `strcpy`. Para adicionar mais caracteres pode-se usar a função `strcat`, ambas descritas no cabeçalho padrão do C "string.h".

Neste exemplo dimensionamos as variáveis `a_linha` e `mais_uma` com o tamanho de 80 caracteres. Neste caso, o programa reservará 80 posições do tipo `char` para cada uma, mesmo que nem todas sejam usadas (veja figura 2.1).

As posições de 0 a 12, cada uma com 1 byte, da cadeia de caracteres conterão as representações ASCII de cada letra, espaço, ponto e um caractere `\0` no final:

45	75	74	72	61	20	6C	69	6E	68	61	2E	00

O restante conterá "zeros" representando o caractere `NULL`.

2.4 Linguagens funcionais e array

Embora o processamento de listas nas linguagens funcionais sejam extremamente fáceis, a implementação de *array* não o é. Na verdade, o conceito de array é extremamente imperativo, uma vez que tal estrutura é tratada como uma coleção de variáveis relacionadas aos endereços de memória e está fortemente vinculada à arquitetura da máquina de von Neumann.

Mesmo assim, é possível implementar arrays em Haskell de forma funcional. Em Haskell, um array é do tipo:

```
Ix a => (a,a) -> [(a,b)] -> Array a b
```

Se `a` é do tipo índice e `b` é do tipo "any", o array com índices em `a` e elementos em `b` é notado como `Array a b`. Assim,

```
array (1,3) [(1,20),(2,10),(3,15)]
```

corresponde ao vetor $[20, 10, 15]^T$ com índices $(1, 2, 3)$ e

```
array (0,2) [(0,20),(1,10),(2,15)]
```

é o mesmo vetor com índices $(0, 1, 2)$. Um array bidimensional pode ser escrito como:

```
array((1,1),(2,2)) [((1,1),2),((1,2),2),((2,1),5),((2,2),8)]
```

que representa a matriz:

$$\begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} = \begin{vmatrix} 2 & 2 \\ 5 & 8 \end{vmatrix}$$

2.5 Representação de dados e matrizes

Vimos que as matrizes podem ser representadas por estruturas como arranjo de dados (*array*). Além disso, conceitos de estruturas relacionadas com algoritmos funcionais, tais como grafos e árvores dependem de conceitos um pouco mais aprofundados de matrizes, principalmente os relacionados com álgebra booleana (matrizes binárias) e matrizes esparsas.

As matrizes são entidades matemáticas usadas para representar dados em duas dimensões de ordenação. Uma matriz pode sofrer vários tipos de operações, tais como adição, multiplicação, inversão etc..

2.5.1 Matrizes: conceitos gerais

Denominamos matriz de ordem $m \times n$ à ordenação bidimensional de m por n elementos dispostos em m linhas e n colunas. Por exemplo:

$$\mathbf{A} = \begin{vmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,n} \\ a_{2,1} & a_{2,2} & \cdots & a_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m,1} & a_{m,2} & \ldots & a_{m,n} \end{vmatrix}$$

1. A matriz onde $m = n$ é denominada de **matriz quadrada** e é representada por A_n ou $A_{(n,n)}$ e é dita de ordem n.

2. A matriz, onde $m \neq n$ é chamada de **matriz retangular** e é de ordem $m \times n$.

3. A matriz de ordem $m \times 1$ é chamada de **matriz coluna** ou **vetor coluna** e a matriz de ordem $1 \times n$ é chamada de **matriz linha** ou **vetor linha**, por exemplo:

$$\mathbf{A}_{(4,1)} = \begin{bmatrix} 3 \\ -2 \\ -5 \\ 4 \end{bmatrix} \quad ; \quad \mathbf{A}_{(1,3)} = \begin{bmatrix} 5 & 2 & -3 \end{bmatrix}$$

4. Cada elemento de uma matriz é representado por índices i, j (por exemplo: $a_{i,j}$) onde o primeiro índice (i) representa a linha e o segundo (j) representa a coluna do elemento da matriz.

5. Em uma matriz quadrada $A = [a_{i,j}]$ de ordem n tem uma **diagonal principal** que é dada pelos elementos $a_{i,j}$ onde $i = j$, ou seja, a diagonal principal de uma matriz é dada pelos elementos $(a_{1,1}, a_{2,2}, \ldots a_{n,n})$. Chama-se de **diagonal secundária** ao conjunto de elementos $a_{i,j}$ em que $i + j = n + 1$, ou seja, a diagonal secundária é dada pelos elementos $(a_{1,n}, a_{2,n-1}, \ldots, a_{n,1})$.

6. Uma matriz em que os elementos $a_{(i,j)}$ são nulos quando $i \neq j$, é chamada de **matriz diagonal**. Exemplo:

$$\mathbf{A} = \begin{vmatrix} a_{1,1} & 0 & 0 & \cdots & 0 \\ 0 & a_{2,2} & 0 & \cdots & 0 \\ 0 & 0 & a_{3,3} & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & a_{n,n} \end{vmatrix}$$

7. A matriz diagonal de ordem n em que os elementos $a_{i,j}$, para $i = j$, são iguais à unidade é chamada **matriz unidade** ou **matriz identidade** e é notada por $\mathbf{I}$ ou $\mathbf{I}_n$.

8. Uma matriz $m \times n$ com todos os elementos nulos é chamada de **matriz nula** e é notada por $\mathbf{0}_{m,n}$.

9. Dada uma matriz $A = [a_{i,j}]$ de ordem $m \times n$, e a matriz $B = [b_{i,j}]$, onde $b_{i,j} = -a_{i,j}$, dizemos que B é a **matriz oposta** de A, que pode ser representada por $-A$. Exemplo:

$$\mathbf{A} = \begin{vmatrix} 5 & 9 & -2 \\ 7 & -6 & 3 \end{vmatrix} \quad ; \quad \mathbf{B} = \begin{vmatrix} -5 & -9 & 2 \\ -7 & 6 & -3 \end{vmatrix} \quad ;e \quad \mathbf{B} = -\mathbf{A}$$

10. Dada uma matriz $A = [a_{i,j}]$ de ordem $m \times n$, existe uma matriz $B = [b_{i,j}]$ de ordem $n \times m$ em que $b_{i,j} = a_{j,i}$, então B é a **mmatriz transposta** de A que pode ser notada como A^T ou A^t. Exemplo:

$$\mathbf{A} = \begin{vmatrix} 1 & -5 & 4 \\ 6 & -3 & 2 \\ 3 & 1 & 1 \\ -2 & 3 & 5 \end{vmatrix} \quad ; \quad \mathbf{B} = \begin{vmatrix} 1 & 6 & 3 & -2 \\ -5 & -3 & 1 & 3 \\ 4 & 2 & 1 & 5 \end{vmatrix} \quad ;e \quad \mathbf{B} = \mathbf{A}^T$$

11. Uma matriz $A = [a_{i,j}]$ é denominada **matriz simétrica** se, e somente se, os elementos $a_{i,j} = a_{j,i}$ para $i \neq j$.

12. A matriz quadrada $A_{(n)}$ é chamada de **matriz triangular superior**, quando os termos $a_{i,j}$ da matriz, para $i < j$, são iguais a zero. Quando os termos $a_{i,j}$ da matriz, para $i > j$, são nulos. a matriz é chamada de **matriz triangular inferior**.

2.5.2 Operações com matrizes

A manipulação de dados organizados em estruturas, uniões, array, listas e listas de listas podem ser do mesmo tipo de operações com as matrizes. As principais operações com matrizes são: adição de matrizes, multiplicação por escalares, multiplicação de matrizes e transposição de matrizes.

Adição de matrizes

A adição de duas matrizes de mesma ordem é resultante da adição termo a termo de cada matriz. A adição de duas matrizes A e B de ordem $m \times n$, definida como $A + B$, é uma matriz C de mesma ordem, onde $c_{i,j} = a_{i,j} + b_{i,j}$. Por exemplo, seja:

$$\begin{vmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{vmatrix} + \begin{vmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \end{vmatrix} = \begin{vmatrix} a_{1,1} + b_{1,1} & a_{1,2} + b_{1,2} \\ a_{2,1} + b_{2,1} & a_{2,2} + b_{2,2} \end{vmatrix}$$

A subtração de duas matrizes A e B, definida por $A - B$ pode ser expressa pela diminuição termo a termo das duas matrizes ou pela soma da matriz A pela oposta de B, definida como $A + (-B)$.

Propriedades da adição de matrizes: sejam A, B e C matrizes da mesma ordem, então

1. $A + (B + C) = (A + B) + C$;

2. $A + B = B + A$;

3. Seja 0 uma matriz nula de mesma ordem, então $A + 0 = A$;

4. $A + (-A) = 0$.

Multiplicação por escalares

Seja $A = [a_{i,j}]$ e α um escalar, o produto αA é uma matriz $B = [b_{i,j}]$ tal que $b_{i,j} = \alpha a_{i,j}$. Por exemplo:

$$2 \times \begin{vmatrix} 4 & -2 & 6 \\ -3 & 5 & 1 \\ 2 & -1 & 4 \end{vmatrix} = \begin{vmatrix} 8 & -4 & 12 \\ -6 & 10 & 2 \\ 4 & -2 & 8 \end{vmatrix}$$

Propriedades da multiplicação por escalares: sejam α e β dois escalares e A e B duas matrizes de mesma ordem, então

1. $(\alpha\beta)A = \alpha(\beta A)$;

2. $(\alpha + \beta)A = \alpha A + \beta A$;

3. $\alpha(A + B) = \alpha A + \alpha B$; e

4. $1A = A$.

Multiplicação de matrizes

Dadas uma matriz A de ordem $m \times n$ e B de ordem $n \times p$, o produto das duas é uma matriz C de ordem $m \times p$, tal que:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k}\, b_{k,j}$$

Por exemplo, considere as matrizes $A_{(2\times3)}$ e $B_{(3\times4)}$:

$$\mathbf{A} = \begin{vmatrix} 2 & -1 & 4 \\ 3 & 2 & -2 \end{vmatrix} \quad \text{e} \quad \begin{vmatrix} 3 & 1 & 2 & -1 \\ -2 & 2 & 1 & 3 \\ 4 & -1 & -3 & 0 \end{vmatrix}$$

O produto de $A \times B$ é uma matriz $C_{(2\times4)}$, onde

$$
\begin{aligned}
c_{1,1} &= a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} + a_{1,3} \times b_{3,1} \\
&= 2 \times 3 + (-1) \times (-2) + 4 \times 4 \\
&= 24 \\
c_{1,2} &= a_{1,1} \times b_{1,2} + a_{1,2} \times b_{2,2} + a_{1,3} \times b_{3,2} \\
&= 2 \times 1 + (-1) \times 2 + 4 \times (-1) \\
&= -4 \\
c_{1,3} &= a_{1,1} \times b_{1,3} + a_{1,2} \times b_{2,3} + a_{1,3} \times b_{3,3} \\
&= 2 \times 2 + (-1) \times 1 + 4 \times (-3) \\
&= -9 \\
c_{1,4} &= a_{1,1} \times b_{1,4} + a_{1,2} \times b_{2,4} + a_{1,3} \times b_{3,4} \\
&= 2 \times (-1) + (-1) \times 3 + 4 \times 0 \\
&= -5
\end{aligned}
$$

$$
\begin{aligned}
c_{2,1} &= a_{2,1} \times b_{1,1} + a_{2,2} \times b_{2,1} + a_{2,3} \times b_{3,1} \\
&= 3 \times 3 + 2 \times (-2) + (-2) \times 4 \\
&= -3 \\
c_{2,2} &= a_{2,1} \times b_{1,2} + a_{2,2} \times b_{2,2} + a_{2,3} \times b_{3,2} \\
&= 3 \times 1 + 2 \times 2 + (-2) \times (-1) \\
&= 9 \\
c_{2,3} &= a_{2,1} \times b_{1,3} + a_{2,2} \times b_{2,3} + a_{2,3} \times b_{3,3} \\
&= 3 \times 2 + 2 \times 1 + (-2) \times (-3) \\
&= 14 \\
c_{2,4} &= a_{2,1} \times b_{1,4} + a_{2,2} \times b_{2,4} + a_{2,3} \times b_{3,4} \\
&= 3 \times (-1) + 2 \times 3 + (-2) \times 0 \\
&= 3
\end{aligned}
$$

resultando na matriz

$$
\mathbf{C} = \left| \begin{array}{cccc} 24 & -4 & -9 & -5 \\ -3 & 9 & 14 & 3 \end{array} \right|
$$

Nota: As operações com matrizes ainda compreendem a inversão de matrizes e o cálculo do determinante, que são operações importantes para a resolução de sistemas lineares.

2.5.3 Matrizes booleanas

Matrizes booleanas são conjuntos de dados organizados sob a forma de arranjo bidimensional de zeros e uns. Além das operações normais da álgebra

sobre estas matrizes, elas podem sofrer operações da álgebra de Boole, ou seja, operações lógicas do tipo `AND`($\wedge$) e `OR` ($\vee$). Estas operações são denominadas de multiplicação lógica e soma lógica, respectivamente.

As operações lógicas (ou binárias) são efetuadas sobre os valores 0 e 1, sendo que a multiplicação lógica ($\wedge$) retorna o mínimo entre os operandos, ou seja, $x \wedge y = \min(x, y)$ e a soma lógica ($\vee$) retorna o máximo entre os operandos, ou seja, $x \vee y = \max(x, y)$ (veja tabela 2.2).

x	**y**	**x** $\wedge$ **y**
0	0	0
0	1	0
1	0	0
1	1	1

x	**y**	**x** $\vee$ **y**
0	0	0
0	1	1
1	0	1
1	1	1

Tabela 2.2: Tabelas lógicas `AND` e `OR`.

A multiplicação e soma lógicas de matrizes (operações `AND` e `OR`) segue as mesmas regras da soma de duas matrizes, ou seja, para duas matrizes de mesma ordem, estas operações são feitas termo a termo. Entretanto o **produto lógico** de duas matrizes de ordens apropriadas segue a regra da multiplicação de matrizes, com as operações lógicas (`AND` e `OR`. Assim, o produto lógico de duas matrizes booleanas, notado por $A \times B$, é dado por:

$$A \times B = C \quad | \quad c_{i,j} = \bigvee_{k=1}^{m} (a_{i,k} \wedge b_{k,j})$$

ou seja, considere as matrizes:

$$\mathbf{A} = \begin{vmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \end{vmatrix} \qquad e \qquad \mathbf{B} = \begin{vmatrix} b_{1,1} & b_{1,2} \\ b_{2,1} & b_{2,2} \\ b_{3,1} & b_{3,2} \end{vmatrix}$$

o produto lógico de

$$\mathbf{A} \times \mathbf{B} = \mathbf{C} = \begin{vmatrix} c_{1,1} & c_{1,2} \\ c_{2,1} & c_{2,2} \end{vmatrix} \quad ,onde \begin{array}{rcl} c_{1,1} & = & (a_{1,1} \wedge b_{1,1}) \vee (a_{1,2} \wedge b_{2,1}) \vee (a_{1,3} \wedge b_{3,1}) \\ c_{1,2} & = & (a_{1,1} \wedge b_{1,2}) \vee (a_{1,2} \wedge b_{2,2}) \vee (a_{1,3} \wedge b_{3,2}) \\ c_{2,1} & = & (a_{2,1} \wedge b_{1,1}) \vee (a_{2,2} \wedge b_{2,1}) \vee (a_{2,3} \wedge b_{3,1}) \\ c_{2,2} & = & (a_{2,1} \wedge b_{1,2}) \vee (a_{2,2} \wedge b_{2,2}) \vee (a_{2,3} \wedge b_{3,2}) \end{array}$$

Por exemplo, sejam:

$$\mathbf{A} = \begin{vmatrix} 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{vmatrix} \quad \text{e} \quad \mathbf{B} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{vmatrix}$$

então, a soma, multiplicação e produto lógicos de A e B são, respectivamente:

$$\mathbf{A}\wedge\mathbf{B} = \begin{vmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{vmatrix} \quad , \quad \text{A}\vee\mathbf{B} = \begin{vmatrix} 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{vmatrix} \quad \text{e} \quad \text{A}\times\mathbf{B} = \begin{vmatrix} 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 \end{vmatrix}$$

2.5.4 Matrizes esparsas

Matrizes esparsas são conjuntos de dados organizados sob a forma de arranjo bidimensional, onde grande parte dos valores são nulos. Tais matrizes podem ser representadas na forma esparsa ou em uma forma compacta. A importância do estudo das formas de representação destas matrizes nas formas esparsas e compactas está na compactação de dados organizados de forma espalhada (esparsa).

Vamos tomar, por exemplo, uma matriz esparsa:

$$\mathbf{A} = \begin{vmatrix} 1 & 0 & 0 & 5 & 0 & 0 & 0 & 0 \\ 0 & -2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 15 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -7 \\ 0 & 0 & 0 & 0 & -9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \end{vmatrix}$$

A matriz $A_{(6\times8)}$ representa um conjunto esparsos de dados que podem ser descritos como um array que ocupa 48 posições de memória:

```
int A[6][8] = {
{1,0,0,5,0,0,0},
{0,-2,0,0,0,0,0,0},
{0,0,0,0,15,0,0,0},
{0,0,0,0,0,0,0,-7},
{0,0,0,0,-9,0,0,0},
{0,0,0,0,0,0,0,4}
}
```

Entretanto, esta matriz, por ser esparsa, pode ser representada de forma mais racional e compacta. Para isso, é necessário estabelecer algumas regras.

Dada uma matriz esparsa A de ordem $m \times n$, existe uma matriz compacta B de ordem $(k+1) \times 3$, onde k representa o número de elementos não-zero da matriz:

- O elemento da primeira linha e primeira coluna representa o número de linhas da matriz, ou seja, $b_{1,1} = m$;
- O elemento da primeira linha e segunda coluna representa o número de colunas da matriz, ou seja $b_{1,2} = n$;
- O elemento da primeira linha e terceira coluna representa o número de elementos não-zero da matriz, ou seja, $b_{1,3} = k$;
- A partir da segunda linha da matriz compacta, cada linha representa a posição de um elemento da matriz, ou seja, sua linha, sua coluna e o próprio elemento.

A matriz do exemplo acima ($A_{(6\times8)}$) tem 7 elementos não zero, logo, sua forma compacta deverá ser uma matriz $B_{(8\times3)}$ e, portanto:

$$\mathbf{B} = \begin{vmatrix} 6 & 8 & 7 \\ 1 & 1 & 1 \\ 1 & 4 & 5 \\ 2 & 2 & -2 \\ 3 & 5 & 15 \\ 4 & 8 & -7 \\ 5 & 5 & -9 \\ 6 & 8 & 4 \end{vmatrix}$$

Assim, a primeira linha ([6 8 7]) indica que a matriz é de ordem 6×8 e que possue 7 elementos não-zero. A segunda linha ([1 1 1]) indica que $a_{1,1} = 1$; a terceira indica que $a_{1,4} = 5$; a quarta que $a_{2,2} = -2$; e assim por diante.

O array para representar esta matriz ocupará, portanto 24 posições de memória, ou seja, a metade da matriz esparsa:

```
int B[8][3] = {
{6,8,7},
{1,1,1},
```

```
{1,4,5},
{2,2,-2},
{3,5,15},
{4,8,-7},
{5,5,-9},
{6,8,4}
}
```

Evidentemente, para matrizes muito maiores e mais esparsas (o que é freqüente em imagens) pode-se economizar muita memória ou espaço de armazenagem de dados.

2.6 Tipos de dados definidos pelo usuário

A linguagem C possui vários tipos de dados que podem ser definidos pelo programador. Podemos ter um conjunto de registros, com suas etiquetas (*labels*) definidas por tipos diferentes que podem ser referenciados sob um mesmo nome (estruturas); por bits (campo de bits); por tipos em uma mesma porção da memória (uniões); por lista de símbolos (enumeração) ou por novos tipos definidos pelo usuário (TDA) pelo uso de `typedef`.

Destes, duas formas são muito úteis para a representação de coleção de dados relacionados entre si. Uma delas é a declaração de um tipo estrutura (`struct`)e outra é do tipo união (`union`), onde os ítens são referenciados por um identificador chamado membro.

2.6.1 Estruturas

Uma estrutura é um conjunto de dados organizados em uma declaração `struct` onde cada dado é referenciado por um identificador. Comparativamente a um banco de dados, uma estrutura é algo semelhante a um "registro" e cada membro da estrutura pode ser comparado a um "campo". Por exemplo:

```
struct nome {
  char primeiro[10];
  char meio[10];
  char ultimo[20];
};
```

O exemplo acima cria uma estrutura chamada "nome" com os membros: "primeiro", "meio" e "ultimo". Pode-se, então, criar variáveis com o mesmo

tipo, usando-se a declaração `struct nome n;`. Nesta declaração é criada uma variável "n" do mesmo tipo da estrutura "nome".

Observe que, quando uma estrutura é definida, está sendo definido apenas um tipo complexo de variável e não a variável, propriamente dita. De forma semelhante, em uma linguagem SQL de banco de dados, tal estrutura é equivalente a:

```
CREATE TABLE nome (
  primeiro char(10) default NULL,
  meio     char(10) default NULL,
  ultimo   char(20) default NULL,
);
```

Uma estrutura pode conter um membro do tipo de outra estrutura. Por exemplo, considere a seguinte estrutura:

```
struct agenda {
  struct nome reg;
  char endereco[60];
  char cidade[20];
  char uf[2];
  char fone[16];
};
```

Neste caso, temos uma estrutura dentro de outra, ou seja, o primeiro membro da estrutura "agenda" é uma variável chamada "reg" do tipo estrutura "nome". O compilador vai alocar memória para todas os elementos da estrutura, contendo:

nome	→	40 bytes
endereco	→	60 bytes
cidade	→	20 bytes
uf	→	2 bytes
fone	→	16 bytes

A partir desta estrutura, pode-se, a qualquer momento, criar uma variável dimensional para referenciar 200 registros do tipo "agenda", por exemplo:

```
struct agenda ag[200];
```

Qualquer membro da variável `ag[200]` pode ser referenciado por um construtor "." (ponto). Assim:

```
printf ("%s - %s\n", ag[2].reg.primeiro, ag[2].fone);
```

imprimirá o terceiro registro "primeiro" de "reg" da estrutura "ag" e seu respectivo "fone".

A estrutura também pode ser declarada no mesmo momento em que é definida, por exemplo:

```
struct ender{
  char nome[30];
  char rua[40];
  char cidade[20];
  char estado[3];
  unsigned long int cep;
}info_ender, outra_ender;
```

neste caso as variáveis `info_ender` e `outra_ender` são variáveis declaradas do tipo `ender`. No caso de ser necessário apenas uma variável do tipo da estrutura, o nome da estrutura torna-se desnecessário, por exemplo:

```
struct{
  char nome[30];
  char rua[40];
  char cidade[20];
  char estado[3];
  unsigned long int cep;
}info_ender;
```

que declara uma variável `info_ender` do tipo da estrutura que a precede.

Com a finalidade de compreender como uma estrutura ocupa a memória, vamos considerar uma estrutura e verificar como os dados ficam armazenados na memória (veja listagem 2.9).

