Introdução à Computação

Rosane Minghim e Guilherme P. Telles

9 de Agosto de 2012

Capítulo 2

Algoritmos e Elementos Básicos de Linguagem

Neste capítulo introduzimos o conceito de algoritmo. Os algoritmos são usados para ajudar a aprender os conceitos básicos relacionados às linguagens de programação e para ajudar a transformar idéias em programas que serão executados em computadores.

2.1 Algoritmo

Um **algoritmo** é uma receita[1]. A palavra normalmente é usada em matemática e em computação para nomear um procedimento de cálculo, um cômputo. Ele dá instruções para a execução de uma tarefa, como no exemplo abaixo.

Exemplo 2.1 *Bolinhos de Chuva*

Em uma tigela, bata o açúcar, a manteiga e o ovo.

Em outro recipiente misture a farinha, o fermento, a canela, uma pitada de sal, o leite e a outra mistura. Misture bem.

Aqueça óleo e pingue colheradas da massa, fritando os bolinhos até dourar. Escorra bem, polvilhe açúcar e sirva.

Um algoritmo tem uma **entrada**, que são informações ou materiais que ele necessita para executar sua tarefa, e produz uma **saída**, que é o resultado

[1]Formalmente a palavra algoritmo significa a descrição de uma Máquina de Turing, que é um modelo que define o que é "realizar um cômputo". Mais informalmente, a palavra algoritmo é usada significando um programa de computador ou a descrição de um programa de computador. Nesta seção nós a utilizamos com um significado quase coloquial, apenas com finalidade didática. Isso torna a expressao "algoritmos computacionais" estranha, mas acreditamos que colabora com a intuição dos iniciantes.

do algoritmo. No caso do algoritmo para fazer bolinhos de chuva, a entrada é a quantidade dos ingredientes e a saída são os bolinhos prontos.

Um algoritmo pode ser expresso em diferentes níveis de detalhe. Nosso algoritmo para bolinhos de chuva poderia ser descrito em menos detalhes, como no exemplo abaixo.

Exemplo 2.2 *Bolinhos de Chuva*

Faça uma massa da maneira tradicional com o açúcar, a manteiga, o ovo a farinha, o fermento, a canela e uma pitada de sal. Frite os bolinhos e polvilhe com açúcar.

Por outro lado, nosso algoritmo para bolinhos de chuva poderia ser descrito em mais detalhes, como abaixo.

Exemplo 2.3 *Bolinhos de Chuva*

Em uma tigela, bata o açúcar e a manteiga. Depois, adicione um ovo.

Em outro recipiente, peneire a farinha e adicione o fermento, a canela e uma pitada de sal. Vá juntando essa mistura da farinha à outra mistura, alternando com leite. Misture bem.

Em uma panela, aqueça uns 5 centímetros de óleo. Quando o óleo estiver quente, pingue colheradas da massa e frite os bolinhos até dourar. Escorra bem, polvilhe açúcar e sirva.

O nível de detalhamento de um algoritmo depende do objetivo da receita. Se queremos dar apenas as idéias gerais do que um algoritmo faz, podemos dar poucos detalhes, como no Exemplo 2.2. Se queremos detalhar os passos, podemos optar por uma descrição completa, como no Exemplo 2.3. O executor do algoritmo também é importante. No caso dos bolinhos de chuva cozinheiros experientes provavelmente precisarão de menos instruções.

Várias das ações e tarefas que realizamos corriqueiramente podem ser descritas por um algoritmo. E podemos usar mais ou menos detalhes em tais algoritmos. A seguir mostramos exemplos de algoritmos para uma viagem intermunicipal de ônibus, primeiro em menos detalhes, depois em mais detalhes.

Exemplo 2.4 *Algoritmo para uma viagem de ônibus.*

Vá até a rodoviária.
Informe-se sobre os horários e preços de passagens.
Vá até o guichê de sua escolha e compre a passagem.
Preencha o formulário de identificação.
Aloje-se no ônibus.
Aproveite a viagem.

Fim da viagem.

Exemplo 2.5 *Algoritmo para uma viagem de ônibus.*
Vá até a rodoviária.
Verifique as empresas de ônibus que vendem a passagem.
Para cada empresa que vende a passagem desejada,
    informe-se sobre os horários e preços da passagem.
Vá até o guichê de sua escolha.
Solicite a passagem.
Verifique os dados da passagem e faça o pagamento.
Dirija-se a uma área de espera.
Preencha o formulário de identificação.
Espere a chegada do ônibus.
Quando o ônibus chegar,
    apresente sua passagem e formulário de identificação.
Se tiver bagagem,
    coloque sua bagagem no bagageiro e receba os comprovantes.
Aloje-se no ônibus e espere a partida.
Aproveite a viagem.
Quando chegar ao destino
    Desça do ônibus.
    Se tiver bagagem,
        retire sua bagagem do bagageiro.
Fim da viagem.