Listagem 2.9: Exemplo de estrutura: est-ender.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct mista{
  int campo1;
  double campo2;
  char campo3[10];
  float campo4;
  int campo5;
};

int main(int argv, char *argc[]){

  struct mista var1={2,3.65,"abcdefghijk",92.3,5};

  printf ("campo 1 (int): %d\n", var1.campo1);
  printf ("Endereco de 'campo1': %x\n", &var1.campo1);
  printf ("\ncampo 2 (double): %f\n", var1.campo2);
  printf ("Endereco de 'campo2': %x\n", &var1.campo2);
  printf ("\ncampo 3 (char *): %s\n", var1.campo3);
  printf ("Endereco de 'campo3': %x\n", &var1.campo3[0]);
  printf ("\ncampo 4 (float): %f\n", var1.campo4);
  printf ("Endereco de 'campo4': %x\n", &var1.campo4);
  printf ("\ncampo 5 (int): %d\n", var1.campo5);
  printf ("Endereco de 'campo5': %x\n\n", &var1.campo5);
  return 0;
}
```

Nesta estrutura temos vários tipos de membros (`int, double, char, float`) e, depois de compilado, o programa imprime os valores e endereços de cada membro. Desta forma pode-se observar o quanto cada membro ocupa na memória. A estrutura "mista" ocupa, na memória, o espaço somado de todos os seus membros.

Nota: Observe que o `campo3` tem 11 elementos e apenas 10 posições alocadas. A compilação com `'-Wall -ansi -pedantic'` retornará as mensagens de *warning* mas o programa será compilado. Ao rodar o programa voce poderá observar que o "k" foi desprezado.

As estruturas podem ser passadas para funções de duas formas. Uma delas é por valor e outra por referência. Veja, por exemplo, a listagem 2.10.

Listagem 2.10: Passando estruturas para funções: cards.c

```
#include <stdio.h>
#include <stdlib.h>

struct cartas {
  int indice;
  char naipe;
};
void imprime_por_valor (struct cartas qual);
void imprime_por_refer (struct cartas *p);

int main(){
  struct cartas rei_paus = {13,'p'};
  struct cartas dama_copas = {12,'c'};
  imprime_por_valor (rei_paus);
  imprime_por_refer (&dama_copas);
  return 0;
}

void imprime_por_valor (struct cartas qual){
  printf("%i de %c\n", qual.indice, qual.naipe);
}

void imprime_por_refer (struct cartas *p){
  printf("%i de %c\n", p->indice, p->naipe);
}
```

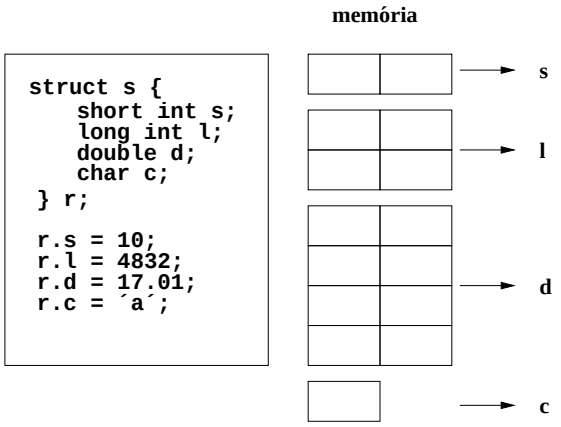


Figura 2.2: Estrutura e espaço de memória

A primeira função recebe por valor na variável "qual" como uma estrutura do tipo "cartas". Assim, a referência é feita pelos valores de `qual.indice` e `qual.naipe` em relação à variável da chamada da função (no caso `rei_paus`).

Na segunda função, o valor é passado por referência pelo apontador (`*p`) do endereço da variável (`&dama_copas`) que é passada na chamada da função. A referência é feita por `p->indice` e `p->naipe`. A notação `p->` é correspondente à (`*p`), ou seja, `p->indice` é equivalente a `(*p).indice`.

2.6.2 Uniões

Outro tipo de organização de dados, como as estruturas, são as uniões (`union`), que são representadas e referenciadas da mesma forma que elas. Entretanto, o conceito de união é fundamentalmente diferente do conceito de estrutura, no que se refere à organização dos membros na memória.

Em uma estrutura, o espaço de memória é ocupado seqüencialmente por cada um dos membros da estrutura, enquanto que no caso de uma união, o mesmo espaço de memória pode ser ocupado por diversas variáveis de tamanhos diferentes. Por exemplo, na figura 2.2 temos um exemplo de como se organizam os membros de uma estrutura na memória e na figura 2.3 os membros de uma união na memória.

No tipo `union`, uma variável pode conter objetos de diferentes tipos e tamanhos, mas não ao mesmo tempo. O compilador se incumbe de manter a referência dos requisitos de tamanho e alinhamento da memória. Assim,

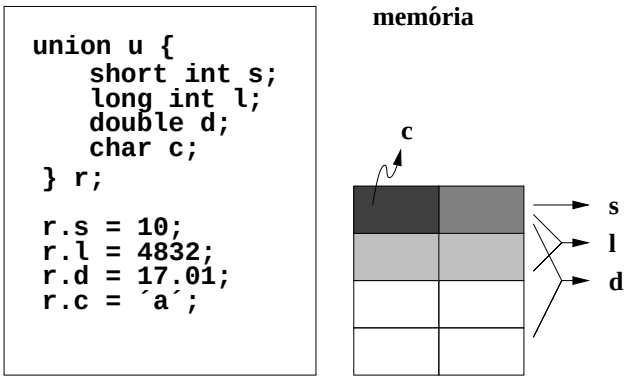


Figura 2.3: Uniões e espaço de memória

este tipo de estrutura permite que se manipule tipos diferentes de dados em uma mesma área da memória, sem que se tenha que se preocupar com as dependências de arquitetura das máquinas no desenvolvimento do programa.

2.6.3 Campos de bits

Embora os tipos `struct` e `union` sejam os conceitos mais usados em estruturas de dados, existe ainda um outro tipo que tem uma determinada importância na comunicação entre portas de hardware. Por exemplo, um adaptador serial de comunicação entre dois equipamentos está organizado em uma estrutura de **campo de bits** de 1 byte, como:

Bit	**Se ligado**
0	clear-to-send alterado
1	data-set-ready alterado
2	pico de portadora
3	rcpt alterado
4	cts
5	dsr
6	ring
7	recepção de linha

A estrutura de bits para representar o estado de comunicação pode ser expressa da seguinte forma:

```
struct status_tipo{
  unsigned var_cts  :1;
  unsigned var_dsr  :1;
  unsigned carrier  :1;
  unsigned var_rcpt :1;
  unsigned cts      :1;
  unsigned dsr      :1;
  unsigned call     :1;
  unsigned line_rcp :1;
} status
```

A estrutura pode ser usada para determinar quando a aplicação pode ou não enviar dados pelo canal de comunicação, por exemplo:

```
status =  get_status();

if(status.cts){
  printf ("livre para enviar\n");
}
if(status.dsr){
  printf("dados prontos\n");
}
```

2.7 Exercícios

1. Em C, a função `scanf` é usada para ler dados de entrada do teclado e gravar estes dados em variáveis. Por exemplo:

```
#include <stdio.h>
#include <stdlib.h>

#define NUMEROS 10

int main (int argv, char *argc[]){
   int num[NUMEROS];    /* vetor de numeros */
   int i, total;
   float media;
```

```
    total = 0;
    for(i=0; i<NUMEROS; i++){
      scanf("%d", &num[i]);
      total += num[i];
    }
    media = (float) total / (float) NUMEROS;
    printf("Media = %f\n", media);
    return 0;
}
```

(a) Escreva um programa que receba como entrada o valor n e os valores de duas matrizes A e B $n \times n$ e, como saída, a soma $A + B$ e os produtos $A \cdot B$ e $B \cdot A$.

(b) Escreva um programa que receba como entrada a dimensão e os elementos de uma matriz A e apresente como saída a matriz transposta ($[A]^T$).

2. Usando os conceitos de Array e as propriedades das matrizes, escreva um programa que receba dois vetores e retorne o produto entre eles, se forem apropriados.

3. A partir de um arquivo que contenha uma matriz de ordem $n \times m$, binária e esparsa, determine o formato deste arquivo e construa um programa para:

   (a) achar a transposta a partir da transposição de matrizes;

   (b) achar a transposta por meio da matriz compacta.

   Qual algoritmo é mais eficiente? Teste e Justifique.

Capítulo 3

Listas

Alguns tipos de dados podem ser organizados em uma cadeia, seja de números ou de caracteres que denominamos de lista. As listas podem ser implementadas de várias maneiras, tanto em uma linguagem procedural como em uma linguagem funcional. As listas são estruturas dinâmicas, em oposição aos vetores (arrays) que são estáticas e contém um conjunto específico de dados. Virtualmente, uma lista pode conter infinitos elementos.

As operações sobre listas são limitadas ao tipo de dados que uma lista contém. Por exemplo, listas de dados numéricos podem ser tratadas matematicamente, enquanto listas de caracteres podem ser ordenadas, concatenadas etc..

Elas podem ser classificadas em pilhas, filas e listas ligadas, dependendo do tipo de acesso que pode ser feito aos dados que ela contém:

- Listas LIFO (*Last In, First Out*) ou pilhas;
- Listas FIFO (*First In, First Out*) ou Filas; e
- Em qualquer posição (listas ligadas).

3.1 Definições e declarações

Podemos definir uma lista como um conjunto ordenado de elementos, geralmente do mesmo tipo de dado. Por exemplo, `[20,13,42,23,54]` é uma lista numérica de inteiros; `[‘a’,‘d’,‘z’,‘m’,‘w’,‘u’]` é uma lista de caracteres; e `["banana","abacaxi","pera"]` é uma lista de frutas.

Cada linguagem tem uma sintaxe específica para organizar os dados em forma de listas. Em C/C++ uma lista pode ser descrita por uma estrutura

ou uma enumeração. Por exemplo, considere uma linha de texto como sendo uma lista de caracteres:

```
/* Neste exemplo, "struct linebuffer" é uma estrutura que
 * pode armazenar uma linha de texto. */

  struct linebuffer{
    long size;
    char *buffer;
  };
```

Nas linguagens funcionais, a declaração de listas usa os construtores: `[ ]` e `:` , onde `[]` representa uma lista vazia. Assim, uma lista pode ser declarada como `[1,2,3]` ou `(1:2:3:[])`. Além disso, pode-se declarar uma lista usando-se uma expressão conhecida como *list comprehension*:

```
[x | x <- [1,2,3]]
```

Estas expressões derivam do cálculo-λ e podem ser escritas como: `(\x -> x) [1,2,3]` que, em cálculo-λ é escrita como:

$$\lambda x.x\{1,2,3\}$$

Liguagens como HASKELL , Clean, CAML e Erlang, por exemplo, suportam tanto declarações explícitas de listas, tais como `[1,2,3,4,5]` quanto implícitas, ou seja, `[1..5]`. Este último caso representa uma progressão aritmética (PA) de razão 1. Assim, por exemplo:

```
[2,4..20]      -- representa [2,4,6,8,10,12,14,,16,18,20] ou
               -- uma PA de 2 a 20 de razão 2
[1,4..15]      -- representa [1,4,7,10,13], ou uma PA de 1 a
               -- 15 de razão 3.
```

As listas implícitas podem ser declaradas de forma finita (`[1..20]`, `[2,4..100]`, `['a'..'m']` etc.) ou de forma infinita: (`[1..]`, `[2,4..]` etc.).

Existe, ainda, uma maneira de representar uma lista por `(x:xs)`[1] que é uma meta-estrutura onde x representa o primeiro elemento da lista (chamado de "cabeça") e xs representa o restante da lista (chamado de "cauda").

[1] Lê-se 'xis' e 'xises'.

3.2 Operações com listas

As linguagens funcionais implementam algumas funções para operações sobre dados de uma lista. Por exemplo, a função `map`, em `map f L` aplica uma função `f` em uma lista `L`:

```
Prelude> map (^2) [1..10]
[1,4,9,16,25,36,49,64,81,100]
Prelude> map odd [1..10]
[True,False,True,False,True,False,True,False,True,False]
```

a função `filter` testa uma condição em uma lista:

```
Prelude> filter odd [1..10]
[1,3,5,7,9]
Prelude> filter (>10) [1..20]
[11,12,13,14,15,16,17,18,19,20]
Prelude> filter (>5) (filter (<10) [1..20])
[6,7,8,9]
```

Na tabela 3.1, encontram-se descritas algumas funções.

As operações sobre listas podem ser definidas em funções pelo usuário. Por exemplo, dada uma lista de elementos finitos, podemos gerar todas as combinações possíveis destes elementos. A listagem 3.1 mostra um exemplo de programa de permutações.

O módulo "Permuta"retorna todas as listas possíveis a partir de uma lista finita, por exemplo:

```
Prelude> :l Permuta
Permuta> perms "abcd"
["abcd","bacd","bcad","bcda","acbd","cabd","cbad","cbda","acdb",
"cadb","cdab","cdba","abdc","badc","bdac","bdca","adbc","dabc",
```

Função	Descrição	Sintaxe	Exemplo
`concat`	Concatena duas listas	`concat [<lista 1>] [<lista 2>]`	`concat [1..3] [6..8]` Resultado: `[1,2,3,6,7,8]`
`filter`	Aplica uma condição e retorna os elementos que casam com o padrão	`filter <padrão> [<lista>]`	`filter odd [1..10]` Resultado: `[1,3,5,7,9]`
`map`	Aplica uma função em todos os elementos de uma lista	`map <função> [<lista>]`	`map (*2) [1,2,3,4]` Resultado: `[2,4,6,8]`
`take`	Toma n elementos de uma lista	`take n [<lista>]`	`take 5 [10,20..]` Resultado: `[10,20,30,40,50]`
`zip`	Forma duplas com os elementos de duas listas	`zip [<lista 1>] [<lista 2>]`	`zip [1,2,3] ['a','b','c']` Resultado: `[(1,'a'),(2,'b'),(3,'c')]`

Tabela 3.1: Exemplos de funções para operações sobre listas em HASKELL .

Listagem 3.1: Programa Permuta.hs para explicar operações sobre listas em Haskell.

```
module Permuta (
    perms
    ) where
perms :: [a] -> [[a]]
perms []             = [[]]
perms (x:xs)    = concat (map (inter x) (perms xs))

inter x []                    = [[x]]
inter x ys'@(y:ys)  = (x:ys') : map (y:) (inter x ys)
```

```
"dbac","dbca","adcb","dacb","dcab","dcba"]
Permuta> perms [1..4]
[[1,2,3,4],[2,1,3,4],[2,3,1,4],[2,3,4,1],[1,3,2,4],[3,1,2,4],
[3,2,1,4],[3,2,4,1],[1,3,4,2],[3,1,4,2],[3,4,1,2],[3,4,2,1],
[1,2,4,3],[2,1,4,3],[2,4,1,3],[2,4,3,1],[1,4,2,3],[4,1,2,3],
[4,2,1,3],[4,2,3,1],[1,4,3,2],[4,1,3,2],[4,3,1,2],[4,3,2,1]]
```

A relação das linguagens funcionais com listas tem uma característica de eficiência quando se trata de listas infinitas. Por exemplo:`take 5 [1..]` retorna uma lista finita de 5 elementos (`[1,2,3,4,5]`). Isto se deve a um processo denominado de *lazy evaluation* que em inglês significa avaliação "preguiçosa", mas que poderia ser melhor traduzida por avaliação "esperta", ou avaliação sob demanda. Este processo gera a lista infinita até o ponto onde a computação é necessária. Por exemplo:

```
Prelude> takeWhile (<30) [10..]
[10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29]
```

Neste caso, dada uma lista infinita, cujo primeiro elemento é 10, a função `takeWhile` testa quais são menores que 30 e retorna a lista com estes elementos.

Vamos considerar o exemplo da listagem 3.2, que descreve uma função `primo` como:

```
primo :: Integral a => a -> Bool
primo n = (verif n == [])
      where
         verif n = [ x | x <- [2 .. n-1], n `mod` x == 0]
```

e a aplica em uma função `primos` que retorna os números primos de uma lista:

```
primos :: Integral a => [a] -> [a]
primos []     = []
primos (n:ns) = case (primo n) of True  -> n : primos ns
                                  False -> primos ns
```

Usando-se a função takeWhile podemos recuperar uma lista de primos conforme as condições. Neste caso, `primos (takeWhile (<1000) [1..])` retornará a lista de todos os números primos menores que 1000.

Listagem 3.2: Programa Primos.hs para encontrar os números primos em uma lista.

```
module Primos (
    primo, primos
    ) where

primo :: Integral a => a -> Bool
primo n = (fator n == [])
    where
        fator n = [ x | x <- [2 .. n-1], n `mod` x == 0]

primos :: Integral a => [a] -> [a]
primos []     = []
primos (n:ns) = case (primo n) of True  -> n : primos ns
                                  False -> primos ns
```

```
Prelude> :l Primos
Compiling Primos           ( Primos.hs, interpreted )
Ok, modules loaded: Primos.
*Primos> primos (takeWhile (<1000) [1..])
```

Nota: Este algorithmo não é o mais eficiente para cálculo de números primos. Na lista de exercícios voce deverá implementar o "crivo" de Erastóstenes.

3.3 Pilhas

Um tipo especial de listas, onde os elementos são ordenados como um empilhamento de dados se denomina **pilha** (*stack*, em inglês). As pilhas representam uma das estruturas mais importantes no processamento de dados. Embora sua estrutura seja bastante simples, as pilhas desempenham um importante papel nas liguagens de programação, pois são utilizadas como acumuladores em laços do programa e em chamadas de subrotinas ou funções.

Uma pilha pode ser definida como um conjunto ordenado de dados, no qual se pode inserir e retirar dados apenas em uma determinada ordem, ou seja em

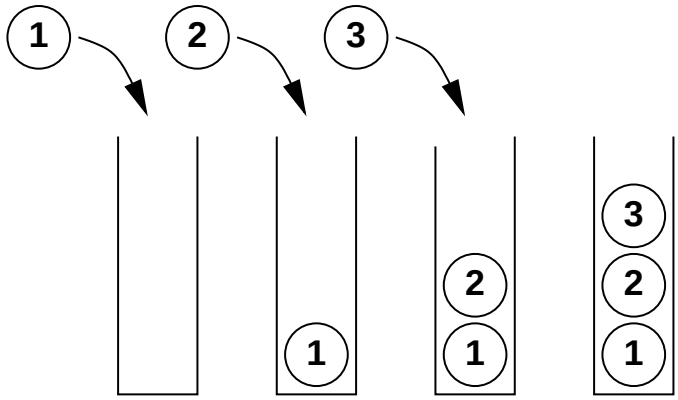


Figura 3.1: Representação de uma pilha de bolas numeradas

uma posição denominada "topo da pilha". Considere, por exemplo a figura 3.1

Na figura um temos um tubo onde podemos empilhar as bolas numeradas. Neste caso, cada bola vai para o topo da pilha. É claro que, para se retirar a segunda bola, é necessário retirar antes a terceira. Ou seja, o último elemento a entrar em uma pilha é o primeiro a sair. Por isto, a pilha recebe o nome de **LIFO**, do inglês *Last In First Out*.

3.3.1 Operações e TDA

Existem duas primitivas (operações básicas) que podem ser efetuadas sobre uma estrutura em pilha. Uma delas adiciona um elemento a uma pilha (`push`) e a outra retira um elemento de uma pilha (`pop`). Por exemplo:

```
/* insere o elemento 'i' em uma pilha 's' */
push (s,i);

/* passa o elemento no topo da pilha 's' para 'i' */
i = pop (s);
```

Uma pilha não tem tamanho finito e definido, assim, a operação `push` pode ser feita indefinidamente até que se esgote a memória, produzindo o que se denomina "estouro da pilha" (*stack overflow*). Ainda, a operação `pop` não pode ser aplicada a uma pilha vazia, pois não há elemento para ser desempilhado.

Listagem 3.3: TDA de uma pilha

```
abstract typedef <<t_elem>> STACK (t_elem);

abstract empty (s)
STACK (t_elem) s;
postcondition empty == (len (s) == 0);

abstract t_elem pop (s)
STACK (t_elem) s;
precondition  empty (s) == FALSE;
postcondition pop == first (s');
                s == sub (s', 1, len (s') - 1);

abstract push (s, elem)
STACK (t_elem) s;
t_elem elem;
postcondition s == <elem> + s';
```

Esta operação, em uma pilha vazia, causa um erro denominado de "colapso da pilha" (*stack underflow*).

Uma função `empty (s)` pode ser usada para testar e uma pilha está ou não vazia. Esta função retornará TRUE se a pilha estiver vazia ou FALSE no caso contrário.

Outra operação que se pode executar sobre pilhas é a função `stacktop`. Esta função retorna o topo da pilha sem remover o elemento. Ou seja:

```
i = stacktop (s);

/* equivale a: */
i = pop (s);
push (s,i);
```

A listagem 3.3 representa o TDA de uma pilha. A pilha pode ser usada em qualquer situação que requeira uma rotina do tipo LIFO em um determinado

Listagem 3.4: Exemplo de pilha em C

```
/* Define uma pilha de inteiros */

#define TAM_PILHA 100

struct pilha {
  int topo;
  int elems[TAM_PILHA];
};
```

padrão de agrupamento de dados como, por exemplo, chamada de rotina, funções, operações com parênteses etc..

3.3.2 Implementação e Exemplos em C

Com a finalidade de implementar a solução de um problema que utilize uma estrutura de dados em pilhas, vamos ver como se pode representar uma pilha em C. Antes, porém, é preciso ressaltar a diferença fundamental entre vetores e pilhas. Um vetor é um conjunto ordenado de elementos com um número fixo, enquanto uma pilha é um conjunto dinâmico, cujo tamanho está sempre mudando conforme se opera sobre sua estrutura.

Na prática, pode-se contornar esta característica declarado-se um espaço finito reservado para uma pilha, desde que tenha tamanho sufuciente para armazenar o tamanho máximo necessário para os seus elementos. Uma extremidade é o início da pilha, enquanto a outra é o topo da pilha. Assim, é necessário que se tenha um outro campo que indique em que ponto da pilha se encontra o topo durante a execução do programa (ver listagem 3.4).

O código da listagem 3.4 descreve uma pilha de inteiros de tamanho 100. Mas, se for necessáario descrever uma pilha com elementos diferentes de inteiros podemos especificar `float elems[TAM_PILHA]` para números com ponto flutuante ou `char elems[TAM_PILHA]` para caracteres, ou ainda qualquer outro tipo. Mas, podemos criar uma pilha que possa armazenar qualquer tipo de dados (veja listagem 3.7).

Listagem 3.5: Exemplo de pilha genérica em C

```
/* Define uma pilha que pode conter inteiros,
 * floats e strings */

#define TAM_PILHA 100
#define INTEIRO 1
#define DECIMAL 2
#define CADEIA  3

struct elem_pilha {
  int elem_tipo; /* pode ser inteiro, float ou string */
  union {
    int ival;
    float fval;
    char *sval; /* ponteiro para string */
  } elemento;
};

struct pilha {
  int topo;
  struct elem_pilha elems[TAM_PILHA];
};
```

Listagem 3.6: pilha.h

```
#include <stdlib.h>
#include <stdio.h>

typedef struct Pilha PL;

struct Pilha{
  int top, *elem;
  int max ; /* tamanho máximo da pilha */
};

void PUSH(PL *P,int A);

int POP(PL *P, int A);
```

As funções `push` e `pop` podem ser implementadas com facilidade. Por exemplo, considere um cabeçalho (pilha.h) que defina uma estrutura de pilha de inteiros (listagem 3.8).

A função `push` pode ser descrita como:

```
#include "pilha.h"

void PUSH(PL *P,int A){
  if (P->top < P->max - 1){
    P->elem[++(P->top)] = A;
  }
  else{
    printf("Pilha Cheia\n");
  }
}
```

E a função `pop` pode ser:

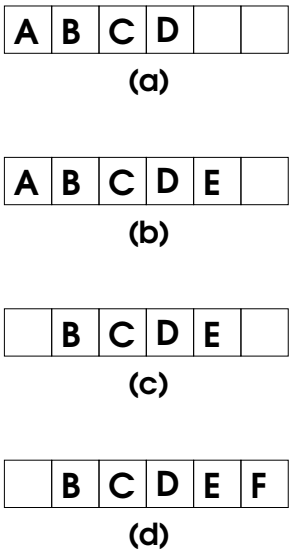


Figura 3.2: Exemplo inserção e remoção de elementos de uma estrutura em fila

```
#include "pilha.h"

int POP(PL *P, int A){
  A=P->elem[--(P->top)];
  return A;
}
```

3.4 Filas

Outro tipo de estruturas em listas são as **filas**. Filas são representações de uma estruturas seqüencial, como as pilhas, mas com a diferença de que os elementos podem ser inseridos na fila por uma extremidade chamada fim da fila e podem ser retirados pela outra extremidade ou início da fila. Considere, por exemplo a figura 3.2. A figura 3.2(a) representa uma fila; 3.2(b) a inserção de um elemento na fila; 3.2(c) a remoção de um elemento; e 3.2(d) outra inserção.

Pode-se notar que, após remover um elemento do início da fila, o novo início é o segundo elemento. E a inserção de um novo elemento, no final da fila, passa a ser o novo final. Assim, o primeiro elemento a entrar em uma fila

é o primeiro a sair. Por isto, a fila recebe o nome de **FIFO**, do inglês *First In First Out*.

3.4.1 Filas: operações e TDA

Assim como nas pilhas, existem duas operações básicas que podem ser efetuadas sobre uma estrutura de fila. Uma delas adiciona um elemento no final da fila (insere) e a outra retira um elemento do início de uma fila (remove). Por exemplo:

```
/* insere o elemento 'i' em uma fila 'q' */
insere (q,i);

/* passa o elemento no início da fila 'q' para 'i' */
i = remove (q);
```

De modo semelhante às pilhas, as filas não têm tamanho definido (depende de quanto queremos alocar para esta estrutura) logo, a operação `insere` pode ser feita até que se atinja o limite de memória ou um tamanho pode ser estabelecido, provocando um "estouro da fila" (*queue overflow*). Obviamente, a operação `remove` não pode ser aplicada a uma fila vazia, pois não há elemento para ser removido. Esta operação causa um erro denominado de "colapso da fila" (*queue underflow*). Uma função `empty (q)`, que já vimos para pilhas, pode ser usada para testar se uma fila está ou não vazia.

A representação de uma fila como TDA é feita por meio de um `t_elem`, que indica o tipo de elemento da fila, e a parametrização do tipo da fila, segundo `t_elem` simples (veja listagem 3.7)

Uma pilha pode ser usada em qualquer situação que requeira uma rotina do tipo FIFO em um determinado padrão de agrupamento de dados.

3.4.2 Implementação em C

Podemos implementar uma fila, usando uma estrutura, da mesma maneira como o fizemos com a pilha. Entretanto, esta estrutura deve conter variáveis que apontam para o início e o final da fila, como as pilhas as têm para indicar o topo da pilha. O cabeçalho (`.h`) de uma fila pode ser implementado como na listagem 3.8

Listagem 3.7: TDA para uma fila

```
abstract typedef <<t_elem>> QUEUE (t_elem);

abstract empty (q)
QUEUE (t_elem) q;
postcondition empty == (len (q) == 0);

abstract t_elem remove (q)
QUEUE (t_elem) q;
precondition  empty (q) == FALSE;
postcondition remove == first (q');
                  q == sub (q', 1, len (q') - 1);

abstract insere (q, elem)
QUEUE (t_elem) q;
t_elem elem;
postcondition q == <elem> + q';
```

Listagem 3.8: fila_int.h

```
/* Define uma fila de inteiros */

#define TAM_FILA 100

struct fila {
  int inicio, final;
  int itens[TAM_FILA];
};
```

Se desconsiderarmos, por enquanto, as condições de *underflow* ou *overflow*, em uma fila implementada desta forma, as funções `insere` e `remove` poderiam ser descritas, respectivamente, como:

```
struct fila q;
  q.itens[++q.final] = x;  /* insere */
  y = q.itens[q.inicio++]; /* remove */
```

No entanto, isto pode levar a um momento em que a fila poderá estar vazia e não se poder inserir um elemento na fila. Assim, o mais lógico é remanejar

todos os outros elementos da fila, tão logo o primeiro elemento seja removido. Ou seja:

```
struct fila q;
x = q.itens[0];
for(i=0; i<q.final; i++){
q.itens[i] = q.itens[i+1];
}
q.final--;
```

Entretanto, este procedimento tem um alto custo computacional, pois considerando-se uma fila muito grande, para cada operação efetuada todos os elementos da fila teriam que ser realocados.

3.4.3 Filas circulares

Uma solução é imaginar a fila como uma fila circular, ou seja, imagine uma fila vazia de 4 posições (0, 1, 2, 3), como na figura 3.3(a). Na figura 3.3(b) inserimos na fila os elementos 'A', 'B' e 'C'. Neste caso o início da fila está em '0' e o final está em '3' (a posição para inserir um novo elemento). Se, como na figura 3.3(c), removemos, 'A', que está no início da fila, o início da fila passa para '1' (que contém 'B') e o final continua em '3'. Podemos, então, inserir 'D' e 'E', como na figura 3.3(d).

O final da fila está em '1', bem como o início. Nesta situação, o elemento a ser removido na fila é 'B', mas nenhum elemento pode ser inserido antes disso, pois a fila está cheia e a operação deve retornar um *overflow*. É claro que ainda devemos tratar as condições de *underflow*.

Uma fila circular pode ser implementada com alguns cuidados, pois deve-se identificar a condição da fila, antes de se executar as operações sobre ela, evitando tanto o *underflow* como o *overflow*. As listagens 3.9 e 3.10 apresentam o cabeçalho `fila.h` e as funções para uma fila circular (`fila.c`).

Listagem 3.9: fila.h

```
/* descreve uma fila em C */

#define TRUE 1
#define FALSE 0
```

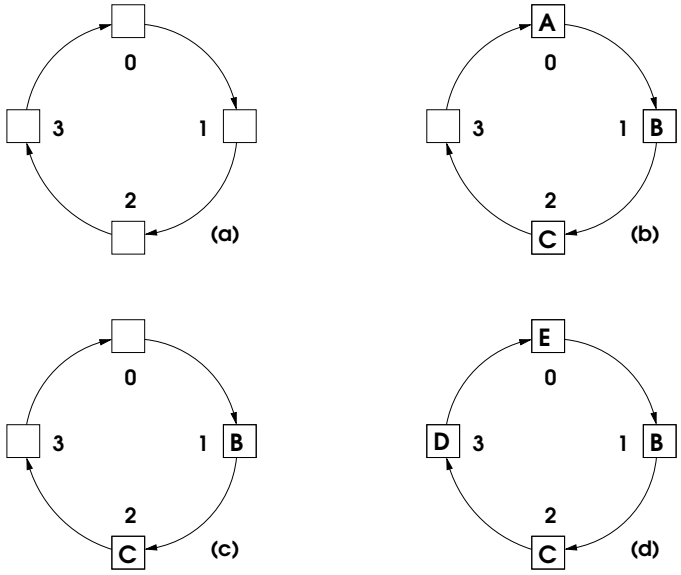


Figura 3.3: Exemplo de uma fila circular

```
#define TFILA 100   /* tamanho da fila */

struct fila {
  int itens[TFILA];
  int inicio, final;
};
```

3.4.4 Listas ligadas

Existem tipos especiais de listas que são denominadas de **listas ligadas**. Para que se possa compreender como funcionan as listas ligadas, vamos imaginar que tenhamos uma lista de n elementos. Nesta lista devemos inserir um elemento em uma posição $k < n$. O processo exigira deslocar todos os elementos das posições de k até n, para abrir espaço para o novo elemeto, resultando na lista de $n + 1$ elementos (veja figura 3.4).

Se, entretanto, tivermos em cada nó da lista, um ponteiro para o próximo nó, a inserção de um elemento, em qualquer posição desta lista, requer uma simples operação de mudar o ponteiro do nó anterior para o novo nó; e apontar o novo nó para o nó seguinte da lista (veja figura 3.5).

Listagem 3.10: fila.h

```
/* Funcoes para operacoes em filas */
#include <stdlib.h>
#include <stdio.h>
#include "fila.h"

typedef struct fila F;

int empty (F *pq){
  if (pq->inicio == pq->final){
    return TRUE;
  }
  else{
    return FALSE;
  }
}

float remover (F *pq){
  if (empty(pq)) {
    printf ("underflow na fila\n");
    exit (1);
  }
  if (pq->inicio == TFILA - 1){
    pq->inicio = 0;
  }
  else{
    (pq->inicio)++;
  }
  return (pq->itens[pq->inicio]);
}

void insere (F *pq, float x){
  /* espaco para novo elemento */
  if (pq->final == TFILA - 1){
    pq->final = 0;
  }
  else{
    (pq->final)++;
  }
  /* verifica se estourou a fila */
  if (pq->final == pq->inicio){
    printf ("overflow na fila\n");
    exit(1);
  }
  pq->itens[pq->final] = x;
  return;
}
```

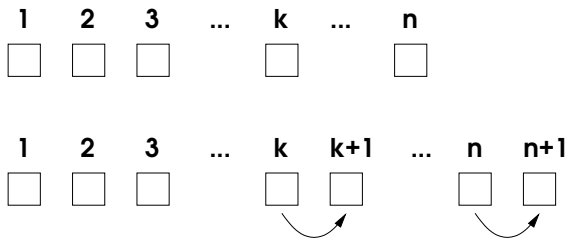


Figura 3.4: Exemplo de inserção de um elemento em uma lista não ligada.

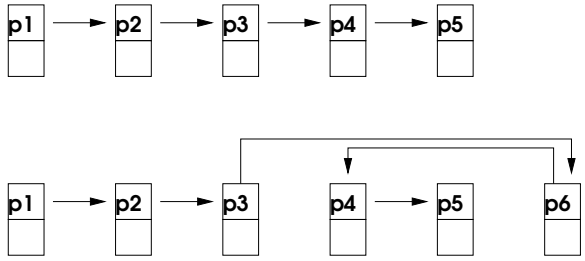


Figura 3.5: Exemplo de inserção de um elemento em uma lista ligada.

Uma estrutura para representar a lista ligada, pode ser definida como:

```
typedef struct LST_LIG {
  struct LST_LIG *next;
  } LST_LIG;
```

Além disso, é necessário que se defina apontadores para a "cabeça" e "cauda" da lista e, também o seu número de elementos:

```
typedef struct {
  int count;  /* numero atual de elementos na lista */
  LST_LIG *head; /* ponteiro para cabeca da lista */
  LST_LIG *z; /* ponteiro para o ultimo no da lista */
  LST_LIG hz[2]; /* vetor para os nos head e z */
  } LIST;
```

3.4.5 Operações com listas ligadas

Antes de implementar as listas como listas ligadas é preciso estabelecer quais as operações que podem ser implementadas. Para isso, deve-se definir algumas funções básicas. A primeira operação é a criação de uma lista ligada. Esta função não precisa receber parâmetros e deve retornar um ponteiro para a lista:

```
PUBLIC LIST *lst_init(void){
  LIST *l;

  if ((l = (LIST*)malloc(sizeof(LIST))) != NULL) {
    l->count = 0;
    l->head = &(l->hz[0]);
    l->z = &(l->hz[1]);
    l->head->next = l->z->next = l->z;
  }
  else {
    fprintf(stderr,"Memoria insuficiente para alocar a lista\n");
    raise(SIGABRT);
    return NULL;
  }

  return l;
}
```

Podemos implementar uma função `lst_newnode` para alocar a memória para um novo nó e retorna o um apontador para o nó alocado:

```
PUBLIC void *lst_newnode(int size){
  LST_LIG *node;

  if ( !(node = (LST_LIG*)malloc(size + sizeof(LST_LIG))) ){
    fprintf(stderr,"Memoria insuficiente para alocar o no.\n");
    raise(SIGABRT);
    return NULL;
  }
```

```
  /* OK... retorna o ponteiro */
  return LST_LOC(node);
}
```

Em algum momento, durante as operações, será necessário liberar um nó que não está em uso. Podemos criar uma função `lst_freenode` para executar esta operação:

```
PUBLIC void lst_freenode(void *node){
  free(LST_HEADER(node));
}
```

Para destruir uma lista, pode ser implementada uma função `lst_kill`:

```
PUBLIC void lst_kill(LIST *l,void (*freeNode)(void *node)) {
  LST_LIG *n,*p;

  n = l->head->next;
  while (n != l->z){  /* Libera todos os nos da lista */
    p = n;
    n = n->next;
    (*freeNode)(LST_LOC(p));
  }
  free(l);      /* Libera a propria lista */
}
```

A operação de inserir um nó depois de outro, pode ser feita por uma função `lst_insertafter`:

```
PUBLIC void lst_insertafter(LIST *l,void *node,void *after) {
  LST_LIG *n = LST_HEADER(node),*a = LST_HEADER(after);

  n->next = a->next;
  a->next = n;
```

```
  l->count++;
}
```

Para se eliminar um nó depois de outro. pode ser criada uma função `lst_deletenext`:

```
PUBLIC void *lst_deletenext(LIST *l,void *node) {
  LST_LIG *n = LST_HEADER(node);

  node = LST_LOC(n->next);
  n->next = n->next->next;
  l->count--;
  return node;
}
```

Finalmente, precisamos ter funções para obter o ponteiro para o primeiro elemento da lista e o ponteiro para o próximo nó:

```
PUBLIC void *lst_first(LIST *l){
  LST_LIG *n;

  n = l->head->next;
  return (n == l->z ? NULL : LST_LOC(n));
}

PUBLIC void *lst_next(void *prev) {
  LST_LIG *n = LST_HEADER(prev);

  n = n->next;
  return (n == n->next ? NULL : LST_LOC(n));
}
```

Com estas funções, podemos fazer todos os tipos de operações sobre as listas ligadas.

Listagem 3.11: potencia.c

```
#include <stdio.h>

double potencia (int, int);

int main (){
  int x = 2;
  long int d;

  d = potencia (x, 5);
  printf ("%f\n", d);

  return 0;
}

double potencia (int n, int p){
  long int resultado = n;
  while (--p > 0){
    resultado *= n;
  }
  return resultado;
}
```

3.5 Pilhas e Funções

Observe o código da listagem 3.11

O que ocorre, na realidade é que quando a função `main` é iniciada, cria-se uma pilha para armazenar 'x' e 'd'. Quando a função `potencia` é chamada, os valores '5' e '2' (valor de x), são colocados na pilha. A função pega os parâmetros 'n' e 'p' na pilha que são '5' e '2', respectivamente.

A função coloca a variável resultado no topo da pilha com o valor '2' e inicia um loop em que a variável é multiplicada por '2' a cada vez, até que 'p' seja zerada (veja figura 3.6).

A variável `resultado` contém, então, o resultado das multiplicações. O comando `return` termina a função e passando o valor de `resultado` para um

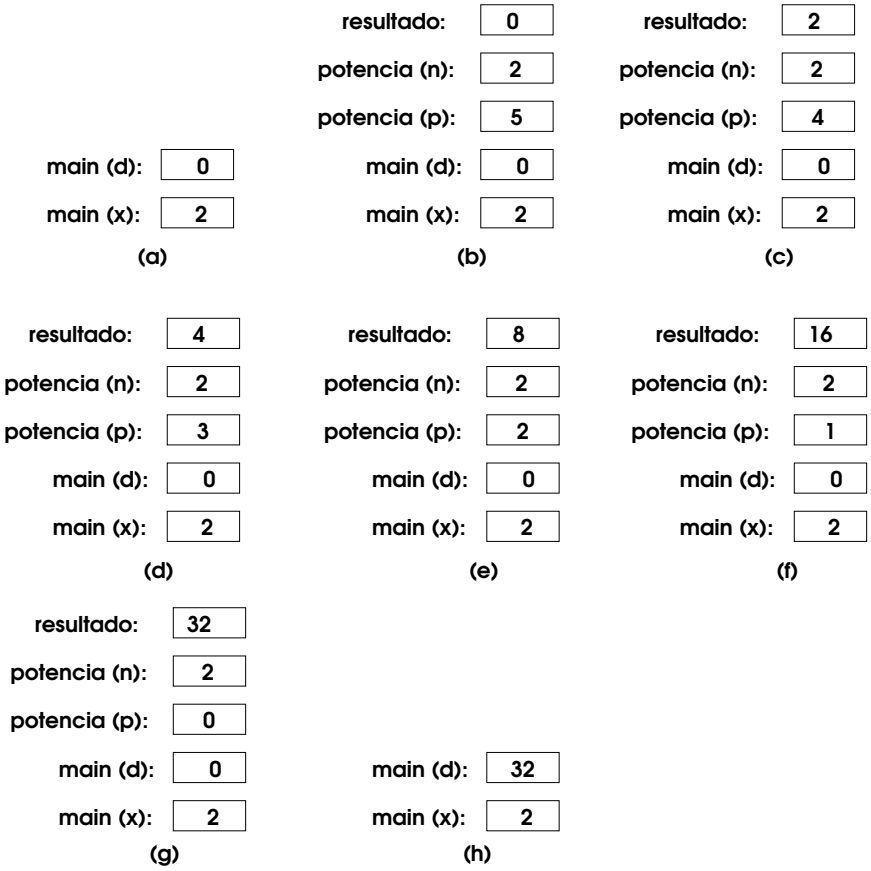


Figura 3.6: Pilha de chamada de uma função

registrador e destruindo (por meio do pop) as variáveis, `resultado`, `n`, `p` e `d`. Finalmente o valor de `resultado` no registrador é transferido para a pilha no espaço reservado para `d`, que é, então, processado (função printf).

O programa principal (`main`) termina com o código `return 0;`, o que coloca '0' no registrador e destrói suas variáveis 'x' e 'd', na pilha.

Na figura 3.6, (a) representa `pop x` e `pop d`; (b) representa `pop p`, `pop n` e `pop resultado`; de (c) a (g) se encontram o *loop* de multiplicação por '2' até $p = 0$; e, finalmente, (h) representa o `push resultado`, `push n`, `push p`, `push d` e `pop resultado` (agora em 'd').

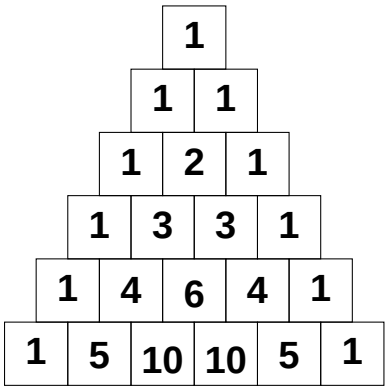


Figura 3.7: Triângulo de Pascal

3.6 Exercícios

1. Tendo compreendido o processo acima, implemente o mesmo algoritmo utilizando as funções push e pop, considerando os seguintes passos:

   - A chamada da função faz o PUSH dos parâmetros
   - A função é executada
   - A função pega os parâmetros na pilha (POP)
   - A função coloca seus parâmetros na pilha (PUSH)
   - A função termina e faz POP de seus parâmetros
   - O resultado da função é passado para onde foi chamado

2. O famoso Triângulo de Pascal (Figura 3.7) é uma estrutura infinita onde um número em uma linha é a soma dos dois números da linha imediatamente acima. A fórmula para se achar um número neste triângulo pode ser dada por:

$$f(n, r) = \frac{n!}{r! \cdot (n - r)!}$$

   Uma lista de listas das linhas deste triângulo, pode ser implementada como:

```
-- Triângulo de Pascal
```

```
module TPascal(
     pascal
  ) where

pascal :: Num a => Int -> [[a]]
pascal n = take n (iterate (\row -> zipWith (+)([0]++row)(row++[0])) [1])
```

a) Implemente um algoritmo para gerar esta lista em uma linguagem procedural.

b) Usando a fórmula com fatoriais, implemente um outro algoritmo.

Capítulo 4

Grafos e árvores

4.1 Grafos

Grafos são estruturas matemáticas usadas para representar idéias ou modelos, por intermédio de uma ilustração, gráfico ou esquema. Estritamente, em matemática, a teoria dos grafos é utilizada para representar o relacionamento entre dois ou mais conjuntos, grandezas ou valores. A representação de um mapa de rotas, redes e outros modelos semelhantes pode ser feita por meio do que se denomina grafos.

Vamos tomar, por exemplo, um mapa de rotas aéreas. Na figura 4.1, nota-se que existem várias rotas de um ponto a outro, por exemplo, de Florianópolis a Brasília existem as possibilidades via Curitiba, via São Paulo ou via Rio de Janeiro. Algumas rotas podem ter mais escalas, por exemplo, pode-se ir a Brasília por São Paulo, passando por Curitiba, Rio e Belo Horizonte. Neste exemplo, temos a linha que liga cada dois pontos. Note, entretanto, que existe uma rota de Florianópolis a Florianópolis que, digamos, pode representar um vôo panorâmico pela ilha que decola e retorna ao mesmo ponto.

O mapa de rotas contém pontos, que representam as cidades, e estão ligados por linhas, que representam as rotas aéreas.

Cada um dos pontos de um grafo é denominado nó e cada uma das ligações são denominadas arcos. A representação de dados como estrutura de dados, pode ser aplicadas em:

- Mapas de distâncias
- Mapas Metabólicos
- Diagramas e Fluxogramas

Figura 4.1: Representação de rotas aéreas

- Redes de computadores
- Redes Neurais
- Estruturas químicas

4.1.1 Definições

Grafos são definidos, matematicamente, como triplas ordenadas (n, a, g), onde n é um conjunto não-vazio de nós ou vértices, a é um conjunto de arcos ou arestas e g é uma função que associa cada arco a a um par não ordenado x–y de n, denominados extremidades de a.

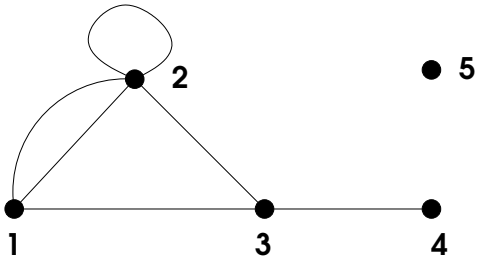


Figura 4.2: Exemplo de um grafo, seus arcos e nós.

A figura 4.2, representa um grafo com 5 nós (vértices) e 6 arcos (arestas), onde a função g de cada arco pode ser descrita por:

$$\begin{aligned} g(a1) &= 1-2 \\ g(a2) &= 1-2 \\ g(a3) &= 2-2 \\ g(a4) &= 2-3 \\ g(a5) &= 1-3 \\ g(a6) &= 3-4 \end{aligned}$$

Como o nó 5 não está ligado por nenhum arco, não temos uma função que o utilize.

Os arcos de um grafo podem ser simples ou direcionados, o que significa que a relação entre os nós têm um sentido. Neste caso, o grafo recebe o nome de **dígrafo**.

Dígrafos (ou grafos direcionados) são triplas ordenadas (n, a, g), onde n é um conjunto não-vazio de nós, a é um conjunto de arcos ou vértices e g é uma função que associa cada arco a a um par ordenado (x, y) de n, onde x é o ponto inicial e y é o ponto final de a.

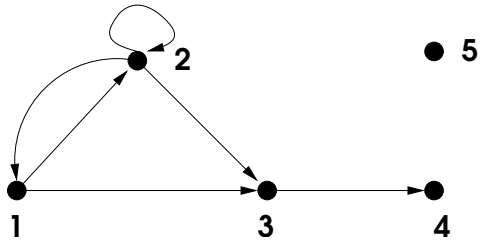


Figura 4.3: Representação de grafo direcionado

A figura 4.3 representa um grafo direcionado (dígrafo) com 5 nós e 6 arcos, que pode ser descrito pelas funções:

$$\begin{aligned} g(a1) &= 1,2 \\ g(a2) &= 2,1 \\ g(a3) &= 2,2 \\ g(a4) &= 2,3 \\ g(a5) &= 1,3 \\ g(a6) &= 3,4 \end{aligned}$$

4.1.2 Terminologia

A terminologia utilizada na teoria dos grafos compreende as definições dos elementos, características e propriedades dos grafos. Os elementos de um grafo

compreendem:

- **Laços**: São arcos que conectam apenas um nó.
- **Nós Adjacentes**: São dois nós ligados por um arco.
- **Arcos Paralelos**: Dois arcos com as mesmas extremidades, ou seja, conectam os mesmos nós.
- **Nó isolado**: Um nó que não é adjacente a nenhum outro nó.
- **Grau de um Nó**: Número de arcos que conectam um nó.

Por exemplo, na figura 4.4, o nó 2 tem um laço; o nó 5 é adjacente aos nós 2,3 e 4, equanto o nó 2 é adjacente ao 1, 5 e ele mesmo; os arcos e e f são paralelos; o nó 6 é um nó isolado; e o nó 6 tem grau zero, o nó quatro tem grau 2, os nós 1, 2 e 5 têm grau 3 e o nó 3 tem grau 4.

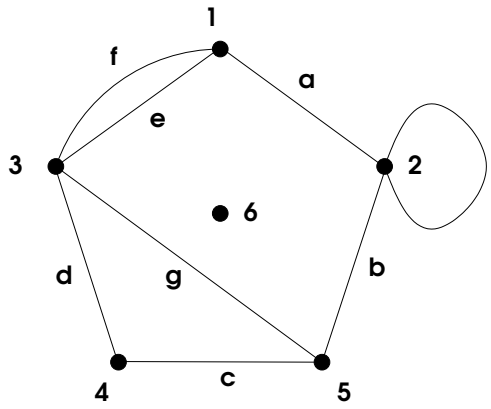


Figura 4.4: Elementos de um grafo

As características de um grafo podem ser descritas como:

- **Grafos Simples**: São grafos sem laços ou arcos paralelos.
- **Grafos Completos**: São grafos onde quaisquer dois nós distintos são adjacentes.
- **Grafos Conexos**: São grafos onde sempre existe um caminho de um nó a outro, por meio de um ou mais arcos.

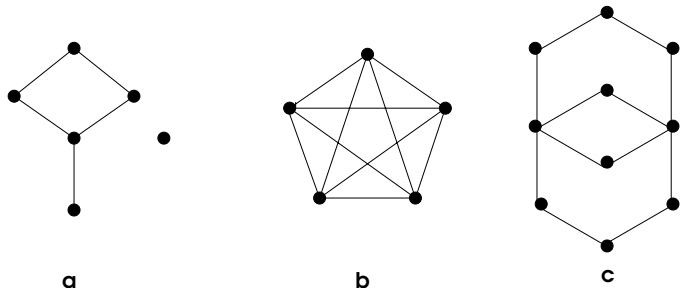


Figura 4.5: Características dos grafos: (a) Simples, (b) completo e (c) Conexo.

A figura 4.5 exemplifica as características de um grafo.

Quanto às propriedades, os grafos podem ser **isomórficos** e/ou **planares**.

Um grafo é dito isomorfos quando pode ser representado de duas ou mais maneiras, sendo que têm os mesmos nós, os mesmos arcos e a mesma função que associa cada extremidade dos arcos, ou seja, Dizemos que dois grafos $(N1, A1, G1)$ e $(N2, A2, G2)$ são isomorfos se, e somente se $\exists f1 : N1 \rightarrow N2$ e $f2 : A1 \rightarrow A2$ (veja figura 4.6).

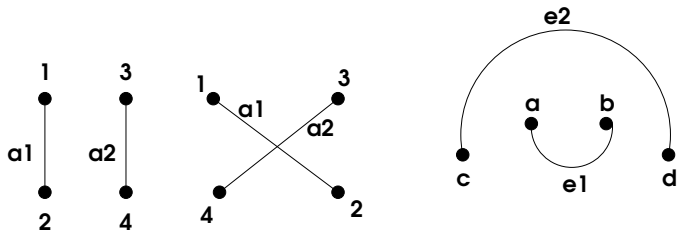


Figura 4.6: Grafos isomorfos

Na figura 4.6, as funções de nós e arcos podem ser escritas como:

$$\begin{aligned} f_1 &: 1 \rightarrow a; 2 \rightarrow b; 3 \rightarrow c; 4 \rightarrow d \\ f_2 &: a_1 \rightarrow e_1; a_2 \rightarrow e_2 \end{aligned}$$

A planaridade é a propriedade de um grafo poder ser representado em um plano, de modo que seus arcos não se intersectam entre si. A figura 4.7 exemplifica grafos planares.

Euler observou que a o número de nós n, arcos a e regiões r delimitadas pelos arcos de um grafo planar é dado pela equação $(n - a) + r = 2$, que é conhecida como **fórmula de Euler**.

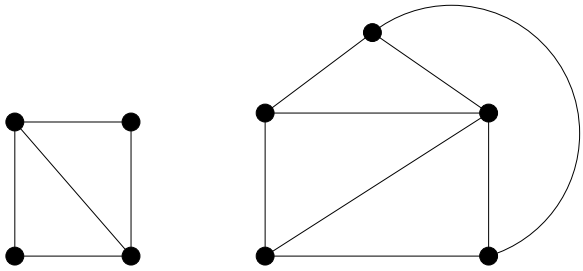


Figura 4.7: Grafos planares

4.2 Grafos e Estrutura de dados

Para a representação de dados de um conjunto que pode ser modelado como um grafo, pode-se usar uma representação matricial (array) ou uma lista ligada. Estas representações levam em conta as relações entre nós adjacentes e, por isso, são denominadas de **matriz adjacência** ou **lista de adjacência** de um grafo.

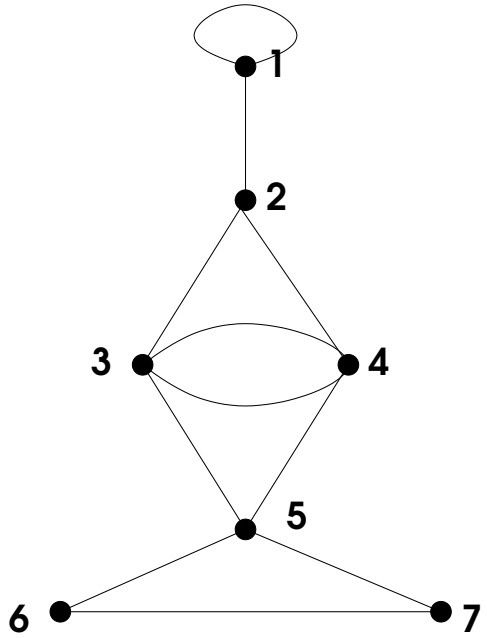


Figura 4.8: Exemplo

Considere, por exemplo, o grafo da figura 4.8. Podemos representá-lo como uma matriz, onde cada elemento representa a conexão entre o nó_{linha}

e o nó$_{coluna}$, ou seja,

$$a_{i,j} \rightarrow G(a_k) = (i - j)$$

e pode ser representado pela matriz:

$$G = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 2 & 1 & 0 & 0 \\ 0 & 1 & 2 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

Observe que cada elemento da matriz representa o número de conexões entre o nó representado pelo número da linha e o nó representado pelo número da coluna. Por exemplo: $a_{1,1}$ representa o laço do nó 1; $a_{2,3}$ representa o arco entre o nó 2 e o nó 3; e $a_{4,3}$ representa as duas conexões entre o nó 4 e o nó 3. Observe ainda que a matriz que representa um grafo é simétrica, pois $a_{i,j} = a_{j,i}$.

4.2.1 Grafos direcionados e matriz adjacência

A representação de grafos direcionados (dígrafos) em uma matriz adjacência, quase nunca gera uma matriz simétrica. Na maioria das vezes, a matriz é uma matriz boolena esparsa. Por exemplo, o grafo da figura 4.9 pode ser representado pela matriz:

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Neste caso, podemos observar que a matriz não é simétrica, e os arcos $(6, 7)$ e $(7, 6)$ não são paralelos, apesar de suas posições simétricas na matriz adjacência.

Grafos como este podem ser representados, ainda, por relações binárias, ou seja, se N é um conjunto de nós de um grafo direcionado, então (n_i, n_j) é um par ordenado, tal que:

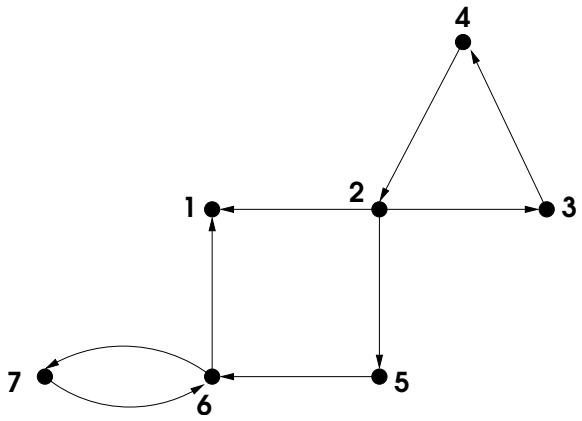


Figura 4.9: Exemplo de grafo direcionado

$$\exists(n_i, n_j) \iff \exists g(a_{i,j}) = n_i \sim n_j.$$

No caso do grafo acima, podemos representá-lo como um conjunto de pares ordenados:

$$N = (2,1), (2,3), (2,5), (3,4), (4,2), (5,6), (6,1), (6,7), (7,6)$$

Note que somente existe um conjunto de pares ordenados N para um grafo G se este não tiver arcos paralelos. E, como uma relação binária em um conjunto pode ter as propriedades de **reflexibilidade**, **simetria**, **antisimetria** e **transitividade**, qualquer destas propriedades que o conjunto N apresentar, refletirá em G e na matriz booleana.

4.2.2 Acessibilidade dos nós de um grafo

Dizemos que um nó n_j de um grafo pode ser acessado a partir de um nó n_i se, e somente se, existir um caminho de n_i para n_j. O conceito de acessibilidade de um nó é de grande importância no estudo e análise de redes, principalmente de redes de computadores, de telefonia, de energia, água etc..

Considere o grafo da figura 4.10.

Sua matriz adjacência,

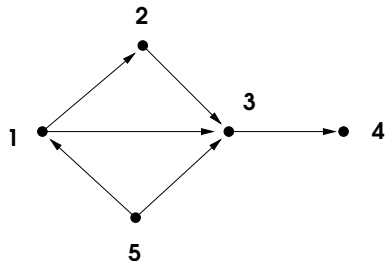


Figura 4.10: Acessibilidade dos nós

$$A = \begin{pmatrix} 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 \end{pmatrix}$$

não dá muita informação, pelo menos de maneira clara, sobre a acessibilidade de cada nó, uma vez que ela explicita um arco (o caminho de um nó para o adjacente).

Entretanto, se tomarmos a matriz adjacência e calcularmos o produto booleano dela com ela mesmo, teremos o nó seguinte (o caminho após dois arcos). Esta é uma matriz de acesso de um nó a outro, por meio de dois arcos:

$$A^{(2)} = A \cdot A = \bigvee_{k=1}^{n} (a_{i,k} \wedge a_{k,j})$$

Note que $A \cdot A$, aqui representada por $A^{(2)}$, é um produto booleano e não o quadrado da matriz A (A^2). A matriz $A^{(2)}$ é, portanto:

$$A^{(2)} = \begin{pmatrix} 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

Os p caminhos seguintes, onde p é o máximo de caminhos possíveis, podem ser encontrados a partir de:

$$A^{(p)}[i,j] = A^{(p-1)} \cdot A = \bigvee_{k=1}^{n} (A^{(p-1)}[i,k] \wedge a_{k,j})$$

$$A^{(3)} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

$$A^{(4)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

A multiplicação lógica das matrizes obtidas gera uma matriz que indica o alcance de um nó por $1, 2, \cdots, p$ arcos:

$$M = A \vee A^{(2)} \vee \cdots \vee A(p)$$

$$M = \begin{matrix} 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 \end{matrix}$$

Finalmente, a matriz acessibilidade B é obtida pela transposição de M, ou seja:

$$B = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

4.3 Caminho mínimo

Os grafos podem representar ligação entre dados, onde os arcos apresentam "pesos" ou valores que especificam alguma característica da conexão. Em muitos casos, a solução de um problema depende de se encontrar o caminho de menor "custo" ou de menor valor ou, ainda, de maior eficiência. Este caminho recebe o nome de **caminho mínimo** entre dois nós de um grafo.

4.3.1 Um exemplo

Vamos considerar um problema de cristalografia, onde a orientação do cristal é importante para se descobrir as arestas de corte (veja figura 4.11).

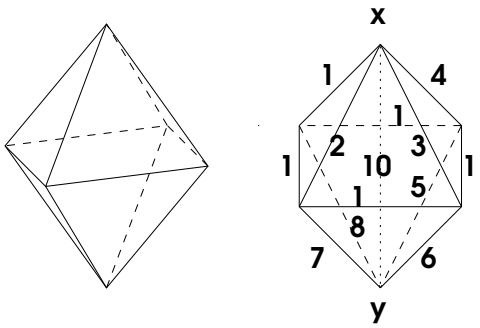


Figura 4.11: Representação de uma malha cristalina

Neste cristal hipotético, as arestas de corte são representadas pelas arestas de menor malha cristalina entre x e y. Assim, deve-se encontrar, entre x e y, o caminho mínimo pelas arestas de menor peso. A solução do problema está em determinar os caminhos, somar seus "pesos"e determinar qual é o menor valor (ou os menores valores).

4.4 Algoritmos para grafos

Quando os grafos têm poucos nós, todas as operações e cálculos que envolvam matrizes (adjacência ou acessibilidade) podem, facilmente, ser efetuados manualmente. No entanto em situações onde o número de nós é maior, a utilização de um computador pode auxiliar no cálculo destas estruturas. Assim, vamos verificar como se pode implementar tais algoritmos. Para isso, vamos considerar a descrição do algoritmo que pode ser implementado em qualquer linguagem.

4.4.1 Algoritmo de Warshall

O primeiro algoritmo nos permite achar a matriz acessibilidade de um grafo. Para isso, vamos considerar um grafo G com n nós, para o qual se necessita de $n+1$ matrizes para cada nível de acessibilidade. Tal algoritmo é conhecido como **algoritmo de Warshall** (veja listagem 4.1).

Listagem 4.1: Algoritmo de Warshall