O Exemplo 2.4 mostra os passos gerais para uma viagem intermunicipal de ônibus, enquanto o Exemplo 2.5 detalha os passos individualmente. Podemos notar que cada um dos passos no Exemplo 2.4 pôde ser desenvolvido em mais detalhes com pouca interferência nos demais passos.

Algoritmos computacionais

Quando nos voltamos para os computadores, os algoritmos passam a ser receitas que queremos que o computador execute. Mas para que um computador execute uma tarefa é necessário usar uma linguagem de programação para construir um programa executável. Isso faz com que tenhamos que transformar a nossa **idéia** da tarefa que deve ser realizada em um **programa**.

Este livro é uma ajuda para quem quer aprender a programar computadores usando linguagens de estrutura semelhante àquela da linguagem Pascal. Para isso, vai ser necessário aprender a transformar idéias em programas respeitando algumas limitações. Dentre elas podemos citar:

- As operações que um computador é capaz de realizar são limitadas a um pequeno conjunto.
- A forma de escrever um algoritmo, isto é, sua sintaxe, deve seguir um certo padrão bem definido.
- A entrada para o algoritmo e os dados que ele manipula devem ser bem especificados.

Pascal é uma linguagem estruturada, assim como C, Modula 2, Perl e outras. Então ao invés de estudar Pascal ou outra linguagem diretamente, vamos definir uma linguagem padrão para construir algoritmos computacionais chamada de **pseudo-código** e usar esse pseudo-código para apresentar os conceitos comuns às linguagens estruturadas. Teremos as seguintes vantagens com esse método:

- Poderemos usar uma sintaxe mais flexível que a de uma linguagem de programação real, o que permitirá que pensemos nos passos que o algoritmo computacional deve completar sem nos preocuparmos demais com a forma de escrevê-los. A ênfase, então, será maior nas **idéias**, e não nos detalhes, que serão relevantes apenas para a linguagem de programação.
- Poderemos construir um programa em uma linguagem estruturada com facilidade se tivermos um algoritmo em pseudo-código estruturado adequadamente, porque os elementos do pseudo-código são os mesmos das linguagens estruturadas. Isto é, depois de desenvolver as idéias, a tradução para linguagem de programação será um processo simples e mecânico.

Nas próximas seções vamos apresentar os elementos que formam nosso pseudo-código e introduzir conceitos relacionados a linguagens de programação estruturadas: como os dados são armazenados, manipulados e transformados por um algoritmo computacional, quais tipos de comandos podem ser usados em algoritmos e quais as formas de estruturar algoritmos. São esses conceitos que permitirão a construção de algoritmos que serão "compreendidos" por um computador e que permitirão adquirir proficiência em linguagens como Pascal, C, Modula2 e outras.

Assim como uma receita culinária, um algoritmo computacional pode ser expresso em níveis de detalhes diferentes. O que influencia a escolha do nível de detalhes é:

1. A decisão (tomada por quem faz o algoritmo) de delinear apenas as idéias principais e grandes tarefas que o programa deve realizar ou de fornecer detalhes sobre o programa que deve ser construído.

2. A classe da linguagem de programação em que o programa será construído.

Para exemplificar o processo de construção de um algoritmo computacional, finalizamos esta seção com um exemplo de algoritmo para resolver uma equação do segundo grau da forma $ax^2 + bx + c = 0$. Vamos apresentar o algoritmo em dois níveis de detalhes diferentes: primeiramente em menos detalhes, delineando os passos do programa, e depois usando pseudo-código, já bem próximo de um programa Pascal.

Exemplo 2.6

```
Algoritmo Raízes
  Sejam a, b e c os coeficientes da equação do segundo grau
  Calcule delta
  Se delta for negativo, imprima a mensagem "não há raízes reais"
  Se delta for positivo, calcule as raízes e imprima
fim
```

Exemplo 2.7

```
Algoritmo Raízes
  {Algoritmo para calcular as raízes reais de uma equação do
  segundo grau}

  variável
    a,b,c: real
    delta: real
    x1,x2: real


  leia(a,b,c)
  delta ←   b*b - 4*a*c

  se delta < 0 então
    escreva('Esta equação não possui raízes reais.')
  senão
    x1 ← (-1*b - raiz(delta,2)) / 2*a
```

```
    x2 ← (-1*b + raiz(delta,2)) / 2*a
    escreva(x1,x2)
  fim se
fim
```

A partir deste ponto, toda vez que a palavra algoritmo for utilizada, estaremos nos referindo a algoritmos computacionais.