```
// ALGORITMO de Warshall

warshall( Matriz(n x n) )

// Matriz inicial é a matriz adjacência de
// um dígrafo sem arcos paralelos.

para k=0 até (n - 1); faz
  para i=1 até n; faz
    para j=1 até n; faz
      M[i][j] = M[i][j] || ( M[i][k+1] && M[k+1][j])
    fim
  fim
fim
retorna Matriz  // no final a matriz resultante
                // é a matriz acessibilidade
```

O algoritmo de Warshall começa com a matriz adjacência (M_0) e calcula as outras matrizes M_1, M_2, ... , M_n. Note que é suficiente calcular até o número de nós, pois qualquer nó pode ser acessível pelo número de passos igual ao número de nós, não interessando as outras repetições.

4.4.2 Caminho de Euler

O problema do caminho de Euler que levou o nome do matemático suíço Leonard Euler, também é conhecido como "problema das sete pontes" , ele é baseado em uma cidade dividida por um rio e com uma porção insular, ligadas por sete pontes, em sua época denominada de Königsberg, atualmente Kaliningrad (veja figura 4.12).

O problema consistia em determinar se era possível fazer um tour pela cidade passando em cada uma das pontes apenas uma vez. Euler solucionou o problema, matematicamente, representando as pontes de Kaliningrado como arcos de um grafo como o da figura 4.13

O algoritmo do caminho de Euler serve para determinar se existe um caminho de Euler em um grafo conexo qualquer. O algoritmo conta o número de nós adjacentes a cada nó para determinar se este número é par ou ímpar.

Listagem 4.2: Algotitmo para o caminho de Euler

```
// ALGORITMO do Caminho de Euler: Determina se existe
// um caminho de Euler em um grafo conexo
// a partir da matriz adjacência

cam_euler( Matriz(n x n) )

// Variaveis
  int Imp   // número de nós de nível impar encontrados
  int Grau  // Grau de um nó
  int i, j

  Imp = 0
  i = 1
  enquanto Imp <=2 && i = n; faz
    Grau = 0

    para j = 1 ate n; faz
      Grau = Grau + M[i][j]
    fim

    se impar(Grau); entao
      Imp = Imp + 1
    fim

  fim
  se Imp > 2; entao
    print "Nao existe um caminho de Euler"
  senao
    print "Existe um caminho de Euler"
  fim
fim
```

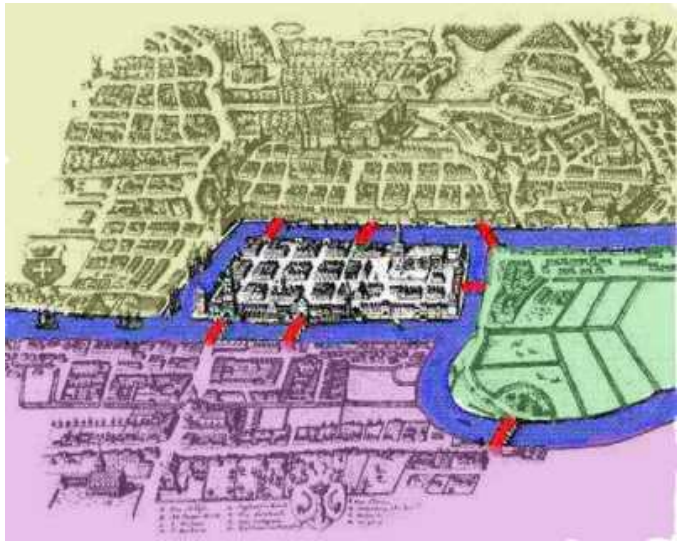


Figura 4.12: Kaliningrad: o problema das sete pontes.

Teorema: Existe um caminho de Euler em um grafo conexo se, e somente se, não existirem nós de grau ímpar ou existirem exatamente dois nós ímpares.

O algoritmo do caminho de Euler é semelhante ao chamado "circuíto de Hamilton", cujo problema é determinar se existe, em um grafo, um ciclo contendo todos os nós.

4.4.3 Algoritmo do Caminho Mínimo

Suponha um mapa de cidades que sejam todas interligadas e que possa ser representado por um grafo simples conexo. Neste caso existe um caminho entre quaisquer cidades x e y e, considerando as distâncias entre elas, percorrida por vários caminhos, existirá um ou mais deles que será o mais curto. O problema de identificar o caminho mais curto é denominado " caminho mínimo" de um grafo.

A solução de tal problema é bastante importante para redes de computadores ou de comunicação. O algoritmo para resolver este problema foi estabelecido por Dijkstra e, por isso, ficou conhecido como **algoritmo de Dijkstra** e é válido para qualquer grafo simples conexo com pesos positivos em seus arcos.

```
// Algoritmo do Caminho Mínimo (Dijkstra)

dijkstra( A (n x n), N, x, y )

// A eh a matriz adjacência modificada de um grafo simples conexo.
// x e y são os nós para os quais se deseja calcular o caminho mínimo.
// N: os nós do grafo

// Variaveis:
  array Min_nos  // conjunto de nós cujo caminho mínimo é determinado
  nos temp1, temp2  // nós temporários
  int dist[]  // distancia de x para cada nó em Min_nos
  int no[]    // distancia para cada nó anterior no caminho mínimo
  int d_ant   // distância anterior para comparar

  // Valores iniciais de Min_nos, dist[] e no[]
  Min_nos = {x}
  dist[x] = 0
  para cada temp1 em Min_nos; faz
    dist[temp1] = A[x][temp1]
    no[temp1]   = x
  fim
  // Coloca nós em Min_nos
  enquanto y nao em Min_nos
  // acrescenta o nó de distancia mínima em Min_nos
    temp2 = no[temp1] nao em Min_nos com dist[temp1] min
    Min_nos = Min_nos ++ {temp2}
  // calcula novo dist[temp1] em Min_nos, acertando no[]
    para cada N nao em Min_nos; faz
      d_ant       = dist[temp1]
      dist[temp1] = min ( dist[temp1],
                    dist[temp2] + A[temp2][temp1] )
      se dist[temp1] != d_ant; entao
        no[temp1] = temp2
      fim
    fim
  fim

  // grava os nós do caminho
  print "Nós em ordem inversa:" , print y
  temp1 = no[temp1]
  faz
    print no[temp1]
    temp1 = no[temp1]
  ate temp1 = x

  // escreve a distancia minima
  print "caminho minimo: ", print dist[y]
fim
```

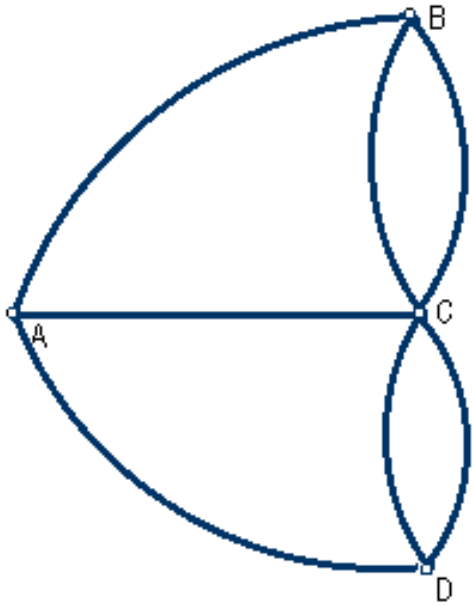


Figura 4.13: Representação do problema de Euler

4.4.4 Problema do Caixeiro Viajante

Existem várias versões deste problema, mas a forma mais conhecida é a cada passo escolher a cidade vizinha mais próxima. Com as seguintes regras:

1. Começar numa cidade qualquer;

2. Dessa cidade, ir para cidade mais próxima que ainda não foi visitada;

3. Continuar dessa forma, até todas as cidades terem sido visitadas; da última, regressar à cidade de onde se saiu.

Suponha que representemos soluções deste problema como uma lista L em que L_k é a cidade a visitar em i-ésimo lugar, a função deve devolver a cidade que ainda não foi visitada mais próxima do último elemento da lista.

Uma outra estratégia para a solução deste problema consiste em, a cada passo, adicionar o arco de menor custo de forma a que não se formem ciclos nem nodos com grau maior que 2. O último arco a ser incluído fecha o ciclo.

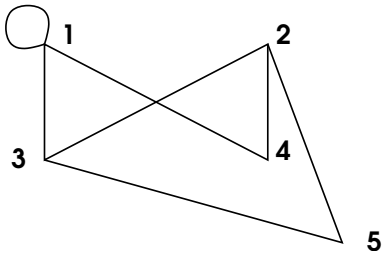


Figura 4.14: Modelo de grafo para lista de adjacências

4.5 Listas adjacência

Uma das principais formas de aplicação do conceito de grafos é na construção de modelos para a representação de informações. No entanto, para que se possa criar estes modelos e representar, armazenar e manipular as informações, é necessário que se use um tipo de estrutura de dados.

Como foi visto na teoria, Existem, basicamente, duas maneiras de representar os grafos. Uma delas é uma matriz, denominada **matriz adjacência**, e a outra é uma lista, denominada **lista de adjacências**.

4.5.1 Matriz Adjacência

Uma matriz adjacência é uma matriz simétrica booleana,

$$M = [x_{1,1}...x_{i,j}...x_{m,n}],$$

onde o elemento $x_{i,j}$ corresponde à função que descreve a ligação de dois nós, onde $g(a_k) = n_i - n_j$.

4.5.2 Lista de Adjacências

A construção de modelos de representação de dados, nem sempre é uma estrutura completa ou conexa. No processamento da informação, ao tratarmos dados modelados em grafos, é muito comum obtermos como resultado uma matriz esparsa (com muitos zeros). Em uma estrutura como esta, a localização de um elemento, no conjunto de n dados, são necessários n^2 dados para representar a matriz adjacência.

Assim, a representação de um grafo com poucos arcos, pode ser feita, mais facilmente, por uma **lista encadeada** ou **lista de adjacências**. Por exemplo, no caso do grafo da figura 4.14, podemos representá-lo em uma lista que contém

o ponteiro para cada nó, o próximo nó ligado a ele e um apontador para o próximo nó. Assim, sua representação seria:

$$
\begin{array}{lllllllll}
1 & \rightarrow & 1 & \rightarrow & 3 & \rightarrow & 4 & \cdot \\
2 & \rightarrow & 3 & \rightarrow & 4 & \cdot & & \\
3 & \rightarrow & 1 & \rightarrow & 2 & \rightarrow & 5 & \cdot \\
4 & \rightarrow & 1 & \rightarrow & 2 & \cdot & & \\
5 & \rightarrow & 2 & \rightarrow & 3 & \cdot & &
\end{array}
$$

4.6 Exercícios

1. Escreva um programa que receba, como entrada, uma matriz adjacência de um grafo e produza, como saída, uma lista de adjacências.

2. Escreva um programa que faça o inverso.

3. Faça uma pesquisa para descobrir o que é o caminho de Euler e escreva um algoritmo que determine a existência ou não de um caminho de Euler em um grafo conexo.

4.7 Árvores

Podemos definir uma **árvore** como um grafo conexo e não cíclico, com um nó denominado "raíz", em um nível zero, de onde partem os arcos (caminhos) para outros nós, denominados "folhas" (veja figura 4.15). Cada folha pode ter os seus descendentes, sendo que os nós do nível imediatamente abaixo são denominados "filhos" e os de nível imediatamente acima são denominados "pais".

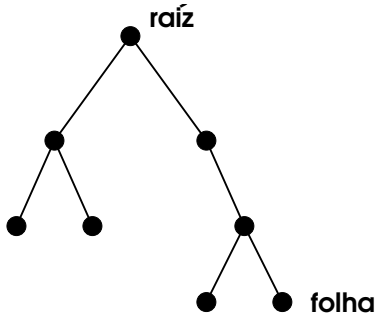


Figura 4.15: Exemplo de uma árvore

De particular interesse para estrutura de dados, são as árvores binárias, em que cada nó tem, no máximo, dois filhos no nível seguinte. O nó esquerdo inferior (NEI) e o nó direito inferior (NDI) são denominados "filho esquerdo" e "filho direito", respectivamente.

As árvores binárias podem ser completas, quando todos os nós do nível $d-1$ de uma árvore binária com d níveis tiverem filhos esquerdos e direitos, ou quase completa quando não existirem nós com apenas um dos filhos[1] (veja exemplos na figura 4.16).

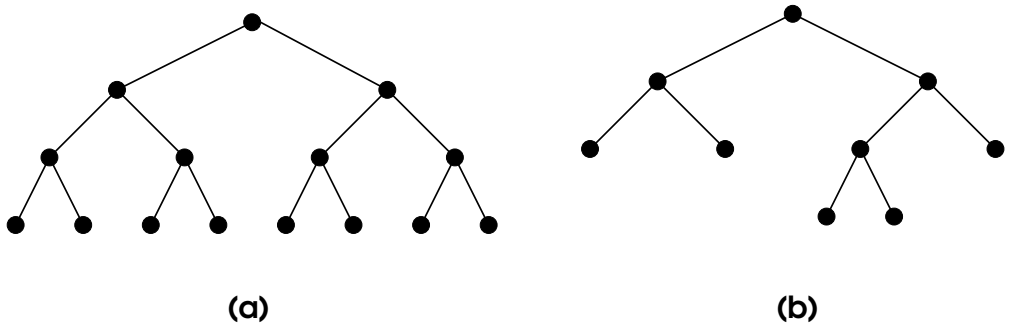


Figura 4.16: Exemplo de árvores (a) completa e (b) quase completa.

Existem ainda árvores **estritamente binárias**, que são árvores cujos nós, quando não-terminais (folhas), têm ambos os filhos (veja figura 4.17).

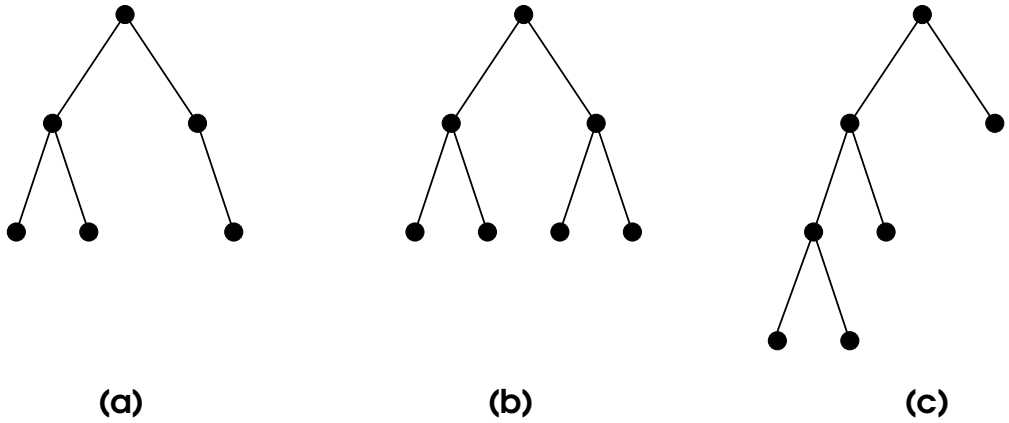


Figura 4.17: Exemplo de árvores (a) binária, (b) e (c) estritamente binárias.

[1] Alguns autores chamam de árvore cheia a árvore completa e de árvore completa o que definimos aqui como árvore quase completa.

Uma árvore binária com m nós no nível d, terá, no nível $d+1$, no máximo $2m$ nós, ou seja, o dôbro de nós do nível anterior. Assim, o máximo número de nós de um nível d em uma árvore binária será de 2^d e o total (T) de nós de uma árvore binária pode ser expresso por:

$$T = \sum)k = 0^d 2^k$$

onde d é o número de níveis.

4.7.1 Percurso

Existem 3 maneiras diferentes de se percorrer uma árvore binária. Estes percursos são fundamentais para que se possa utilizar a estrutura de árvores na computação de dados. Assim, é necessário que se estabeleça algoritmos de percurso para escrita e recuperação de elementos em posições representadas pelos nós de uma árvore. Para isso, são usados 3 algoritmos:

- Percurso em Pré-ordem;
- Percurso em Ordem Simétrica; e
- Percurso em Pós-ordem.

Esta nomenclatura se refere à ordem em que a raíz da árvore e das sub-árvores são percorridas.

O percurso em **pré-ordem** começa pela raíz da árvore, descendo primeiro pelas raízes das sub-árvores mais à esquerda e depois as da direita. A figura 4.18 representa uma árvore binária e o percurso em pré-ordem.

O percurso em Pré-ordem visita primeiro a raíz a e depois vai para a raíz da sub-árvore à esquerda (b), seguindo sempre à esquerda para o nó d. Como d não é raíz de sub-árvore, visita a próxima raíz à direita (e), seguindo sempre nesta ordem. Resultando em:

$$a \rightarrow b \rightarrow d \rightarrow e \rightarrow h \rightarrow i \rightarrow c \rightarrow f \rightarrow g \rightarrow j \rightarrow k$$

O percurso em **Ordem simétrica** visita a raíz de cada sub-árvore à esquerda, antes da direita, separando os nós e sub-árvores de forma simétrica em relação às suas raízes (veja figura 4.19).

Assim, o percurso para a árvore começa em d passando por b e depois a sub-árvore com raíz em e, e assim por diante, resultando em:

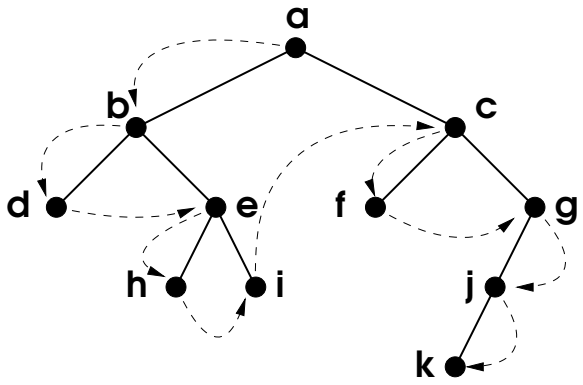


Figura 4.18: Exemplo do percurso, em pré-ordem, de uma árvore binária.

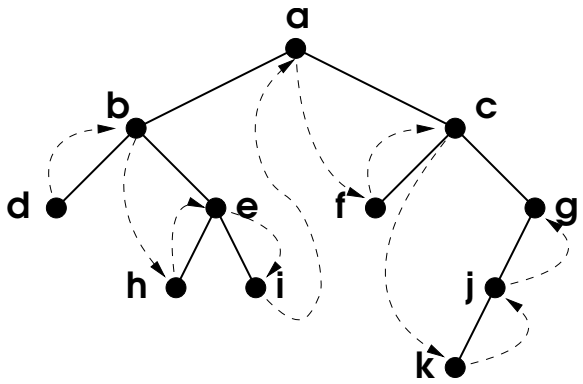


Figura 4.19: Exemplo do percurso, em ordem simétrica, de uma árvore binária.

$$d \to b \to h \to e \to i \to a \to f \to c \to k \to j \to g$$

Finalmente, o percurso em **Pós-ordem** visita a sub-árvore à esquerda das folhas para a raíz, deixando a raíz da árvore por último (conforme figura 4.20), resultando no percurso:

$$d \to h \to i \to e \to b \to f \to k \to j \to g \to c \to a$$

4.7.2 Um exemplo

As árvores podem ser utilizadas para representar operações algébricas. Por exemplo, a expressão $(3+x)-(5/2)$ pode ser representada por por uma árvore binária como a da figura 4.21.

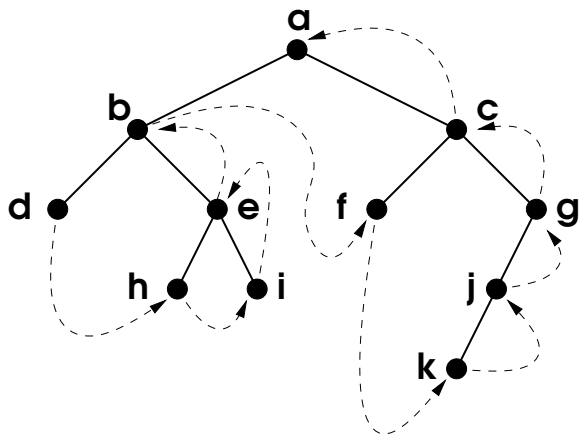


Figura 4.20: Exemplo do percurso, em pós-ordem, de uma árvore binária.

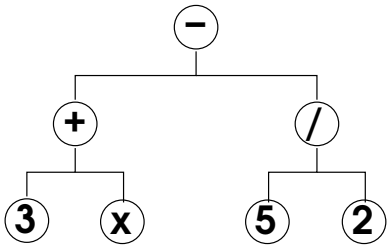


Figura 4.21: Exemplo de representação da equação algébrica $(3 + x) - (5/2)$ por uma árvore binária.

É evidente que a notação infixa[2] deriva do percurso de ordem simétrica da árvore. No caso do percurso por pré-ordem, o resultado seria a notação prefixa (notação polonesa) da expressão (*e.g.* $-+3x/52$); e o percurso em pós-ordem, resultaria na notação posfixa (polonesa reversa) da expressão (*e.g.* $3x+52/-$).

Como as expressões polonesas podem representar expressões de forma mais eficiente (pois dispensa o uso de parênteses), os compiladores mudam a notação infixa dos programas para a notação polonesa reversa.

4.8 Árvore Geradora Mínima

Seja $G = [V, N]$ um grafo conexo não direcionado e com pesos em seus arcos, onde V representa o conjunto de vértices, que possuem um peso positivo, e N representa o conjunto de nós, existe um subconjunto T de tal que:

[2] O correto seria chamá-la de "forma entrefixa".

$$T \in N$$

e T conecta todos os vértices de V, tal que a soma de seus pesos seja mínimo. O grafo $G' = [V, T]$ é denominado **Árvore Geradora Mínima** (AGM) de G.

O conceito de AGM é utilizado em uma série de tipos de redes, tais como rotas aéreas, metrôs, redes de comunicação etc.. Uma AGM representa, portanto, o menor custo de uma conexão de vários pontos.

Basicamente, existem dois algoritmos para a implementação de uma AGM: o **algoritmo de Kruskal** e o **algoritmo de Prim**.

4.8.1 Algoritmo de Kruskal

Inventado em 1956, o algoritmo de Kruskal determina a AGM de um grafo da seguinte maneira:

1. Cria um conjunto vazio T;
2. Cria um conjunto de caminhos C que contém, cada um dos V vétices;
3. Examina os nós N em ordem crescente de peso;
4. Se um nó tem liga dois componentes em C adiciona a T e os adiciona a C. Caso contrário rejeita;
5. Repete os dois passos anteriores até que C contenha apenas um componente.

4.8.2 Algoritmo de Prim

O algoritmo de Prim apareceu em 1957, apesar de já ter sido proposto eem 1930 por Jarnik (o que Prim desconhecia), ficou conhecido como **algoritmo de Prim** e é implementado da seguinte maneira:

1. Cria um conjunto vazio de arcos T;
2. Cria um conjunto vazio de nós C;
3. Seleciona um nó arbitrário V e acrescenta a C;
4. Escolhe o arco $a(m, n)$ de menor peso, tal que m esteja em C e n não;
5. Acrescenta n a C e a T;

6. Repete os dois passos anteriores até que C seja igual a N, ou seja, contém todos os nós.

4.9 Exercícios

1. Implemente os dois algoritmos e teste-os, usando o grafo da figura 4.22.

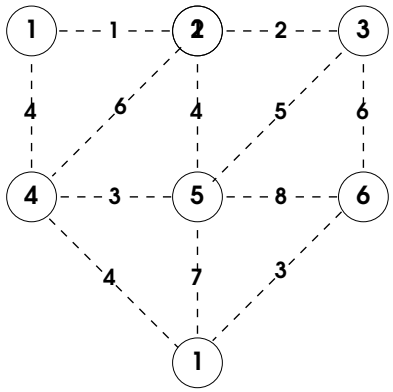


Figura 4.22: Grafo para achar a AGM

Dica: O grafo pode ser implementado por meio de uma matriz adjacência ou uma lista de adjacência.

Capítulo 5

Máquinas de Estado

Máquinas de estado são modelos estruturais que representam o estado de algum símbolo ou conjunto de símbolos em um determinado tempo, que podem receber uma entrada (*input*) para uma mudança, que vai alterar o estado atual para um novo estado. Um computador, por exemplo é uma máquina de estado[1], mas podemos usar o modelo de máquina de estado para desenvolver, descrever e/ou modelar interações de sistemas e programas.

Podemos descrever uma máquina de estado como sendo:

- Um estado inicial descrito;
- Um conjunto de eventos de entrada;
- Um conjunto de eventos de saída;
- Um conjunto de estados;

e que utiliza funções que mapeiam:

- Estados e entradas para novos estados; e
- Estados para saídas.

[1] A arquitetura von Neumann é um modelo determinístico de uma seqüência de operações, onde para um mesmo conjutno de dados entrada sempre há um conjunto constante e imutável de dados de saída.

5.1 Máquinas de estado finito

Máquina de estado finito é aquela que tem um limite ou número finito de estados. Ela pode ser definida por um estado inicial s_0; um conjunto finito de estados S; um conjunto finito de eventos I de entradas e O de saídas e duas funções:

$$\begin{aligned} f_s &: S \to I \to S \\ f_o &: S \to O \end{aligned}$$

onde, f_s é uma função de mudança de estado e f_o é uma função de saída.

Por exemplo, considere um estado $s_n(t)$ em um tempo t, qualquer, então

$$s_{n+1}(t+1) = f_s(s_n(t), I_n(t))$$

ou seja, o estado do tempo $t+1$ é obtido pela aplicação da função de estado f_s ao par (s, I).

5.1.1 Um exemplo

Para compreender como operam as máquinas de estado vamos observar um exemplo. Para isso vamos considerar uma máquina de estado finito $S = s_0, s_1, s_2, s_3$ com entradas $I = 0, 1$ e saídas $O = 0, 1$, cujo sistema é descrito pelas funções de estado:

$$\begin{aligned} f_s &: (S_n, 0) \to S_{n+1} \\ f_s &: (S_n, 1) \to S_n \end{aligned}$$

e funções de saída descritas por:

$$\begin{aligned} f_o &: s_0 \to 0 \\ f_o &: s_1 \to 1 \\ f_o &: s_2 \to 1 \\ f_o &: s_3 \to 0 \end{aligned}$$

ou seja, cada vez que a máquina recebe zero, ela escreve o símbolo de saída correspondente ao seu estado e muda para o estado seguinte; e quando recebe um, ela escreve o símbolo de saída de seu estado e permanece no mesmo estado. As saídas correspondentes estão descritas na tabela 5.1.

Vamos, em seguida, considerar um conjunto de entradas para esta máquina descrita pelas funções da tabela 5.1 e, depois, vamos observá-la em operação, em cada passo, até chegar ao resultado final.

Estado	Entrada → Novo Estado	Saída
s_0	$0 \rightarrow s_1$ $1 \rightarrow s_0$	0
s_1	$0 \rightarrow s_2$ $1 \rightarrow s_1$	1
s_2	$0 \rightarrow s_3$ $1 \rightarrow s_2$	1
s_3	$0 \rightarrow s_0$ $1 \rightarrow s_3$	0

Tabela 5.1: Operações da máquina de estado

Digamos, por exemplo, que esta máquina receba, como entrada, a seqüência $\{0, 1, 1, 0, 1, 0, 1, 0\}$. Se observarmos a descrição desta máquina, a sua saída deve resultar em $\{0, 1, 1, 1, 1, 1, 0, 0\}$.

Esta operação pode ser observada passo por passo, na seqüência de figuras 1–8[2].

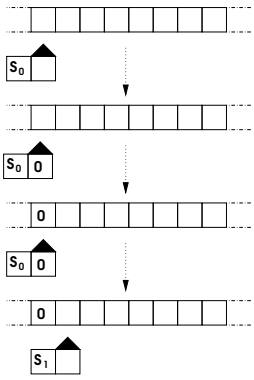


Figura 5.1: Passo 1

No primeiro passo (figura 5.1), a máquina se encontra no estado s_0, recebe 0 como entrada, escreve 0 (que é a sua função de saída) na posição onde se encontra e passa para o estado s_1 na posição seguinte.

[2]As imagens representam uma "fita" de dados em branco e uma "cabeça" de gravação que se movimenta sempre para a direita depois de gravar a sua função de saída e passar para o novo estado (que pode ser o mesmo).

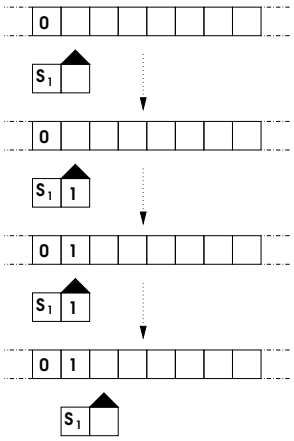


Figura 5.2: Passo 2

No segundo passo (figura 5.2), a máquina se encontra no estado s_1, recebe 1 como entrada, escreve 1 (sua função de saída) e permanece no estado s_1.

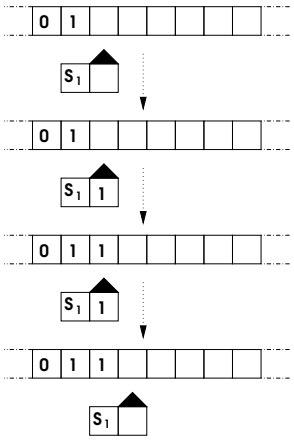


Figura 5.3: Passo 3

No terceiro passo (figura 5.3), a máquina se encontra no estado s_1, recebe 1 como entrada, escreve 1 (sua função de saída) e permanece no estado s_1.

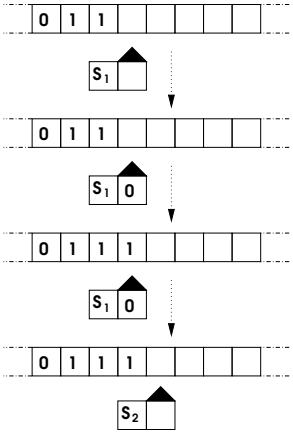


Figura 5.4: Passo 4

No quarto passo (figura 5.4), a máquina se encontra no estado s_1, recebe 0 como entrada, escreve 1 (sua função de saída) e passa para o estado s_2.

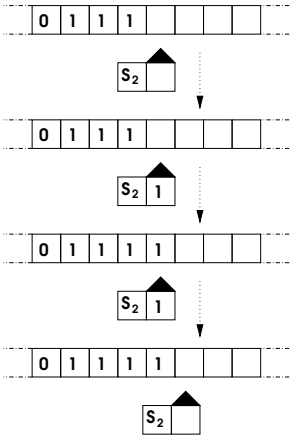


Figura 5.5: Passo 5

No quinto passo (figura 5.5), a máquina se encontra no estado s_2, recebe 1 como entrada, escreve 1 (sua função de saída) e continua no estado s_2.

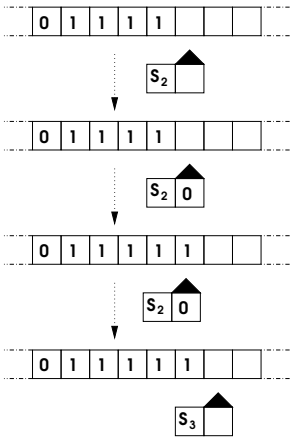


Figura 5.6: Passo 6

No sexto passo (figura 5.6), a máquina se encontra no estado s_2, recebe 0 como entrada, escreve 1 (sua função de saída) e passa para o estado s_3.

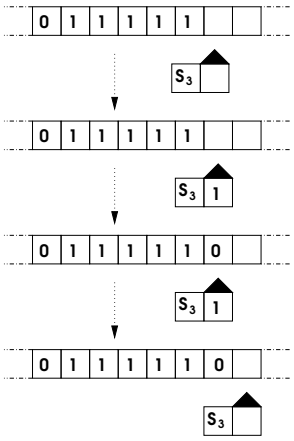


Figura 5.7: Passo 7

No sétimo passo (figura 5.7), a máquina se encontra no estado s_3, recebe 1 como entrada, escreve 0 (sua função de saída) e continua no estado s_3.

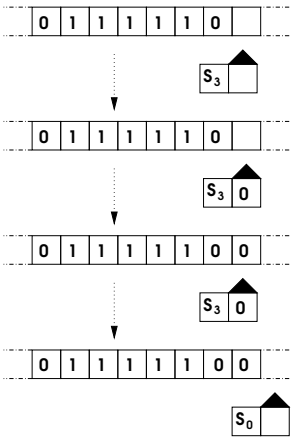


Figura 5.8: Passo 8

No oitavo passo (figura 5.8), a máquina se encontra no estado s_3, recebe 0 como entrada, escreve 0 (sua função de saída) e passa para o estado s_0. Uma vez que não há mais dados de entrada, a máquina para.

5.2 Exercícios

1. Uma geladeira tem um sistema de termostato que deve manter a temperatura interna entre 2 e 5 graus. Projete uma máquina de estado para o controle de temperatura desta geladeira, implmentando um algoritmo para este processo.

2. Dado qualquer arquivo de texto, voce deve projetar e implementar uma máquina de estado finito para a contagem de palavras neste arquivo.

5.3 Máquinas de Turing

Embora uma máquina de estado finito possa representar um procedimento computacional, nem todos os procedimentos podem ser representados por uma máquina de estado finito, por exemplo, quando se torna necessário o uso de memória adicional. Tais procedimentos podem ser simulados por um outro tipo de máquina de estado que se denomina **Máquina de Turing** em homenagem a Alan Turing[3], que a propos em 1936 como um sistema formal que fazia operações computacionais.

[3]Matemático inglês que, independente e simultaneamente a Alonso Church, provou a tese da computabilidade e criador da máquina "bombe" para decifrar o código "enigma" durante

Com base na manipulação de símbolos e um sistema de regras, Turing demonstrou que era possível automatizar operações, tornando possível o processamento simbólico e a abstração de sistemas computacionais e dos conjuntos de operações numéricas.

De uma maneira mais genérica, podemos considerar a máquina de Turing como um dispositivo semelhante à uma máquina de estado finito, com a habilidade adicional de reler dados de entrada e, ainda, apagar e regravar os dados de entrada como uma memória ilimitada. O fato de a memória de uma máquina de Turing ser ilimitada faz com que ela seja uma "máquina ideal" (um modelo) e não uma "máquina real".

Uma máquina de Turing é definida, basicamente, por 4 funções declaradas em um conjunto de 5 elementos de entrada. Estas funções são:

1. Pára (não faz nada);
2. Imprime um símbolo do alfabeto definido na célula lida (pode ser o mesmo);
3. Passa para o próximo estado (pode ser o mesmo);
4. Move a cabeça de leitura para a direita ou esquerda, de uma célula.

Os 5 elementos de entrada formam uma quíntupla (e,s,i,p,d) que representam:

e	s	i	p	d
↓	↓	↓	↓	↓
estado atual	símbolo atual	símbolo impresso	próximo estado	direção do movimento

Por exemplo [6], considere uma máquina de Turing, descrita pelas seguintes quíntuplas :

(0,0,1,0,D)
(0,1,0,0,D)
(0,b,1,1,E)
(1,0,0,1,D)
(1,1,0,1,D)

Considerando uma fita, particular, podemos determinar o resultado desta máquina de Turing, conforme a figura 5.9.

a segunda guerra mundial.

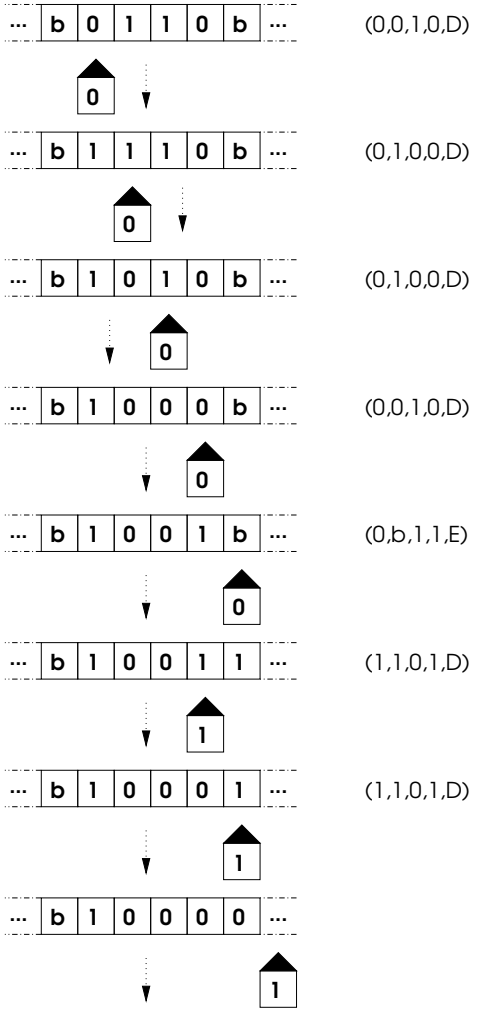


Figura 5.9: Exemplo do funcionamento de uma máquina de Turing

No último estado, a leitora encontra uma célula em branco e como não existe uma quíntupla que descreva a ação a ser tomada ela pára. Neste caso, quando o estado inicial é 1 e o símbolo lido é b não existe uma quíntupla como $(1, b, x, y, z)$.

5.4 Exercícios

1. Encontre uma máquina de Turing que calcule a função:

$$f(n) = \begin{cases} n & \Longleftrightarrow \quad n\text{par} \\ n+1 & \Longleftrightarrow \quad n\text{impar} \end{cases}$$

2. Projete um algoritmo para descrever uma máquina de Turing que calcule a função:

$$f(n_1, n_2) = n_1 \times n_2$$

Capítulo 6

Algoritmos

A solução de problemas usando-se a computação é feita por meio de algoritmos. Computar, neste caso, não significa, necessariamente, um processo em um computador digital, mas o ato de computar calculando-se o resultado de uma expressão a partir de um dado de entrada.

Normalmente, um algoritmo é uma série de instruções que completam uma tarefa. A partir de um determinado valor de entrada, o dado é processado e retorna um resultado. Podemos dizer que, para solucionar uma equação, o algoritmo é a maneira como a equação é calculada.

Por exemplo, vamos considerar a equação 6.1:

$$f(x) = x^2 + 4 \tag{6.1}$$

neste caso o algoritmo algébrico é a forma de se calcular um resultado a partir da entrada de um valor arbitrário de x. O processo pode ser descrito como:

'Dado um valor de x, multiplique-o por ele mesmo e some a constante 3"

Um programa de computador pode implementar este algoritmo para calcular esta equação e, neste caso, o próprio algoritmo algébrico deve ser implementado em uma linguagem de programação para que o computador digital, ao executar o programa, faça a tarefa automaticamente.

Na linguagem C, poderíamos escrever:

No caso do exemplo da listagem 6.1 o valor de x é fixo ($x = 3$)e o programa sempre retorna o mesmo valor. Ele pode ser modificado para receber um valor arbitrário de x e efetuar o cálculo.

Listagem 6.2: Rotina da função f em Assembly

Listagem 6.1: Programa fdx.c

```
#include <stdio.h>

int f (int); /* funcao que calcula f(x) */

int main (){
  int x, y;
  x=3;
  y=f(x); /* chamada da funcao */
  printf("f(x) = %d\n", y);
  return 0;
}

int f (int x){
  return (x*x + 4);
}
```

```
    pushl       $3, %esp
    movl        %esp, %ebp
    movl        %ebp, %eax
    imull       %ebp, %eax
    addl        $4, %eax
    leave
    ret
```

Observe no código Assembly da listagem 6.2[1] que a rotina faz um `pushl` do valor 3 no endereço `%esp` e depois o move para o endereço `%ebp`, tira uma cópia para `%eax`, multiplica `%ebp` e `%eax`, armazenando o resultado em `%eax` e depois soma o valor 4 a `%eax`, retornando o valor em `%eax`.

6.1 Algoritmos determinísticos e não-determinísticos

O computador digital funciona como uma máquina de estado determinística, ou seja, a partir de um *input* gera sempre o mesmo *output*. No entanto, os algoritmos podem ser de dois tipos:

[1]Obtido pela compilação parcial do código em C para obter o Assembly (`gcc -c -Wa,-adln, fdx.c >fdx.a`).

- Determinístico: Um algoritmo totalmente previsível a partir de determinada entrada.

- Não determinístico: Um algoritmo conceitual que permite mais de uma etapa em um determinado momento em que, dependendo do passo tomado o resultado pode ser diferente.

Nota: Conceitualmente, um algoritmo não-determinístico pode rodar em um computador determinístico com um número ilimitado de processadores paralelos. Cada vez que ocorre mais de um passo possível, novos processos são paralelizados e, uma vez um processo terminado com sucesso, um resultado é obtido.

Algoritmos que usam números aleatórios (randômicos) não são determinísticos desde que a geração destes números não seja feita de modo "pseudo-randômico". Por exemplo, considere o programa da listagem 6.3

Listagem 6.3: Programa rand.c

```
/* rand.c */
#include <stdio.h>
#include <sys/types.h>
#include <time.h>

int main(){
  int i;
  time_t t1;
  (void) time(&t1);
  srand48((long) t1);
  /* usa time em segundos para a semente (Seed) */
  printf("5 nums. randomicos (Seed = %d): \n",(int) t1);

  for (i=0;i<5;++i){
    printf("%d\n", lrand48());
  }
}
```

Neste caso existe uma "semente" (*seed*) que é utilizada para a função `srand48`, esta semente é baseada no tempo do sistema. Desta forma, cada vez que o programa rodar, o resultado será diferente. Este algoritmo pode ser considerado não-determinístico.

6.2 Otimização de algoritmos

A construção de um algoritmo nem sempre resulta na forma mais otimizada para o processamento. Muitas vezes, um algoritmo deve ser analizado para que o resultado seja mais rápido e mais eficiente. Vamos observar um problema real e verificar como este pode ser resolvido.

Considere uma parábola descrita pela equação $y = x^2$. Considere, ainda, a rotação desta parábola de um ângulo de 30^o. A equação que descreve os valores dos novos x' e y' pode ser calculada a partir da multiplicação das matrizes:

$$(x \quad x^2) \times \begin{pmatrix} \cos 30 & \sin 30 \\ -\sin 30 & \cos 30 \end{pmatrix} = (x\cos 30 - x^2 \sin 30 \quad x \sin 30 + x^2 \cos 30)$$

ou seja, os novos x' e y' são dados por:

$$x' = x\cos 30 - x^2 \sin 30 \qquad \text{e} \qquad y' = x \sin 30 + x^2 \cos 30$$

Dados n valores de x, podemos implementar o algoritmo:

```
for (i=0; i<n; i++){
  novo_x[i] = x[i]*cos(30) - x*x*sin(30);
  novo_y[i] = x[i]*sin(30) + x*x*cos(30);
}
```

Entretanto, tal algoritmo chama as funções `sin()` e `cos()` $2n$ vezes. Se, porém, implementarmos o algoritmo como:

```
s=sin(30);
c=cos(30);
for (i=0; i<n; i++){
  novo_x[i] = x[i]*c - x*x*s;
  novo_y[i] = x[i]*s + x*x*c;
}
```

as funções `sin()` e `cos()` são chamadas apenas uma vez cada uma.

6.3 Precisão de algoritmo

Não é raro um algoritmo ficar limitado a uma determinada precisão. Alguns cálculos computacionais são feitos por aproximação e deve-se ter em conta a precisão desta aproximação.

Por exemplo, podemos dizer que o número 1 pode ser obtido pela soma dos termos infinitos da série:

$$1 = \sum_{n=1}^{\infty} \frac{1}{n(n+1)}$$

Quanto maior o número de termos n mais próximo do valor 1 fica a expressão. Assim, podemos escrever um programa para calcular esta aproximação:

Listagem 6.4: Programa ser1.c

```
/* ser1.c */
#include <stdio.h>

int main(){
  double um=0.0, n=1.0, err=1.0, trunc=0.0;
  printf("Entre o limite de erro desejado: ");
  scanf("%lf", &trunc);

  while (err > trunc){
    um = um + 1.0/(n*(n+1.0));
    n = n + 1.0;
    err = 1.0 - um;
  }
  printf("err = %1.10lg\tum = %1.10lf\tn = %d\n",
      err, um, (int) n);
  return 0;
}
```

Entretanto, se definirmos, no programa da listagem 6.4 as variáveis como tipo `float`, ficamos limitados ao tamanho do ponto flutuante para calcular o erro desejado da aproximação:

```
float um=0.0, n=1.0, err=1.0, trunc=0.0;
printf("Entre o limite de erro desejado: ");
scanf("%f", &trunc);
```

Neste caso, se optarmos por um erro menor que 0.00015 o programa não conseguirá convergir, entrando em um *loop*.

Outro problema a ser tratado é o arredondamento feito pelo programa para formatar a saída. Aparentemente, podemos achar que o resultado converge quando, na verdade, se aumentarmos a precisão, observamos que tal fato não acontece. Por exemplo:

```
printf("err = %lg\tum = %1.4f\tn = %d\n", err, um, (int) n);
```

neste caso, `%1.4f` vai arredondar a quinta casa depois do ponto flutuante, ou seja, qualquer resultado maior que 0.9999 será exibido como 1.0000, dando a falsa impressão de que a expressão convergiu com sucesso ao seu limite.

6.4 Complexidade

Antes de iniciarmos o estudo de algoritmos, é importante que se tenha em conta que pode existir mais de uma solução computacional para um mesmo problema. Entretanto, uma solução pode ser eficiente na medida de sua complexidade. Mas, exatamente, o que é complexidade?

No caso de algoritmos, complexidade não se refere à implementação mas sim à quantidade de vezes que um determinado passo de um programa é executado, ou seja, é relativo à recursividade. Por exemplo, considere o código:

```
for (i=0; i<=n; i++){
  printf("%d\n", i);
}
```

A complexidade deste algoritmo (normalmente designada por θ, Ω ou O[2]) é linear, ou seja $O(n)$, porque é diretamente proporcional ao número de passos (n) necessários para completar a instrução `for`.

As expressões de complexidade têm significados diferentes:

$$\begin{array}{rcl} f(n)O(g(n)) & \rightarrow & f(n) \text{ aumenta mais rapidamente que } g(n) \\ f(n)\Omega(g(n)) & \rightarrow & f(n) \text{ aumenta menos rápido que } g(n) \\ f(n)\theta(g(n)) & \rightarrow & f(n) \text{ aumenta tão rápido quanto } g(n) \end{array}$$

Existem casos onde a complexidade não é uma função linear, pois pode ser resultado de uma combinação de complexidades, dependendo da quantidade de passos dentro de um mesmo passo como, por exemplo, no caso de laços aninhados (*nested loops*).

Considere, por exemplo, o caso de um algoritmo que compare duas listas de um mesmo tamanho n, onde cada ítem de uma lista deve ser comparado com o de outra lista:

[2] O símbolo O é originado da teoria dos números de Edmund Landau e é conhecido como "símbolo de Landau".