2.2 Elementos Básicos de um Algoritmo

Um algoritmo deve expressar os principais elementos de um programa. Os elementos típicos de um programa são dados (representados como constantes e variáveis), tipos de dados, operadores, comandos, funções e comentários. Tais elementos são usados para definir dados, estruturar o código do programa e realizar operações sobre os dados. A partir deste ponto, vamos introduzir tais elementos, os conceitos relacionados a eles e a forma de expressá-los em pseudo-código.

2.2.1 Algoritmo

Um algoritmo em pseudo-código começa com a palavra `Algoritmo` e termina com a palavra `fim`. Entre essas duas palavras há duas seções que contém todas as operações feitas pelo algoritmo para definir e manipular dados, como mostramos abaixo.

```
Algoritmo identificador
  definições e declarações
  comandos
fim
```

Na seção **definições e declarações** são definidos tipos, constantes e variáveis que definem e armazenam os dados usados pelo algoritmo. A seção **comandos** contém as expressões que manipulam os dados expressos por constates e armazenados em variáveis.

No que diz respeito aos algoritmos, não há distinção entre *definição* e *declaração* (embora esta distinção seja importante para algumas linguagens, como C). Apesar de significarem a mesma coisa, por hábito costumamos dizer *definição de tipo*, *definição de constante* e *declaração de variável*.

Um algoritmo é processado, isto é, interpretado por quem o lê, começando na palavra `Algoritmo` e prosseguindo seqüencialmente linha-a-linha, fazendo

a ação especificada pela linha até alcançar a palavra `fim`. Quando o fim é alcançado, todas as declarações feitas pelo algoritmo perdem o efeito, restando apenas o resultado das ações executadas.

2.2.2 Constantes Literais

Uma **constante** é um dado que aparece literalmente em um algoritmo. Números, valores lógicos, letras, palavras e frases podem ser expressos como constantes em um algoritmo. No exemplo abaixo, onde mostramos algumas constantes da forma como poderão aparecer em pseudo-código, usamos aspas simples para delimitar caracteres, palavras e frases.

Exemplo 2.8

```
6,45
'h'
21
'segunda-feira'
0
'domingo é bom.'
```

2.2.3 Identificadores

Vários elementos de um algoritmo podem ser identificados através de um nome. Este nome é chamado de **identificador**.

Em pseudo-código um identificador é uma única palavra com qualquer número de letras, letras acentuadas, dígitos e símbolos que não sejam operadores. As palavras abaixo são exemplos de identificadores:

```
nome
idade1
preço
preço_de_fábrica
kW
```

Os operadores têm sentido por si mesmo, por isso não devem ser usados em identificadores:

```
hora*
dia/mês
>preço
ingresso+barato
data-importante
```

Em pseudo-código não há diferenciação entre maiúsculas e minúsculas. Por exemplo, `nome`, `Nome` e `NOME` são o mesmo identificador.

2.2.4 Dados e Tipos de Dados

Um **dado** é uma informação que um algoritmo recebe ou manipula. Exemplos de dados são nomes, datas, valores (preços, notas, etc.) e condições (verdadeiro e falso).

Todo dado é de um certo **tipo** que define sua natureza (p. ex., um nome é diferente de um valor), identificando seu uso, e define as operações que podem ser realizadas com o dado (por exemplo, podemos somar dois valores numéricos mas não podemos somar um número e uma frase).

Essencialmente um algoritmo manipula dados e para isso precisa representá-los de alguma forma. Isso é feito definindo um conjunto de tipos básicos de dados que poderão ser usados nos algoritmos.

Os tipos de dados mais básicos em algoritmos são o caracter, o numérico, o lógico e a enumeração. Tipos de dados básicos podem ser estruturados em tipos mais complexos, como veremos em capítulos posteriores. Este é, dentre outros, o caso das palavras e frases, que são construídas a partir de caracteres.