```
...
n = tam_lista;
for(i=0; i<n; i++){
  for(j=0; j<n; j++){
    c = compara(lista1[i], lista2[j]);
  }
}
...
```

a complexidade é proporcinal a $n \times n$, ou quadrática. Neste caso dizemos que a complexidade é $O(n^2)$.

A principal diferença entre os casos é que o segundo algoritmo tem dois laços o que o torna proporcionalmente mais lento na medida que n aumenta. Observa-se que o número de laços aninhados aumenta a complexidade de forma exponencial. Por exemplo, se existirem x laços aninhados, a complexidade pode ser $O(n^x)$. Veja o perfil gráfico das funções de complexidade na figura 6.1.

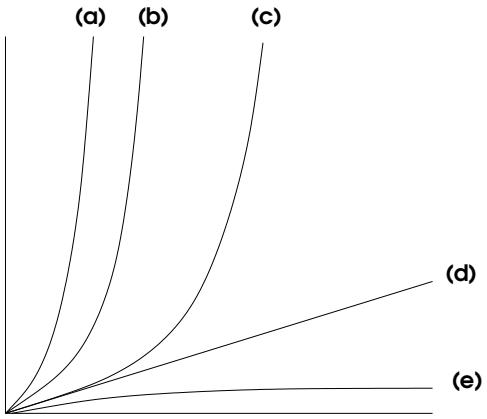


Figura 6.1: Complexidades (a) $O(n^3)$; (b) $O(n^2)$; (c) $O(n \log n)$; (d) $O(n)$; e (e) $O(\log n)$.

Evidentemente, se considerarmos laços aninhados com números diferentes de passos, a complexidade será proporcional ao múltiplo dos passos de cada laço, por exemplo, o algoritmo:

```
...
for(i=0; i<n; i++){
  for(j=0; j<m; j++){
```

```
      faz(algo[i],algo[j]);
    }
  }
  ...
```

será de complexidade $\theta(n \times m)$ ou $O(mn)$.

Consideremos, agora, um conjunto de n dados que devem ser tratados recursivamente mas, a cada passo, pode-se dividir o conjunto por 2, ou seja, em cada passo seguinte, o conjunto passa a ser $n/2$. Neste caso, o aumento de n aumenta a complexidade de forma logarítmica ($\theta(\log_2 n)$) e dizemos que o algoritmo é de complexidade $O(\log n)$.

No caso de um laço de complexidade linear ($O(n)$) conter um laço de complexidade logarítmica ($O(\log n)$), dizemos que a complexidade é $O(n \log n)$.

6.5 Algoritmos de Ordenação e Busca

Em várias situações, um conjunto de dados deve ser ordenado para facilitar sua utilização, por exemplo, uma lista de nomes que deve ser ordenada em ordem alfabética, ou uma lista de números que precisa ser classificada em ordem crescente ou decrescente e assim por diante. Estes algoritmos são chamados **algoritmos de ordenação** (*sort algorithms*) e são de diferentes graus de complexidade e de eficiência.

6.5.1 Método "bolha" (bubble sort)

O algoritmo de "bolha" leva este nome porque, em um vetor vertical de dados, o algoritmo parece fazer os elementos menores "borbulharem" para suas posições. Este é um dos algoritmos mais antigos e tende a ser bem simples quanto à implementação em qualquer linguagem de programação. Ele funciona por meio da comparação de cada item de uma lista com o item seguinte, trocando-os quando necessário. Por exemplo, considere a lista:

21 50 46 35 13 90 85 32

Na primeira passagem (percurso) pela lista acontece o seguinte:

Listagem 6.5: Função bubbleSort

```
void bubbleSort(int numbers[], int array_size){
  int i, j, temp;
  for (i = (array_size - 1); i >= 0; i--){
    for (j = 1; j <= i; j++){
      if (numbers[j-1] > numbers[j]){
        temp = numbers[j-1];
        numbers[j-1] = numbers[j];
        numbers[j] = temp;
      }
    }
  }
}
```

21 com 50	→	não troca
50 com 46	→	troca
50 com 35	→	troca
50 com 13	→	troca
50 com 90	→	não troca
90 com 85	→	troca
90 com 32	→	troca

resultando na lista:

21 46 35 13 50 85 32 90

O algoritmo repete o processo até que a lista seja repassada sem troca dos itens, ou seja, os ítens estão na ordem correta (veja listagem 6.5).

Este algoritmo é considerado um dos mais ineficientes pois no melhor dos casos (a lista já se encontra ordenada) ele é de ordem n. Geralmente, o algoritmo é de ordem $O(n^2)$.

A vantagem do algoritmo é a facilidade de sua implementação e para casos de ordenação de poucos elementos ele pode ser o método escolhido. No entanto, para grandes quantidades de dados ele é ineficiente.

6.5.2 Método Troca de partição (Quick Sort)

O algoritmo é baseado na escolha de um elemento qualquer da lista ("pivot"), de forma que se possa criar uma lista de elementos menores e outra de elementos maiores que ele. Por exemplo, considere a lista:

21 50 46 35 13 90 85 32

Tomemos o 35 como "pivot"; então a teremos a lista (211332) como menor que o pivot e a lista (50469085) como maior, resultando na lista:

(21 13 32) 35 (50 46 90 85)

Recursivamente, podemos escolher novos pivots para as listas menores e maiores, digamos, 21 e 50, respectivamente, obtendo: (13) e (32) como listas menor e maior da menor, e (46) e (9085) como listas menor e maior da maior, resultando em:

(13) 21 (32) 35 (46) 50 (90 85)

E assim, sucessivamente. Fica claro que a estratégia é "dividir para conquistar" e o algoritmo é extremamente rápido (ver listagem 6.6).

A ordem de complexidade do algoritmo pode ser $O(n^2)$, no pior dos casos (a lista está ordenada e o primeiro ítem é escolhido como pivot), mas escolhendo-se, aleatoriamente, um pivô em uma lista randômica, a complexidade do algoritmo cai para $O(n \log n)$.

Listagem 6.6: Funçoes para o quickSort

```
void quickSort(int numbers[], int array_size){
  q_sort(numbers, 0, array_size - 1);
}

void q_sort(int numbers[], int left, int right){
  int pivot, l_hold, r_hold;
  l_hold = left;
  r_hold = right;
  pivot = numbers[left];
  while (left < right){
    while ((numbers[right] >= pivot) && (left < right)){
      right--;
    }
    if (left != right){
      numbers[left] = numbers[right];
      left++;
    }
    while ((numbers[left] <= pivot) && (left < right)){
      left++;
    }
    if (left != right){
      numbers[right] = numbers[left];
      right--;
    }
  }
  numbers[left] = pivot;
  pivot = left;
  left = l_hold;
  right = r_hold;
  if (left < pivot){
    q_sort(numbers, left, pivot-1);
  }
  if (right > pivot){
    q_sort(numbers, pivot+1, right);
  }
}
```

Apêndice A

Introdução à programação funcional

Não se trata de uma apologia da linguagem funcional! Afinal, nem tudo comporta uma programação estritamente funcional. E, evidentemente, integrar processos funcionais e procedurais com uma boa dose de bom senso e sabedoria é o que se deseja...

O motivo principal para a inserção, neste curso, de uma introdução à uma linguagem funcional está nas possibilidades que esta forma de programação oferece na compreensão de estrutura de dados e na falta de material didático, em português, sobre programação funcional. Evidentemente, também existe o interesse em divulgar a Linguagem Funcional como linguagem de programação no Brasil.

Em nosso curso de estrutura de dados, quando utilizarmos uma linguagem funcional para explicitar dados, tipagem e algoritmos estaremos usando a linguagem Haskell e o compilador Glasgow Haskell Compiler (GHC) por sua versatilidade e compatibilidade com bibliotecas de outras linguagens. Os interpretadores Hugs e GHCi serão usados em vários exemplos. De maneira geral, os conceitos podem ser extendidos para outras linguagens funcionais como o Clean, Caml, Erlang[1] etc..

[1]Para maiores informações sobre o Clean e OCAML visite os sites: `http://www.cs.kun.nl/~clean`, `http://caml.inria.fr/` e `http://www.erlang.org/`.

A.0.3 Linguagem Funcional - O que é e porque usar?

Programação Funcional é algo realmente difícil de se definir. Primeiramente, porque não há consenso sobre uma definição que agrade a todos e, em segundo lugar porque cada livro ou tutorial traz uma definição com uma ênfase diferente em certos aspectos inerentes da linguagem funcional.

Estritamente falando, Programação Funcional é um estilo de programação baseado em avaliação de expressões. Daí deriva o nome "funcional", uma vez que a programação é orientada para a avaliação de funções (no senso matemático e estrito da palavra). Nas linguagens funcionais, as expressões são formadas por funções que combinam valores.

Diferentemente das linguagens imperativas (baseadas na arquitetura de von Neumann), as linguagens funcionais são classificadas como declarativas, juntamente com as linguagens lógicas.

É fácil perceber isto em um exemplo prático. Vamos tomar, por exemplo, o cálculo da soma de uma progressão aritmética de 1 a 10 com razão 1. Em uma linguagem procedural como C poderíamos escrever:

Listagem A.1: Programa em C para cálculo da soma dos termos de uma PA de 0 a 10 e razão 1.

```
#include<stdio.h>
#include<stdlib.h>
int main(){
  int soma, i;
  soma = 0;
  for (i=1; i<=10; i++){
    soma += i;
  }
  printf("%d\n", soma);

  return 0;
}
```

Em HASKELL bastaria escrever:

```
main = print (somaPA)
somaPA = sum [1..10]
```

Neste contexto, [1..10] representa uma lista de 1 a 10 e `sum` é uma função que calcula a soma dos valores de uma lista. É evidente que a função `sum` já se encontra implementada, mas podemos implementar uma função, construindo-a a partir da sua definição. Por exemplo, para implementar uma função `soma`:

```
soma :: forall a. (Num a) => [a] -> a
soma (x:xs) = x + soma (xs)
```

Neste exemplo, a primeira linha quer dizer:

> "Soma é definida para todo 'a' que seja numérico, recebendo uma lista de números e retornando um número."

Esta é a forma de tipagem da função e, neste caso, `a` vem da palavra inglesa *any* que significa qualquer. A definição da função, propriamente dita, se encontra na linha seguinte:

> "Soma uma lista de 'xis' e 'xises' é igual a 'xis' mais a soma de 'xises'. "

Listagem A.2: Programa em HASKELL para cálculo da soma dos termos de uma PA de 1 a 10 e razão 1.

```
main print(somaPA)

somaPA = soma [1..10]

soma :: forall a. (Num a) => [a] -> a
soma(x:xs) = x + soma(xs)
```

O programa da listagem A.2 exemplifica como ficaria o código completo. Pode-se perceber, rapidamente, as vantagens da linguagem de programação funcional, não apenas no tamanho do código, mas em sua legibilidade e elegância matemática. Além disso, outras vantagens da programação funcional estão na modularidade que este tipo de linguagem possibilita. Esta modularidade, implícita na orientação ao objeto, é nativa na linguagem funcional devido à sua capacidade de trabalhar com funções hierárquicas (*higher-order functions*) e *lazy evaluation*[2].

[2]Para traduzir *lazy evaluation* seria mais adequada chamá-la de avaliação "esperta"e não de "preguiçosa"como se poderia esperar.

Além destas vantagens, os programas escritos em linguagem funcional apresentam outras. Por exemplo, uma chamada de função não tem outra atribuição além de computar o resultado. Isto evita os efeitos colaterais e grande parte das fontes de bugs e, uma vez que os valores de uma expressão não podem ser alterados, ela pode ser calculada em qualquer parte do programa.

Normalmente, na programação procedural, sua natureza "imperativa" requer um fluxo definido e seqüencial de instruções e isto não existe no estilo funcional de programação. Para os que estão acostumados com linguagens como C / C++, Pascal, Delphi, Java etc. isto pode parecer um tanto absurdo. Não é raro pensarmos em um fluxo de instruções mesmo quando usamos a linguagem funcional. Claro que, essas vantagens podem ser associadas às vantagens de outras linguagens e, como dito na vinheta acima, saber dosar as vantagens de uma ou outra linguagem no desenvolvimento de um aplicativo é essencial para o sucesso de um projeto de programa.

A.0.4 Funções hierárquicas

Como já dissemos, a construção de um programa funcional é fundamentada na computação de funções. Tais funções podem ser agrupadas em funções de outros níveis hierárquicos (*higher-order functions*) até que, finalmente a partir de uma entrada, a função produza o resultado desejado. Por exemplo, se quisermos saber se um número é primo ou não, podemos escrever:

Listagem A.3: Módulo HASKELL para verificar se um número é primo.

```
module Eprimo (ehPrimo) where

ehPrimo :: Int -> Bool
ehPrimo n = (verif n == [])
        where
          verif n = [ x | x <- [2..n-1], n `mod` x == 0]
```

A função `ehPrimo` é definida apenas se a função `verif` retornar uma lista vazia. Observe que, na listagem A.3 a função `verif` de n retorna uma lista vazia apenas se não existir x, onde o resto de todas as divisões de n por x seja zero, sendo x uma lista $[2, 3, 4, ..., n-1]$. A função `ehPrimo` computa se n é uma lista vazia. Como `ehPrimo` recebe um número inteiro e retorna uma expressão booleana ("verdadeiro" ou "falso"), caso `ehPrimo` de n retornar uma lista vazia, o módulo retorna "True".

Note que, no exemplo da listagem A.3, que determina se um número é primo ou não a definição é bastante próxima da definição matemática de número primo e está mais ligado com o "o que é uma função" do que "como achar uma função". Esta diferença faz com que tais programas sejam escritos em um estilo "declarativo" em contraposição ao estilo "imperativo" das linguagens procedurais.

Além disso, não existe um controle de fluxo no algoritmo, não importando onde, dentro do programa, cada função é declarada e em que ordem ela deve ser calculada. Por exemplo, cada elemento de [2..n-1] é testado para verificar n. Entretanto, o algoritmo não especifica a seqüência em que isto deve ser feito. Voce pode imaginar com que facilidade se pode testar os elementos da lista em dois processadores paralelos? Ou, ainda, dividir o processamento entre vários processadores? Pois é, o paralelismo no cálculo de funções é muito fácil de ser implementado, neste tipo de programação.

A.0.5 Lazy Evaluation

O conceito de *Lazy Evaluation* está ligado a uma estratégia de avaliação de um conjunto de dados (*e.g.* uma lista). No entanto, é importante ressaltar que este tipo de avaliação, também chamada de avaliação prorrogada (*delayed evaluation*), é uma técnica computacional que atrasa a computação de uma expressão até que o resultado no passo anterior ao necessário seja conhecido.

Haskell é uma linguagem *lazy*, ou seja, nenhuma computação acontece sem que ela seja necessária. Isto quer dizer que uma expressão em Haskell somente será avaliada se o seu resultado for usado. Esta propriedade da linguagem (e de outras linguagens funcionais) possibilita a criação de estruturas de dimensões infinitas, embora não se use toda a estrutura.

Em uma linguagem procedural, uma função que cria uma lista infinita pode ser criada. Entretanto, uma vez chamada a função ela executará um *loop* infinito (veja listagem A.4).

Listagem A.4: Exemplo de criação de uma lista infinita em C

```
List makeList()
{
List current = new List();
current.value = 1;
current.next = makeList();
return current;
```

```
}
```

A análise do código da listagem A.4 mostra que a função cria uma lista, atribui o valor 1 e então, recursivamente, chama a mesma função para criar o restante da lista. Claro que a execussão do programa somente terminará quando não houver mais memória, porque `makeList` vai ser chamada infinitamente.

As linguagens funcionais podem, assim, usar esta estratégia para avaliar tipos infinitos de dados e listas infinitas, como veremos posteriormente em nosso curso. Isto é possível porque as funções são tratadas de forma não-estrita, contrariamente às linguagens procedurais onde as funções são tratadas de forma estrita, ou seja, conforme descritas pelo usuário.

A.0.6 Um exemplo

Tomemos um exemplo, com o propósito de tornar esta linguagem mais familiar. Vamos considerar um problema onde precisamos encontrar o maior número entre três algarismos inteiros. Assim, a entrada do programa deve ser de três algarismos do tipo inteiro. Vamos, em seguida, dar um nome para a nossa função de `maxTres` que tem o tipo:

```
Int -> Int -> Int -> Int
```

Os três primeiros tipos são relativos à "entrada" da função e o último ao resultado. Em nosso problema, o primeiro passo poderia ser comparar dois números inteiros para obter o maior dentre eles. Para isso, podemos criar uma função `maxi` para efetuar esta computação:

```
maxi a b | a>=b      = a
         | otherwise = b
```

A solução de nosso problema requer que `maxTres` compute `maxi a (maxi b c)` (lembre-se do conceito de funções hierárquicas) assim, o algoritmo seria escrito:

```
maxTres a b c = maxi a (maxi b c)
```

A generalização do problema pode ser extendida para a computação do máximo de uma lista de inteiros. Assim, uma função mais genérica, que podemos chamar de `maxList` seria do tipo:

```
maxList :: [Int] -> Int
```

Uma das primeiras preocupações do programa é: o que fazer com uma lista vazia? Digamos que uma lista vazia retorne 0 e uma lista não vazia tenha outras implicações. Neste caso, teremos que considerar qualquer posição do maior elemento dentro da lista. Uma lista pode ser descrita como formada de uma "cabeça" x e uma "cauda" xs que é representada por $(x : xs)$. Assim, podemos definir nossa função como:

```
maxList (x:xs) = maxi x (maxList xs)
```

Podemos ver, aqui, de que maneira podemos reutilizar uma função previamente definida. Assim, o programa todo poderia reaproveitar o código escrito para calcular outras funções:

Listagem A.5: Exemplo de funções hierárquicas

```
module Maxi (
    maxi,
    maxiTri,
    maxiLst
    ) where

maxi :: Ord a => a -> a -> a
maxi a b | a>=b      = a
         | otherwise = b

maxiTri :: Ord a => a -> a -> a -> a
maxiTri a b c = maxi a (maxi b c)

maxiLst :: (Ord a, Num a) => [a] -> a
maxiLst [] = 0
maxiLst (x:xs) = maxi x (maxiLst xs)
```

Uma vez construido o módulo com as funções (ver listagem A.5) podemos utilizá-las em qualquer programa que necessite destas funções, que pode ser compilado junto com este módulo. Por exemplo, um programa para determinar o maior valor dos elementos de uma lista dada poderia ser escrito como:

Listagem A.6: Exemplo de programa para determinar o maior valor dos elementos de uma lista.

```
module Main (main) where

import Maxi (maxiLst)

main = print (maxiLst [30,29,54,34,95,12,49])
```

Observe na listagem A.6 que o módulo `Main` importa o módulo `Maxi` e deste, importa apenas a função necessária (`maxiLst`).

Os módulos podem ser carregados em um interpretador e a construção de suas funções podem ser testadas.

Usando um interpretador

Para testar as funções e os conceitos, neste curso, vamos utilizar um interpretador da linguagem HASKELL . Para testá-los os voce pode carregá-los em um interpretador como o **GHCi** ou **Hugs**. Para isso recomendamos instalar o GHC (Glasgow Haskell Compiler) que instala o interpretador[3].

Para este procedimento, digite o código em um editor de texto plano (joe, vi, pico etc.) salve o arquivo com a extensão ".hs" (por exemplo: `maxil.hs`) e inicie o interpretador na linha de comandos, digitando `ghci`:

```
-bash-2.05b$ ghci
```

Verá que será carregado o interpretador e os pacotes necessários, terminando com um novo sinal de prompt (`Prelude>`):

```
   ___         ___ _
  / _ \ /\  /\/ __(_)
 / /_\// /_/ / /  | |      GHC Interactive, version 5.04, for Haskell 98.
/ /_\\/ __  / /___| |      http://www.haskell.org/ghc/
\____/\/ /_/\____/|_|      Type :? for help.

Loading package base ... linking ... done.
Loading package haskell98 ... linking ... done.
Prelude>
```

[3]Neste curso utilizaremos um terminal de um sistema Unix (o FreeBSD) para as atividades e práticas dos alunos.

Carregue o programa usando o comando :load ou :l, por exemplo:

```
Prelude> :l maxl
Compiling Maxi             ( Maxi.hs, interpreted )
Ok, modules loaded: Maxi.
*Maxi>
```

O interpretador compilará o módulo carregado, interpretará o módulo e carregará as funções, apresentando um novo prompt (*Main¿). Caso voce deseje saber quais os tipos definidos nas funções, voce pode usar o comando :browse:

```
*Maxi> :browse *Maxi
maxiTri :: forall a. (Ord a) => a -> a -> a -> a
maxi :: forall a. (Ord a) => a -> a -> a
maxiLst :: forall a. (Ord a, Num a) => [a] -> a
*Maxi>
```

Finalmente, experimente o programa para ver se tudo está funcionando como deveria, por exemplo, digite:

```
*Main> maxiLst [3,7,4,53,9,0,1]
53
*Main>
```

Repita os testes com alguns argumentos diferentes e experimente calcular usando uma lista vazia (`maxiLst []`).

Usando um compilador

Compilar nada mais é do que juntar as idéias... Afinal, resolvendo um determinado problema por meio de um programa de computador o que se faz, via de regra, é "fracionar" a solução e, depois, juntar as partes, as funções e as rotinas ou procedimentos.

Em C, por exemplo, juntamos partes de código (.c) com bibliotecas ou cabeçalhos (.h) e "linkamos" os objetos (.o) em um programa que possa ser executado pela máquina e que produza a solução necessária para o referido problema.

Para isto, é necessário um outro programa chamado compilador. Cada linguagem tem seus compiladores e existem vários compiladores de linguagem

funcional. Muitos deles já se encontram em estágios bastante avançados, enquanto outros ainda são experimentais. Neste nosso curso, como já afirmamos, vamos focar a linguagem Haskell e seu compilador mais flexível, o Glasgow Haskell Compiler (GHC)[4]. Nada impede que se use outros compiladores como o Clean que é outro compilador de linguagem funcional excelente, muito estável e poderoso. Como já justificamos antes, a escolha do Haskell se deu por contingência didática, uma vez que a sua sintaxe é bastante fácil e clara, além de ser muito próxima do Cálculo Lambda.

Compilar requer alguns truques para que o código gerado seja rápido, eficiente e enxuto. Qualquer compilador, de qualquer linguagem, oferece opções de compilação que vão desde a compilação estática (com todas as bibliotecas de funções incluídas no código binário final) até a compilação que usa bibliotecas dinâmicas (por ex.: ELF no Linux ou DLL em outro sistema). A grande flexibilidade de opções de compilação do GHC permite a ligação (linking) com as bibliotecas e/ou suas partes.

Apresentamos, a seguir, algumas das principais opções de compilação do GHC. Para os que preferirem o clm do Clean, é bom lembrar que existem diferenças de sintaxe e de opções para a linha de comando. Ao emitir o comando "clm -h", o compilador do Clean retorna um resumo das várias opções.

O compildor GHC pode ser invocado de várias maneiras. Quando invocado com a opção –interative corresponde ao comando "ghci" e o módulo é carregado no interpretador. Quando invocado com a opção –make o compilador compila vários módulos, checando as dependências automaticamente. Outra maneira invocá-lo com as opções de compilação ou, simplemente. com a opção de arquivo de saída e o arquivo a ser compilado:

```
ghc -o <arquivo_de_saída> <arquivo_de_entrada>
```

A linha de comando para chamar o compilador toma o nome do programa (ghc) e outros argumentos. Cada argumento precede os módulos que ele afeta. Por exemplo:

```
ghc -c -O1 teste1.hs -O2 teste2.hs
```

As opções podem ser passadas para o compilador em pragmas dentro do arquivo-fonte. Por exemplo, se desejar compilar um arquivo-fonte com a opção -fglasgow-exts, ela pode ser inserida na primeira linha do código como um pragma:

[4]Página *Web* do GHC: `http://www.haskell.org/ghc/`

```
{-# OPTIONS -fglasgow-exts #-}
module x where
...
```

Normalmente, o compilador emite vários tipos de alertas (*warnings*) sendo que estas opções podem ser controladas na linha de comando. Os alertas, na compilação padrão, são: `-fwarn-overlapping-patterns, -fwarn-deprecations, -fwarn-duplecate-exports, -fwarn-missing-fields e -fwarn-missing-methods`. Além destas, o uso de `-W` emite, além dos alertas padrão, mais `-fwarn-incomplete-patterns` e a opção -w desliga todos os alertas. A opção `-Wall` emite todos os tipos de alertas.

Mais informações podem ser encontradas no *site* do GHC, na área de documentação: `http://haskell.org/haskellwiki/GHC:Documentation`.

A.1 Programação Funcional e listas

Uma vez que uma das principais estruturas em dados podem ser organizadas em listas, como veremos mais adiante em nosso curso, vamos explorar um pouco mais o conceito, tipagem e construção de listas em Haskell .

Agora que já conhecemos um pouco mais sobre tipos e listas, é importante saber que o Haskell (e outras linguagens funcionais como o Clean, Caml e Erlang) apresenta algumas funções definidas para a manipulação de dados de uma lista. Mas, o que há de tanta importância em listas. Bem, primeiramente, já vimos que uma cadeia de caracteres, nada mais é do que uma lista, depois, grande parte das operações matemáticas e tratamento de dados são feitos sobre listas, por exemplo, dados de uma tabela de banco de dados, vetores são listas, matrizes são listas de listas etc.. Se considerarmos que endereços de memória, set de instruções de processadores, fluxo de dados etc., também podem ser tratados como listas, justificamos porque lidar com listas é tão importante. Mais adiante, em nosso curso de Estrutura de Dados, vamos tratar deste assunto com mais detalhes.

A.1.1 Mais sobre listas

A melhor maneira de compreender as funções de uma linguagem é usando-as. Então, vamos dar alguns exemplos de funções que operam com listas. Digamos que a partir de uma função qualquer, queiramos que ela seja aplicada a toda a lista. Para isso podemos usar a função `map`, onde `map f l` aplica uma função `f` a uma lista `l`. Por exemplo:

Listagem A.7: Exemplo do código do mapeamento de uma função

```
main = print(map quad [1..10])

quad :: Int -> Int
quad n = n*n
```

Como pode ser observado, a função `map` "mapeia" uma função em todo o argumento, ou seja, uma lista. O aluno pode testar esta função no interpretador, por exemplo:

```
Prelude> let quad x = x*x
Prelude> map quad [1..10]
[1,4,9,16,25,36,49,64,81,100]
Prelude> map quad (filter even [1..10])
[4,16,36,64,100]
Prelude>
```

O trabalho com listas é bastante prático para lidar com dados. Por isso vamos expor mais um exemplo. Digamos que seja necessário eliminar elementos adjacentes em duplicata, dentre os elementos de uma lista. Então, podemos criar uma função, que chamaremos de `remdup`, para resolver o problema:

Listagem A.8: Função para remover dados duplicados em uma lista.

```
module RemDup (remdup) where

remdup :: [Int] -> [Int]
remdup [] = []
remdup (x:xs@(y:_)) | x == y    = remdup xs
                    | otherwise = x:(remdup xs)
remdup xs = xs
```

Por enquanto vamos utilizar o interpretador para compreender melhor o estilo da linguagem funcional. Mais adiante vamos discutir o problema de entrada/saída (IO) em HASKELL . Outra novidade neste nosso programa é o aparecimento do padrão "_" que funciona como "coringa" (*wildcard*) que pode ser substituído por qualquer argumento. Ainda, o "@" que é chamado de

aliás (*as-pattern*). Estes aparecem sempre como `<identificador>@<padrão>`, onde o identificador representa todo o objeto e o padrão o decompõe em seus componentes.

Para compreender o funcionamento deste programa, carregue-o no interpretador e experimente:

```
*Main> remdup [1,2,2,3,4,4,5]
[1,2,3,4,5]
*Main>
```

É importante ressaltar que os aliases "@" têm precedência sobre o construtor ":" e `a:b@c:d` vai associar `b` com `c` e não com `c:d`. Quando aplicada a expressão `a@(b:c@(d:e))` no argumento `[1,2,3,4]`, teremos `a = [1,2,3,4]`, `b = 1`, `c = [2,3,4]`, `d = 2` e `e = [3,4]`.

Agora observe o que acontece se a lista for uma lista de caracteres:

```
*Main> remdup "Jooao"
:1:
Couldn't match 'Int' against 'Char'
Expected type: [Int]
Inferred type: [Char]
In the first argument of 'remdup', namely '"Jooao"'
In the definition of 'it': remdup "Jooao"
*Main>
```

Isto acontece porque a função `remdup` foi tipada como `Int` (veja na listagem A.8). Como `remdup` foi declarada como uma função que recebe uma lista de inteiros e retorna uma lista de inteiros, ao tentarmos aplicar a função a uma lista de caracteres o programa avisa que não pode "casar" o tipo inteiro com o tipo caractere. Este é um problema típico de estrutura de dados, pois as operações sobre um tipo é limitada pela tipagem dos dados sobre os quais operam.

Para resolver o problema de forma que a função seja aplicável a uma lista genérica de qualquer tipo, vamos introduzir o conceito de **polimorfismo**. Dizemos que uma variável é polimórfica quando aceita qualquer tipo de argumento. Em linguagem funcional usa-se a letra "a" (de *any*) para nos referirmos a este tipo de variável. Então, para que nosso programa funcione para qualquer tipo de lista podemos escrever:

Listagem A.9: Função para remover dados duplicados em uma lista de qualquer tipo.

```
module RemDupAny (remdupany) where

remdupany :: (Eq a) => [a] -> [a]
remdupany [] = []
remdupany (x:xs@(y:_)) | x == y    = remdupany xs
                       | otherwise = x:(remdupany xs)
remdupany xs = xs
```

Agora o programa pode tratar qualquer tipo de lista. Carregue-o no interpretador e experimente com todos os tipos de lista. Note que para definirmos a função usamos uma classe "Eq" que é definida como:

```
class Eq a where {
(/=) :: a -> a -> Bool {- has default method -};
(==) :: a -> a -> Bool {- has default method -};
}
```

Elementos em uma classe são ditos estarem em um contexto. Neste caso, restringimos que as variáveis polimórficas "a" são válidas no contexto onde podem ser testadas pela equalidade (igual `(==)` ou diferente `(/=)`). Tal procedimento é necessário, pois existem casos em que definir [a] -¿ [a] não permitem outras comparações como "menor que" (¡) ou "menor ou igual a" (¡=). Na prática, nem todos os objetos estão sujeitos a um teste de inequalidade. Observe ainda, na listagem acima, outro tipo de comentário: `{- ... -\}`. Este tipo de comentário é equivalente ao tipo `/* ... */` do C, ou seja, comentário de multiplas linhas.

As classes que podem ser utilizadas em declarações com atributos polimórficos são:

- **Eq a** - Restringe o uso em objetos que permitem ser testados quanto à equalidade (== e ¯).
- **Ord a** - Restringe o uso em objetos que podem ser testados quanto à ordenação (<, <=, > e >=).
- **Num a** - Esta classe define que as funções podem ser aplicadas a objetos definidos como numéricos (`Int, Float, Real, Complex, Ratio` etc.).

A.1.2 Programação Funcional e E/S (IO)

A maioria dos programas têm que estabelecer um diálogo ou uma interface com o usuário, com outros programas e/ou arquivos. Para que isto seja possível, estas ações de "entrada" e "saída" (E/S) de dados dependem do tipo de dados que o programa recebe para tratar e que tipos de dados o programa devolve. Em linguagem funcional, a entrada de dados está, geralmente, ligada aos objetos (argumentos) a serem tratados pelas funções. A saída é o resultado desta computação.

Hudak [8] chama a atenção sobre a diferença entre o sistema de E/S das linguagens procedurais (imperativas) e as funcionais (declarativas). Para ele, o sistema de E/S é tão robusto, apesar de funcional, quanto nas liguagens tradicionais de programação. Nas linguagens de programação imperativas o processo é feito por ações e modificações de estado por meio da leitura, alteração de variáveis globais, escrita de arquivos, leituras a partir do terminal e criação de "janelas". Estas ações também podem ser levadas a cabo em um estilo funcional mas, são claramente separadas do cerne funcional da linguagem.

Para lidar com o problema de E/S, a linguagem funcional usa um conceito matemático de **Mônadas**[5], mas o conhecimento estrito do conceito de mônada não é necessário para que possamos compreender como funciona o processo Por enquanto, vamos considerar a **classe IO ()** como um tipo abstrato de dado[6].

Operações de IO

O melhor jeito de compreender as ações de E/S de um sistema é observando, na prática como tudo acontece. Vamos, então, observar como podemos passar um caractere para um programa em Haskell .

Use o compilador Haskell (GHC):

```
ghc -o <nome_do_executável> <nome>.hs
```

No exemplo da listagem A.10, começamos importanto uma função `getChar` de um módulo chamado "IO". Na verdade, o programa funcionaria sem isto, uma vez que esta função já se encontra definida no "Prelude". Entretanto,

[5]Mônadas são construtores de tipos usados com o paradigma de computação sequencial (imperativa), implementada de forma funcional (declarativa).

[6]A linguagem Clean utiliza o conceito de "tipo único", no lugar de mônadas, para lidar com o problema de E/S.

Listagem A.10: Exemplo de IO em linguagem funcional.

```
import IO (getChar)
main :: IO ()
main = do
   putStrLn "Entre 's' ou 'n'e tecle <enter>:"
   carct <- getChar
   let resp | carct == 's'   = "sim"
            | carct == 'n'   = "não"
            | otherwise  = "nem sim, nem não"
   putStrLn resp
```

estamos fazendo isso apenas para demonstrar como se importa algo de um módulo. Esta declaração pode ser comparada ao `#include` do C. Em seguida declaramos que o main é do tipo `IO ()`. O restante do programa é bastante legível. A função `getChar` espera um caractere do teclado para o terminal e o passa para a variável `carct`. Neste caso ela toma apenas um caractere, sendo que se digitarmos mais de um caractere, apenas o primeiro será aceito. A função `putStrLn` imprime uma linha (incluindo
n) no terminal.

Outro tipo de IO seria, por exemplo, o de comunicação entre o programa e arquivos. Para isso, podemos programar um aplicativo que conta as linhas, as palavras e os caracteres de um arquivo texto que chamaremos de "meu-wc.hs" (algo semelhante com o comando wc do unix).

Na listagem A.11, o tipo "$"é um operador de baixa precedência à direita e "." é um operador de alta precedência à direita. Isto significa que a expressão:

```
nWords = length . words $ fstr
```

é equivalente à `length (words (fstr))`.

Manipulando arquivos

A manipulação de arquivos em Haskell ou outra linguagem funcional não difere muito das liguagens tradicionais. Por exemplo, se quisermos copiar um arquivo para outro é preciso usar um manipulador (*handle*) das operações

Listagem A.11: Exemplo de IO com arquivos.

```
import System (getArgs)
main :: IO ()
main = do
   args <- getArgs
   case args of
       [fname] -> do fstr <- readFile fname
             let nWords = length . words $ fstr
                 nLines = length . lines $ fstr
                 nChars = length fstr
             putStrLn "lin\tpal\tcar\tarq"
             putStrLn . unwords $ [ show nLines, "\t"
                                  , show nWords, "\t"
                                  , show nChars, "\t"
                                  , fname]
       _ -> putStrLn "uso: meu-wc <nome_do_arquivo>"
```

de E/S. Quando importamos o módulo IO várias classes e tipos podem ser utilizados e que não estão presentes no módulo padrão (Prelude).

Vejamos então um exemplo prático que podemos chamar de "meu-cp.hs", por ser uma variante do "cp" do unix.

Note que quando abrimos um arquivo, é criado um manipulador do tipo `Handle` para as transações de E/S (listagem A.12). Ao fecharmos o manipulador associado ao arquivo, fechamos o arquivo. Observe também que `IOMode` é uma classe definida como:

```
data IOMode = ReadMode | WriteMode | AppendMode | ReadWriteMode
```

e `openFile` é descrito como:

```
openFile :: FilePath -> IOMode -> IO Handle
```

Um ponto interessante a ser comentado é que, como a função `getContents` usa um *handle* de E/S do tipo `String`:

Listagem A.12: Exemplo de manipulação de arquivos.

```
import IO

main :: IO ()

main = do
   fromHandle <- getAndOpenFile "Copiar de: " ReadMode
   toHandle   <- getAndOpenFile "Copiar para: " WriteMode
   contents   <- hGetContents fromHandle
   hPutStr toHandle contents
   hClose toHandle
   putStrLn "Feito."

getAndOpenFile          :: String -> IOMode -> IO Handle

getAndOpenFile prompt mode =
   do putStrLn prompt
     name <- getLine
     catch (openFile name mode)
           (\_ -> do putStrLn ("Impossível abrir "++ name)
                     getAndOpenFile prompt mode)
```

```
getContents :: IO String
```

Por isso, pode parecer que a função lê um arquivo ou uma entrada inteira e que resulta em uma performance comprometida em certas condições de espaço de memória e tempo de processamento. Aí está uma grande vantagem da linguagem funcional. Como uma *string* é uma lista de caracteres, a função pode retornar uma lista sob demanda devido ao caracter não-estrito da avaliação (*lazy evaluation*).

A.1.3 Módulos

Quando se trata de programação em geral, tornar os programas modulares permite o rápido e eficiente reaproveitamento do código. Normalmente, em C/C++ a criação de bibliotecas (*libs*), cabeçalhos (*header files*), funções e classes são responsáveis pela modularidade e orientação ao objeto (OOP). Nas liguagens funcionais, as funções, classes e tipagens (*types*) podem ser agrupadas ou divididas em módulos. Estes módulos servem para, principalmente, controlar o domínio das funções e abstrações de tipagem dos dados.

Um módulo em HASKELL , por exemplo, contém várias declarações que já vimos. Geralmente, encontramos declarações de delimitação de domínio, declarações de tipagem e dados, declarações de classes e instâncias, definições de funções etc.. Exceto pelo fato de que as declarações a serem importadas de outros módulos, devam ser declaradas no início de um módulo, as declarações do módulo podem aparecer em qualquer ordem.

A listagem A.13 traz um exemplo de um módulo um pouco mais complexo. Este exemplo mostra a definição de um módulo `HasScales` que define os tipos `Scale` e `Sequence` e as funções `makeScale`, `major`, `minor`, `modifyScale`, `flatten`, `sharpen`, `scaleNote` e `scaleNotes`. Além disso, este módulo importa declarações de outro módulo chamado `Haskore`. O tipo `Scale`, por exemplo, é definido como uma lista de `AbsPitch` que é um tipo descrito no módulo `Haskore` importado por este módulo e definido como um `Int`. Na verdade ela serve para a conversão de notas musicais em números, da seguinte maneira:

```
type AbsPitch = Int

absPitch :: Pitch -> AbsPitch
```

Listagem A.13: Exemplo retirado dos módulos do HasChorus v. 1.1

```
module HasScales where

import Haskore

type Scale = [AbsPitch]

makeScale :: [AbsPitch] -> PitchClass -> Scale
makeScale ns pc = map (+ (pitchClass pc)) ns

major, minor :: PitchClass -> Scale
major    = makeScale [0,2,4,5,7,9,11]
minor    = makeScale [0,2,3,5,7,8,10]

modifyScale :: \
            (AbsPitch -> AbsPitch) -> Int -> Scale -> Scale
modifyScale f n sc   = pre ++ ((f it) : post)
   where
     (pre, (it:post)) = splitAt (n-1) sc

flatten, sharpen :: Int -> Scale -> Scale
flatten    = modifyScale (subtract 1)
sharpen    = modifyScale (+1)

type Sequence = [AbsPitch]

scaleNote :: Scale -> AbsPitch -> AbsPitch
scaleNote sc 0        = 0
scaleNote sc (n+1)    = (possibles!!) n
   where
     possibles = concat $ iterate (map (+12)) sc

scaleNotes :: Scale -> [Int] -> Sequence
scaleNotes sc    = map (scaleNote sc)
```

```
absPitch (pc,oct) = 12*oct + pitchClass pc

pitch    :: AbsPitch -> Pitch
pitch    ap       = ( [C,Cs,D,Ds,E,F,Fs,G,Gs,A,As,B] !! mod ap 12,
                      quot ap 12)

pitchClass :: PitchClass -> Int
pitchClass pc = case pc of
     Cf -> -1;  C -> 0;  Cs -> 1
     Df -> 1;   D -> 2;  Ds -> 3
     Ef -> 3;   E -> 4;  Es -> 5
     Ff -> 4;   F -> 5;  Fs -> 6
     Gf -> 6;   G -> 7;  Gs -> 8
     Af -> 8;   A -> 9;  As -> 10
     Bf -> 10;  B -> 11; Bs -> 12
```

Assim, o módulo herda as declarações do módulo importado. Entretanto, em alguns casos, é interessante importar apenas algumas declarações de um módulo. Seja por razões de economia de código, seja para evitar redefinições de outras funções que podem ser sobrescritas durante a importação[7]. Neste caso, deve-se optar pela importação apenas da declaração que interessa, como já vimos em um exemplo da listagem A.10.

```
import IO (getChar)
```

Neste caso, apenas a declaração de `getChar` é exportada pelo módulo IO para o programa que o está importando.

Por meio da construção de módulos, pode-se definir tipos abstratos de dados ou *Abstract Data Types* (ADT). Nestes casos, o módulo exporta apenas um tipo *hidden* de representação e todas as operações são levadas a termo em um nível de abstração que não depende de como o objeto é representado.

Principais aspectos dos módulos

Ao utilizar os módulos, devemos levar sempre em consideração. os seguintes aspectos:

[7]Existem opções de alerta do compilador para evitar conflitos e substituições de declarações. Além disso, existem maneiras de se evitar o conflito explícito de entidades com o mesmo nome.

- Uma declaração `import` pode ser seletiva, por meio do uso de cláusulas `hiding` que permitem a exclusão explícita de entidades ou objetos.
- A declaração `import` pode usar a cláusula as quando for conveniente usar um qualificador diferente do nome do módulo importado.
- Os programas importam, implicitamente por default, o módulo `Prelude`. Se for necessário excluir algum nome durante a importação, é necessário explicitá-la. Por exemplo:

  ```
  import Prelude hiding lenght
  ```

  não importará a função `lenght`, permitindo que se possa redefiní-la.
- Instâncias (*instances*) não são especificadas na lista de importação ou exportação de um módulo. Normalmente, todos os módulos exportam as declarações de `instance` e cada importação traz essas declarações para o escôpo do programa.
- As classes podem ser declaradas com construtores, entre parênteses contendo o nome da classe e suas variáveis.

Nomes qualificados

Um problema óbvio é o fato de poder acontecer de diferentes entidades ter o mesmo nome. Para que se possa evitar este problema é necessário importar usando o atributo `qualified` para a entidade cujo nome se deseja qualificar. O nome importado receberá, como prefixo, o nome do módulo importado seguido de ponto (“.”). Assim, por exemplo:

```
import qualified IO (getChar)
```

Neste caso, o nome `getChar`, será referenciado por `IO.getChar`[8].

[8]Note que x.y é diferente de x . y. O primeiro é um nome qualificado enquanto o segundo é uma função “.” (infix).

Apêndice B

Cálculo λ: Uma introdução

A grande maioria dos processos computacionais são baseados em operações sobre dados discretos e quase todos os tipos de operações sobre dados discretos são, em geral, operações sobre vetores finitos. Em termos de algoritmo, isto significa operar sobre listas finitas de elementos.

A característica principal da linguagem funcional, que pode ajudar na solução de problemas desta natureza, é o fato das linguagens funcionais usarem a avaliação sob demanda (*lazy evaluation*) em listas finitas ou infinitas de dados. Além disso, a facilidade de implementar-se algoritmos paralelos é uma vantagem das linguagens funcionais sobre as procedurais.

A compreensão da base matemática destas linguagens — o cálculo–λ — é fundamental para a compreensão destas estruturas.

B.1 O Cálculo-λ

Um dos mais importantes conceitos matemáticos para a programação funcional foi, com certeza, o cálculo-λ, desenvolvido por Church e Kleene [4]. A teoria desenvolvida por ele tinha como princípio demonstrar a computabilidade efetiva de expressões, ou seja, o caculo-λ pode ser usado para definir, claramente, o que é uma **função computável**[1].

Entretanto, o mais impressionante produto de suas deduções foi o fato de que, reduzindo-se um processo a um conjunto de regras matemáticas simples, é possível reduzir-se as chances de erro.

[1]Paralelamente e independentemente, Allan Turing chegou à mesma conclusão usando o conceito de máquinas de estado, motivo pelo qual a tese é conhecida como Tese de Church-Turing

O cálculo-λ influenciou muitas linguagens de programação funcional. LISP foi a primeira e o *pure* LISP é verdadeiramente funcional. Outras linguagens atuais — HASKELL, MIRANDA, ML e CLEAN — foram desenvolvidas a partir do cálculo-λ, baseadas na definição formal e na semântica denotativa que ele permite.

Em 1940, Haskell B. Curry [5] formulou a teoria da lógica combinatória como uma teoria de funções livres de variáveis. Apesar de fundamentada na teoria de funções de Schönfinkel [14] e no cálculo-λ de Church, sua teoria foi fundamental para o desenvolvimento das linguagens funcionais.

Conceitos

A simplicidade do cálculo-λ pode, algumas vezes, parecer confusa para quem está acostumado ao formalismo algébrico tradicional. Vejamos alguns exemplos.

Imagine, por exemplo, uma função "frita" que, a partir algumas entradas, produza os seguintes resultados:

$$\begin{aligned} \text{banana} &\rightarrow \text{banana frita} \\ \text{batata} &\rightarrow \text{batata frita} \\ \text{aipim} &\rightarrow \text{aipim frito} \\ \text{ovo} &\rightarrow \text{ovo frito} \end{aligned}$$

Na notação tradicional, podemos escrever que $f(x) = x$ frito e esta função pode ser descrita no cálculo-λ como:

$$\lambda x.x \text{ frito}$$

então, se a aplicarmos a um argumento "repolho" a aplicação se reduzirá a "repolho frito":

$$(\lambda x.x \text{ frito}) \text{ repolho} \rightarrow \text{repolho frito}$$

O resultado de uma aplicação λ pode ser, também, uma função quando uma expressão λ é aplicada sobre outra expressão. Vamos utilizar um exemplo de uma função "cobertura" que aplicada à uma expressão retorna uma outra função:

$$(\lambda y.\lambda x.x coberturay) \text{ caramelo} \rightarrow \lambda x.x \text{ caramelado}$$

A sintaxe do cálculo-λ é bastante simples. Ela está fundamentada em identificadores de variáveis, constantes, abstrações de funções, aplicação de funções e subexpressões.

B.1.1 Computabilidade de funções

Para compreender a teoria do cálculo-λ, é necessário que se observe com maiores detalhes as definições das fórmulas matemáticas. Para isso vamos começar com o conceito de identificador delimitado.

Em uma equação, por exemplo,

$$f(x) = x^2 \tag{B.1}$$

podemos dizer que x é uma variável e a expressão $f(x)$ assumirá valores que dependem do valor de x. Ora, esta relação é ambígua, pois se quisermos determinar se $f(x) \leq 25$ a variável x somente poderá assumir valores de -5 a 5, pois para $x < -5$ ou $x > 5$, o resultado será falso. Então temos que explicitar que $x^2 \leq 25 \iff \exists x | x \geq -5,\ x \leq 5$, ou seja $x^2 \leq 25$ se existir x tal que x é maior ou igual a -5 e menor ou igual a 5. Entretanto, se quisermos determinar se $f(x) < x^4$, o resultado será sempre verdadeiro, independentemente do valor que x venha a ter. Este problema está relacionado à computabilidade de uma função e é um dos desafios da lógica simbólica.

O desafio da lógica simbólica de encontrar um algoritmo geral, que decida se uma declaração de primeira ordem é universalmente válida ou não, é conhecido como *Entscheidungsproblem* (problema da decisão). Isto foi provado impossível por Church e Turing, em 1936.

Vamos, então, observar estas entidades matemáticas com maior detalhe. Na equação B.2,

$$\sum_{x=1}^{n} x^2 \tag{B.2}$$

x assumirá valores de $[1 \cdots n]$ e seus quadrados serão somados, ou seja, $[1 + 4 + 9 + \cdots + n^2]$. Neste caso, x pode ser chamado de uma variável dependente ou um identificador que está delimitado (de 1 a n). Note que um identificador delimitado (*bound identifier*) faz com que qualquer das expressões das equações B.3, sejam equivalentes à equação B.2.

$$\sum_{y=1}^{n} y^2,\ \sum_{k=1}^{n} k^2,\ \sum_{a=1}^{n} a^2, \cdots \tag{B.3}$$

Da mesma forma, na teoria dos conjuntos, podemos escrever $x|x \geq 0 = y|y \geq 0$ ou, ainda, $\forall x[x+1 > x]$ é equivalente a $\forall y[y+1 > y]$.

Quando temos uma expressão do tipo:

$$f(x) = x^2 + 4 \tag{B.4}$$

dizemos que o parâmetro formal x é o identificador delimitado, ou seja, a função retornará algum valor para para qualquer que seja o valor de x de forma indefinida. Pode-se perceber facilmente que para $-\infty \leq x \leq \infty$ a função retornará valores entre 4 e ∞.

Podemos definir os termos do cálculo-λ (λ-termo[2].) como termos construídos a partir de recursividades sobre variáveis $x, y, z, \ldots$ que podem assumir as formas descritas na figura B.1

x	`variável`
$\lambda.M$	`abstração onde M é um termo`
(M, N)	`aplicação onde M e N são termos`

Figura B.1: Formas dos termos no cálculo-λ.

Em $(\lambda x.M)$, x é denominada variável dependente (*bounded variable*) e M é o termo que sofre a abstração. Cada ocorrência de x em M está vinculada à abstração. Assim, podemos escrever que:

$$\begin{array}{lcl} (\lambda x.N)M & \equiv & \texttt{let x = M in N} \\ (\lambda f.N)(\lambda x_1 \ldots x_k.M) & \equiv & \texttt{let f x1 ... xk = M in N} \end{array}$$

Podemos representar a equação B.4, em termos do cálculo-λ, como:

$$\lambda x(x^2 + 4) \tag{B.5}$$

e, se $f \equiv \lambda x(x^2 + 4)$, podemos representar $f(2)$ como:

$$\lambda x(x^2 + 4)(2) \tag{B.6}$$

[2]Em programação usa-se `\x` para representar λx

B.1.2 Cópia e reescrita

O cálculo é feito substituindo-se todas as ocorrências de x por 2, ou seja, $(2^2 + 4) \Rightarrow 8$. Este tipo de operação é denominado "regra de cópia" (*copy rule*), uma vez que para $f(2)$ copiamos o valor 2 em todas as ocorrências de x. No cálculo-λ existem apenas três símbolos: $\lambda, (,)$ e identificadores. Estes são regidos por quatro regras básicas:

1. Se x é um identificador, então x é uma expressão do cálculo-λ;
2. Se x é um identificador e E é uma expressão do cálculo-λ, então λxE é uma expressão do cálculo-λ, chamada "abstração". Assim, x é o identificador que delimita a abstração e E o corpo da abstração.
3. Se F e E são expressões do cálculo-λ, então FE é uma expressão do cálculo-λ, chamada "aplicação". Assim, F é o operador da aplicação e E o seu operando.
4. Uma expressão λ é produzida, sempre, pela aplicação múltipla e finita das regras 1–3.

Como estas regras operam por recursividade, podemos dizer que, uma vez que esta abstração é definida em um determinado domínio, ela pode ser aplicada em qualquer entidade. Por exemplo, digamos que queiramos aplicá-la ao domínio de z^2. Então, podemos escrevê-la como:

$$\lambda x(2x + 3)(z^2) \tag{B.7}$$

A equação B.7 pode ser, então, re-escrita por $2z^2 + 3$. Note, por exemplo que se quisermos expressar a função de adição $f(a, b) = a + b$, podemos escrever: $\lambda ab(a + b)$ que, por sua vez, pode ser re-escrita por $\lambda a(\lambda b(a + b))$, que é denominado "currying"[3].

A esta altura voce deve estar pensando, mas que vantagens há nisto? Não seria melhor usar a velha e tradicional notação? Vejamos! Uma vez que podemos representar $f(x)$ simplesmente por fx e, sendo a aplicação da abstração sempre associativa à esquerda, podemos evitar as confusões de parênteses no caso de $f(x)(y)$, $(f(x))y$ e representar apenas por fxy. Outra vantagem é que podemos escrever a identidade como $\lambda a(a)$ e uma identidade que aplica uma combinação em si mesma como $\lambda a(aa)$. Veja, por exemplo, como uma combinação que inverte a ordem de dois combinadores pode, facilmente, ser expressa por $\lambda ab(ba)$.

[3]Essa transformação é possível porque $\lambda ab(a) = \lambda a(\lambda b(a))$. O nome "currying" é usado por ter, essa expansão, sido definida por Haskell B. Curry.

B.1.3 Redução

As expressões do cálculo-λ são computadas por "redução". Por exemplo, digamos que seja preciso avaliar a expressão:

$$\lambda y(y^2)(\lambda x(x^3+2)(2)) \tag{B.8}$$

A equação B.8 pode ser calculada por substituições e reduções sucessivas, da seguinte forma:

$$\begin{aligned} \lambda y(y^2)(\lambda x(x^3+2)(2)) &\Rightarrow \lambda y(y^2)(2^3+2) \\ \lambda y(y^2)(2^3+2) &\Rightarrow \lambda y(y^2)(10) \\ \lambda y(y^2)(10) &\Rightarrow (10^2) \\ (10^2) &\Rightarrow 100 \end{aligned}$$

A semântica do cálculo-λ estabelece duas regras para a redução, na computação de expressões:

1. Regra de renomeação: Uma expressão pode ser reduzida a outra pela renomeação de um identificador delimitado, por qualquer outro identificador que não esteja contido na expressão.

2. Regra de substituição: Uma subexpressão da forma $(\lambda xE)A$ pode ser reduzida pela substituição de uma cópia de E na qual toda ocorrência livre de x é substituída por A, desde que isto não resulte em que qualquer identificador livre de A torne-se um delimitador.

Estas regras são conhecidas por redução-α e redução-β, respectivamente. Para compreendê-las melhor, vamos detalhar um pouco mais estas regras.

A primeira pode ser exemplificada da seguinte maneira: seja uma expressão λxM que pode ser renomeada pela expressão λyN, onde N foi obtida pela renomeação do identificador x em M pelo identificador y. Isto só é possível se y não ocorrer em M. Por exemplo:

$$\lambda x(2x+z) \Rightarrow \lambda y(2y+z) \tag{B.9}$$

Uma vez que y não ocorre à esquerda da equação B.9, ele pode ser usado para renomear x. O mesmo não seria possível no caso de renomear x por z, o que resultaria em erro[4].

Na prática, imagine uma função:

[4]Neste caso a redução resultaria em $\lambda z(3z)$.

$$f = \lambda x(\lambda z(z + x)(4)) \Rightarrow f = \lambda x(4 + x)$$

Imagine, agora, a aplicação errada da redução-α renomeando x para z:

$$f = \lambda x(\lambda z(z + x)(4)) \Rightarrow f' = \lambda z(\lambda z(z + z)(4)) = \lambda z(8)$$

Observe que f' não é a mesma função f, pois $f(1) = 5$ e $f'(1) = 8$.

No caso da regra de redução-β vamos considerar a expressão $\lambda x(2x+1)(3)$. Neste caso, a regra de substituição tem um operador que é uma abstração e pode ser reduzido pela substituição de 3 em toda a ocorrência de x em $2x + 1$, o que resulta em $2 \times 3 + 1 = 7$. No caso de

$$\lambda x(\lambda y(xy))n$$

a aplicação é uma abstração $\lambda x(\lambda y(xy))$ e usando-se a redução-β em todas as ocorrências livres de x teremos:

$$\lambda y(ny)$$

Vejamos, então a restrição da regra da redução-β. Considere a aplicação:

$$\lambda y(\lambda x(\lambda z(z + x)(3))(y))(1) \tag{B.10}$$

Neste caso, podemos aplicar a redução:

$$\begin{aligned} \lambda y(\lambda x(\lambda z(z + x)(3))(y))(1) &\Rightarrow \lambda y(\lambda z(z + y)(3))(1) \\ \lambda y(\lambda z(z + y)(3))(1) &\Rightarrow \lambda z(z + 1)(3) \\ \lambda z(z + 1)(3) &\Rightarrow 3 + 1 \\ 3 + 1 &\Rightarrow 4 \end{aligned}$$

Note que no primeiro passo da redução-β da equação B.10 o identificador x foi substituído por y, então y foi substituído por 1 e, finalmente, z por 3. Vamos, entretanto, observar o que acontece se no primeiro passo substituímos z por y:

$$\begin{aligned} \lambda y(\lambda x(\lambda z(z + x)(3))(y))(1) &\Rightarrow \lambda y(\lambda x(\lambda y(y + x)(3))(y))(1) \\ \lambda y(\lambda x(\lambda y(y + x)(3))(y))(1) &\Rightarrow \lambda y(\lambda y(y + y)(3))(1) \\ \lambda y(\lambda y(y + y)(3))(1) &\Rightarrow \lambda y(6)(1) \\ \lambda y(6)(1) &\Rightarrow 6 \end{aligned}$$

O que aconteceu? Acontece que, quando z foi substituído por y, no primeiro passo, houve uma colisão de identificadores, uma vez que y é um identificador livre em y mas não em $\lambda y(y + x)(3)$. A partir daí, houve uma mudança no significado da expressão, resultando em um valor diferente do esperado. Note a importância desta regra de redução para se evitar colisões entre os identificadores.

B.1.4 Forma normal

Chamamos de "forma normal" a máxima redução possível de uma expressão, ou seja, a regra de substituição não pode mais ser aplicada. A computabilidade de uma expressão depende de sua forma normal ser computável, isto é, quando uma expressão atinge sua forma normal como uma resposta. A tabela B.1 mostra alguns exemplos de expressões em sua forma não-normal e normal.

Não-normal	**Normal**
$\lambda x(x)(y)$	y
$\lambda y(fy)(a)$	fa
$\lambda x(\lambda y(xy))(f)$	$\lambda y(fy)$
$\lambda x(xx)(y)$	yy

Tabela B.1: Exemplos de forma normal, cf. MacLennan [10].

B.2 Aplicações

Embora possa haver aplicações diretas do cálculo-λ no cálculo de funções computáveis, sua aplicação está mais ligada à compreensão das bases teóricas das linguagens funcionais.

O uso de linguagens funcionais para implementar algoritmos de processamento de listas e vetores é uma escolha quase natural. Uma das grandes vantagens das linguagens funcionais é a maneira pela qual elas lidam com listas de dados. Estas funções se aplicam às operações dos coeficientes de análise e manipulação de vetores que representam os dados digitais e discretos, muito comuns em computação, considerando-os como listas finitas.

B.2.1 Exercícios

Reduza às formas normais, se possível, as seguintes expressões λ:

1. $(\lambda x.Ax)(k)$
2. $(\lambda x.Ay)(k)$
3. $(\lambda x.Ax(Kxj))(m)$
4. $(\lambda x.(\lambda x.Kxx)(j))(m)$
5. $(\lambda x.(xy))y$
6. $(\lambda x.x(xy))(\lambda u.u)$
7. $(\lambda x.xxy)(\lambda x.xxy)$
8. Prove que $(\lambda x.x^2)(\lambda y.(\lambda z.(z-y)a)b) \equiv a^2 + 2ab - b^2$

Apêndice C

Exercícios complementares

C.1 Lista de Exercícios - ARRAY

Exercício 1. Considere os escalares $r = 3$ e $s = -2$ e as matrizes abaixo:

$$A = \begin{pmatrix} 2 & 1 \\ -1 & 0 \\ 3 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} 4 & 1 & 2 \\ 6 & -1 & 5 \\ 1 & 3 & 2 \end{pmatrix} \qquad C = \begin{pmatrix} 2 & 4 \\ 6 & -1 \end{pmatrix} \qquad D = \begin{pmatrix} 4 & -6 \\ 1 & 3 \\ 2 & -1 \end{pmatrix}$$

Calcule as seguintes operações (quando possível):

(a) $A + D$
(b) $A - D$
(c) rB
(d) sD
(e) $A + rD$
(f) $B \cdot D$
(g) $D \cdot C$
(h) $A \cdot C$
(i) $C \cdot A$
(j) C^2

Exercício 2. Dadas as matrizes:

$$A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix} \qquad C = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & 2 \\ 0 & 0 & 3 \end{pmatrix} \qquad D = \begin{pmatrix} 1 & 2 & 3 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

calcule

(a) $A \times B$

(b) $B \times A$

Exercício 3. Considerando as matrizes do exercício 2, prove que $ACD = BCD$.

Exercício 4. Dadas as matrizes:

$$A = \begin{pmatrix} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad B = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix}$$

calcule a multiplicação booleana $A \wedge B$, a soma booleana $A \vee B$ e o produto booleano $A \times B$.

Exercício 5. Considerando as matrizes do exercício 4, verifique se o produto booleano $A \times B$ é igual à multiplicação das matrizes $A \cdot B$.

Exercício 6. Existem duas matrizes M e N tal que $M \vee N = M \wedge N$? Se existirem, quando isso ocorre?

Exercício 7. Dada a matriz esparsa:

$$E = \begin{pmatrix} 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 3 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 5 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -3 \end{pmatrix}$$

(a) ache a matriz compacta;

(b) calcule a porcentagem de compactação obtida por este processo; e

(c) usando a matriz compacta, obtenha a matriz transposta da matriz E.

Exercício 8. Dada a matriz:

$$M = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$$

calcule a matriz M^n para $n \geq 1 \in \mathbb{Z}$.

C.2 Lista de Exercícios - Listas

Exercício 1. Descreva, em Haskell, uma lista contendo 2^n para $n = 0, \ldots, 20$, com base em:

$$\lambda x.2^x | x \leftarrow \{0, 1, \ldots, 20\}$$

Exercício 2. Escreva por extenso as listas:

a. `[1..6]`

b. `[2,4..12]`

c. `[2,4..15]`

d. `[3,7..31]`

Exercício 3. Escreva, separadamente, o x e os xs das listas:

a. `[1,2,3,4]`

b. `[25,2,300,151]`

c. `[‘b’,‘k’,‘n’,‘t’]`

d. `["verde", "azul","amarelo", ]`

Exercício 4. Implemente um algoritmo que utilize o “crivo” de Erastostenes para encontrar todos os números primos entre 25000 e 26000.

Exercício 5. Usando o algoritmo da definição dos números primos (ver roteiro da aula de listas na Prática 3) e o crivo de Erastóstenes, trace um gráfico comparativo da velocidade de processamento dos dados nos dois casos.

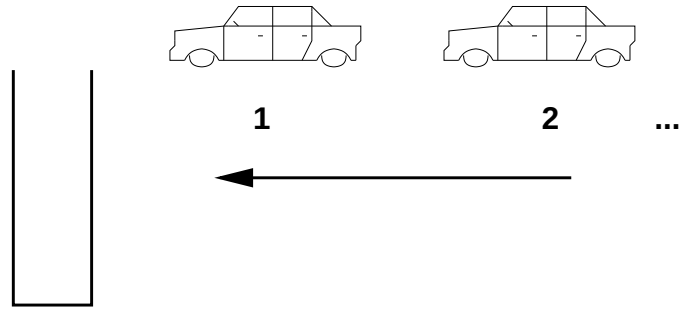
C.3 Lista de Exercícios - Listas, Pilhas e Filas

Exercício 1. Triplas de números positivos inteiros (a, b, c), tal que $a^2+b^2 = c^2$ são denominadas "triplas pitagóricas" por causa do Teorema de Pitágoras. Pode-se implementar um programa em Haskell que defina uma lista infinita destas triplas da seguinte forma:

```
triplas :: [(Integer,Integer,Integer)]
triplas = [(a,b,c) | a <- [1..], b <- [1..], c <- [1..], a^2 + b^2 == c^2]
```

Explique porque esta definição não é muito útil e construa uma definição melhor e mais útil.

Exercício 2. Considere uma garagem, com apenas uma entrada e dois manobristas, na qual os carros fiquem dispostos um atrás do outro:



Considere ainda, 4 carros, numerados de 1 a 4, chegando em ordem, mas em tempos diferentes. Quais as manobras necessárias para que a saída seja em todas as combinações de ordem possíveis? Existe alguma ordem de saída que não seja possível?

Exercício 3. Desenhe um algoritmo para o problema do exercício anterior.

Exercício 4. Considerando o exemplo de filas estáticas:

```
typedef struct Fila FL;

  struct Fila{
    int ini;
    int fim;
    int fila[10];
  };

  void Insere_Elemento(FL *F, int A){
      int aux;
    aux = (F->fim+1)%10;
    if (aux != F->ini){
      F->fim = aux;
      F->fila[aux] = A;
      }
      else
        printf(" Fila Cheia\n");
  }
```

implemente uma função para remover um elemento da fila.

Exercício 5. Implemente as funções PUSH e POP, considerando:

1. uma pilha dinâmica;
2. uma fila estática circular; e
3. uma fila dinâmica

C.4 Lista de Exercícios - Grafos

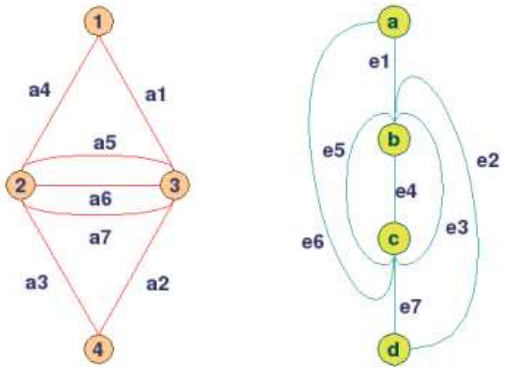
Exercício 1. Construa um grafo com os nós $N = 1, 2, 3, 4, 5$, arcos $A = a_1, a_2, a_3, a_4, a_6$ e a função g dada por:

$$\begin{array}{ll} g_1 = & 1-2 \\ g_3 = & 3-4 \\ g_5 = & 4-5 \end{array} \qquad \begin{array}{ll} g_2 = & 1-3 \\ g_4 = & 3-4 \\ g_6 = & 5-5 \end{array}$$

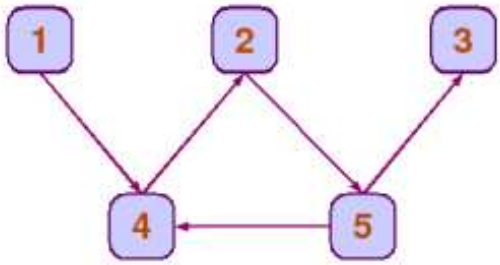
Exercício 2. Desenhe o grafo representado pela matriz adjacência K:

$$K = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Exercício 3. Nos grafos abaixo, verifique se são isomorfos. Se forem, escreva as funções que representem o isomorfismo. Caso não sejam, justifique.



Exercício 4. Dado o grafo:



ache a sua matriz adjacência e sua matriz de acessibilidade.

C.5 Lista de Exercícios - Árvores

Exercício 1. Esboce as seguintes estruturas em árvores:

1. Uma árvore com 9 nós e nível 2.
2. Uma árvore binária completa de nível 3.
3. Uma árvore de 3 níveis onde cada nó do nível i tem i+1 filhos.

Exercício 2. Desenhe uma árvore que represente a expressão algébrica:

$$k \times [(2x - 3y) + 4z]$$

Exercício 3. Desenhe a árvore que tem o percurso em pré-ordem: a,b,c,d,e; e cujo percurso em ordem simétrica é: b,a,d,c,e.

Exercício 4. Encontre uma árvore, cujos percursos em ordem simétrica e em pós-ordem fornecem a mesma lista de nós.

C.6 Lista de Exercícios - Máquinas de estado finito

Exercício 1. Considere uma máquina de estado finito $S = \{s_0, \ldots, s_4\}$ com entradas $I = \{0, 1\}$ e saídas $O = \{0, 1\}$, descrita pelas funções:

Funções de Estado	**Funções de Saída**
$f_s : (s_0, 0) \rightarrow s_0$ $f_s : (s_0, 1) \rightarrow s_3$	$f_o : s_0 \rightarrow 0$
$f_s : (s_1, 0) \rightarrow s_0$ $f_s : (s_1, 1) \rightarrow s_4$	$f_o : s_1 \rightarrow 1$
$f_s : (s_2, 0) \rightarrow s_2$ $f_s : (s_2, 1) \rightarrow s_1$	$f_o : s_2 \rightarrow 1$
$f_s : (s_3, 0) \rightarrow s_0$ $f_s : (s_3, 1) \rightarrow s_4$	$f_o : s_3 \rightarrow 0$
$f_s : (s_4, 0) \rightarrow s_1$ $f_s : (s_4, 1) \rightarrow s_2$	$f_o : s_4 \rightarrow 1$

Considere a entradas abaixo e calcule as saídas:

1. 011011010
2. 010001101
3. 001001001

Exercício 2. Uma máquina de estado finito $S = \{s0, s1, s2, s3\}$ pode ser descrita pelas funções de saída:

$$\begin{aligned} f_o : s_0 &\rightarrow a \\ f_o : s_1 &\rightarrow b \\ f_o : s_2 &\rightarrow a \\ f_o : s_3 &\rightarrow c \end{aligned}$$

As funções de estado desta máquina são controladas pelo conjunto de entradas $I = \{a, b, c\}$, tal que:

$$\begin{array}{lll} f_s : (s_0, a) \rightarrow s_2 & f_s : (s_0, b) \rightarrow s_0 & f_s : (s_0, c) \rightarrow s_3 \\ f_s : (s_1, a) \rightarrow s_0 & f_s : (s_1, b) \rightarrow s_2 & f_s : (s_1, c) \rightarrow s_3 \\ f_s : (s_2, a) \rightarrow s_2 & f_s : (s_2, b) \rightarrow s_0 & f_s : (s_2, c) \rightarrow s_1 \\ f_s : (s_3, a) \rightarrow s_1 & f_s : (s_3, b) \rightarrow s_2 & f_s : (s_3, c) \rightarrow s_0 \end{array}$$

Considerando a entradas abaixo e calcule a saída para esta máquina de estado finito.

1. abccaab
2. abacaba
3. abbacca
4. ababaca

Exercício 3. Considere uma máquina de estado finito $S = s0, ..., s3$ com entradas $I = 0, 1$ e saídas $O = 00, 01, 10, 11$, descrita pelas funções:

$$\begin{array}{lll} f_s(s_0, 0) \rightarrow s_1 & f_s(s_0, 1) \rightarrow s_0 & f_o(s_0) \rightarrow 00 \\ f_s(s_1, 0) \rightarrow s_2 & f_s(s_1, 1) \rightarrow s_0 & f_o(s_1) \rightarrow 01 \\ f_s(s_2, 0) \rightarrow s_3 & f_s(s_2, 1) \rightarrow s_0 & f_o(s_2) \rightarrow 10 \\ f_s(s_3, 0) \rightarrow s_0 & f_s(s_3, 1) \rightarrow s_0 & f_o(s_0) \rightarrow 11 \end{array}$$

Faça um grafo que represente esta máquina. Você poderia imaginar para que ela serve?

Exercício 4. Um sinal de tráfego controla um cruzamento tem dois estados: LO (Leste-Oeste) e NS (Norte-Sul). As saídas são: Vm, Am, Vd (vermelho, amarelo e verde). As regras são as regras comuns de uma sinaleira de trânsito. Experimente desenhar o grafo de uma máquina de estado que modele as funções deste sinal de trânsito.

C.7 Lista de Exercícios - Máquinas de Turing

Exercício 1. Considere uma máquina de Turing descrita pelas quíntuplas:

$$(0,0,0,1,D)(0,1,0,0,D)(0,b,b,0,D)(1,0,1,0,D)(1,1,1,0,E)$$

Descreva a última fita se a fita inicial é:

...	b	1	0	b	...

Exercício 2. Descreva o comportamento da máquina do exercício 1, se a fita for:

...	b	0	1	b	...

Exercício 3. Projete uma máquina de Turing que troque, em uma fita contendo qualquer conjunto de $0, 1$, todos os símbolos 0 por 1 e vice-versa.

Exercício 4. Encontre uma máquina de Turing para calcular a função:

$$f(x) = \begin{cases} 1 & n = 0 \\ 2 & n \neq 0 \end{cases}$$

C.8 Lista de Exercícios - Ordenação e Busca

Exercício 1. Uma moeda, entre cinco, é falsa e é mais pesada ou mais leve que as outras. O problema é identificar a moeda falsa e determinar se ela é mais pesada ou mais leve que as outras.

a. Encontre a cota mínima para o número de comparações necessárias para resolver o problema;

b. Descreva o algoritmo para a cota mínima (desenhe a árvore).

Exercício 2. Dada a lista abaixo, use uma árvore binária para ordená-la.
{35,23,67,12,5,84,25,56,2,6,44,89,32,57,15,18,29}

Bibliografia

[1] Abramsky, S. "The lazy lambda-calculus." *In*: Turner, D. A. (ed.), **Research Topics in Functional Programming**, Year of Programming series, Addison-Wesley, 1990.

[2] Backus, J. "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs", *Communications of the ACM*, 21, 613-641, 1978.

[3] Barwise, J. "Mathematical proofs of computer correctness", *Notices of the American Mathematical Society*, 7, 844-851, 1989.

[4] Church, A. "An Unsolvable Problem of Elementary Number Theory", *Am. J. Math.*, 58: 345–363, 1936.

[5] Curry, H. B.; Feys, R.; e Craig, W. "Combinatory Logic", *in:* **Studies in Logic and the Foundations of Mathematics**, Vol. I, Amsterdan: North-Holland, 1958.

[6] Gersting, J. L. **Fundamentos Matemáticos para a Ciência da Computação**, LTC Editora: Rio de Janeiro, 4a Ed., cap. 8, p. 420, exemplo 25, 2001.

[7] Hudak, P. "Conception, evolution, and application of functional programming languages", *ACM Computing Surveys*, 21, 359-411, 1989.

[8] Hudak, P. e Fasel, J. H. "A Gentle Introduction to Haskell", *Tech. Ret,*, Department of Computer Science, Yale University, 1992 [`http://cs-www.cs.yale.edu/homes/hudak-paul/`]

[9] Kernighan, B., Ritchie, D. **The C Programming Language**, Prentice-Hall: Englewood Cliffs, NJ, 1978. 2nd. Ed., 1988.

[10] MacLennan, B. J. **Functional Programming — Practice and Theory**, Reading, Massachusetts: Addison-Wesley Pub. Co., 1990.

[11] Paulson, L. C. "A higher-order implementation of rewriting", *Science of Computer Programming*, 3, 119-149, 1983.

[12] Raphael, B. "The structure of programming languages", *Communications of the ACM*, 9, 155-156, 1966.

[13] Ritchie, D. "The Development of the C Language", in: **History of Programming Languages**, 2nd Ed., Thomas J. Bergin, Jr. and Richard G. Gibson, Jr., ACM Press: NY and Addison-Wesley: Reading, Mass., (reprint), 1996.

[14] Schönfinkel, M. "Über die Bausteine der mathematischen Logik", *Mathematische Annalen*, 92: 305–316, 1924.

[15] Thompson, S. **The Craft of Functional Programming**, Addison-Wesley: England, 2nd. Ed., 1999.