Tipo de Dados Numéricos

Dados que manipulam valores numéricos podem ser de dois tipos:

- **Inteiro**: representa um número inteiro. Por exemplo -1, 0, 1, e 26 são dados inteiros. Dados deste tipo podem ser usados para idade em anos, número de filhos etc.
- **Ponto flutuante**: também chamado **real**, representa um número real. Por exemplo 1,2; 0,0; 26,4 e -2,49 são dados reais. Dados deste tipo podem ser usados para saldo bancário, altura, peso, temperatura etc.

No projeto de um algoritmo devemos utilizar o tipo numérico mais adequado, ou seja, não devemos usar um número real quando um número inteiro resolve o problema.

Tipo de Dados Caracter

Dados que representam valores alfanuméricos unitários são do tipo **caracter**. Por exemplo, 'A', 'a', '*'. Caracteres podem ser usados para a codificação de algum ítem, como sexo ('m', 'f'), estado civil ('s','c','d','v') etc. Valores

alfanuméricos incluem letras, algarismos e símbolos. Por exemplo, '1' é um caracter se consideramos apenas o símbolo '1' e não o valor 1.

Tipo de Dados Lógico

Dados lógicos podem assumir apenas dois valores: verdadeiro ou falso. Eles são usados em situações onde é necessário expressar uma condição, por exemplo, o fato de que $4 > 5$ é falso ou se o cheque número 00425 já foi compensado ou não.

Tipo de Dados Enumeração

Um dado que pode assumir um valor dentre os valores de um conjunto é uma **enumeração** ou **tipo enumerado**. Por exemplo, um dado que pode assumir qualquer valor dentro do conjunto de frutas

{`banana, maça, pera, uva, jaca`}

é deste tipo.

2.2.5 Variáveis e Atribuição

Um dado pode ser armazenado e recuperado da memória de um computador. A posição de memória onde ele é colocado pode ser identificada através de um nome. Uma **variável** é um elemento de algoritmos que tem essa função: ela associa um nome a uma porção da memória onde um dado pode ser armazenado. A variável possui, além do nome, um tipo, responsável por definir como o dado vai ser armazenado e recuperado da memória.

Em pseudo-código as variáveis são declaradas na seção de declarações, antes da seção de comandos. Para declarar uma ou mais variáveis usamos expressões da forma:

```
variável
 identificador: tipo
 identificador: tipo
 ...
 identificador: tipo
```

Por exemplo, podemos definir uma variável para armazenar a idade de uma pessoa. Para isso, temos que escolher um nome para a variável (p.ex. idade) e temos que escolher o seu tipo (p.ex. inteiro). Os exemplos a seguir são definições de variáveis em pseudo-código:

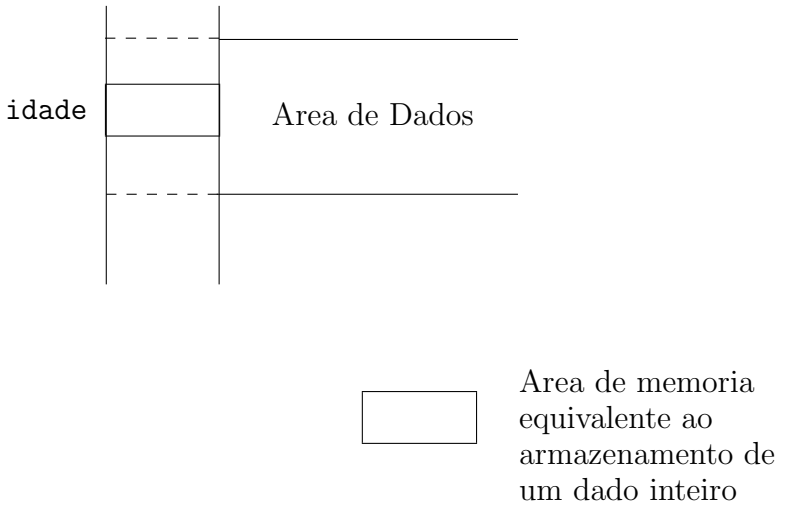

Figura 2.1: Definição da variável idade

```
variável
  idade: inteiro
  sexo: caracter
  válido: lógico
```

A Figura 2.1 ilustra a definição da variável `idade`. Quando uma variável é declarada, ela passa a referenciar uma posição de memória capaz de armazenar um valor do tipo dela.

Para utilizar uma variável para armazenar um dado na memória, utilizamos o **operador de atribuição** ← em comando da forma:

```
variável ← valor
```

No exemplo abaixo aparecem atribuições de valores de vários tipos de dados diferentes:

Exemplo 2.9

```
idade ← 51
válido ← FALSO
sexo ← 'f'
```

Esses comandos armazenam os valores `51, FALSO` e '`f`' nas posições de memória associadas às variáveis `idade, válido` e `sexo` respectivamente. A Figura 2.2 ilustra o efeito do primeiro comando do Exemplo 2.9, isto é, a atribuição do valor 51 à variável `idade`.

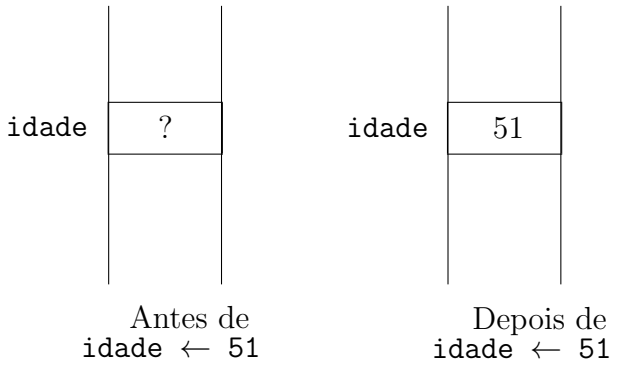

Figura 2.2: Atribuição do valor 51 à variável idade

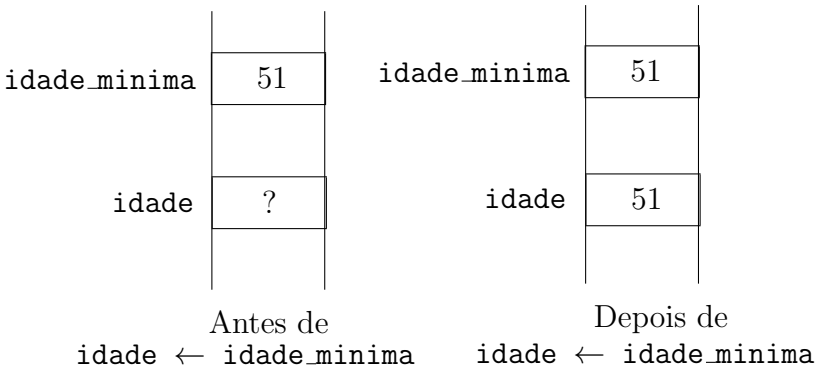

Figura 2.3: Atribuição do valor da variável idade à variável idade_mínima.

O conteúdo armazenado por uma variável pode ser copiado para outra variável do mesmo tipo usando o operador de atribuição.

Exemplo 2.10

```
variável
  idade: inteiro
  idade_mínima: inteiro

idade ← 51
idade_mínima ← idade
```

No exemplo acima, após o comando `idade_mínima ←idade`, ambas as variáveis armazenarão o valor `51`, conforme ilustrado na Figura 2.3.

O valor que está armazenado em uma variável logo que ela é declarada é desconhecido. Isso implica que se queremos garantia de que um certo valor é

armazenado em uma variável devemos usar uma atribuição para armazená-lo. O termo do jargão da programação de computadores para a primeira atribuição de valor a uma variável é **inicialização**.

O tipo de uma variável não muda durante todo o algoritmo que a utiliza. As atribuições entre variáveis podem ser feitas apenas com variáveis de mesmo tipo ou de tipo que seja compatível. Dentre os tipos definidos até o momento só existe compatibilidade entre inteiro e real, ou seja, uma variável inteira pode ser atribuída a uma variável real, mas não o contrário. Isso porque os inteiros são números reais cuja parte fracionária é zero, mas temos mais de uma opção para a parte fracionária de um real se quisermos torná-lo inteiro (arredondar para baixo ou arredondar para cima).

2.2.6 Constantes Identificadas

É possível dar nome a constantes utilizadas em um algoritmo. Isso é feito definindo um identificador para elas, na seção de declarações do algoritmo. A expressão utilizada para identificar constantes possui a forma:

```
constante
  identificador = valor
  identificador = valor
  ...
  identificador = valor
```

O tipo de uma constante é definido em função do valor atribuído a ela. No exemplo abaixo vemos definições de duas constantes, uma inteira e outra real.

Exemplo 2.11

```
constante
  idade_base = 51
  preço_máximo = 231,00
  salário_mínimo = 240,00
```

As constantes identificadas, assim como as constantes literais, podem ser atribuídas a variáveis, como no exemplo abaixo:

Exemplo 2.12

```
constante
  idade_base = 51
  preço_máximo = 231,00
```

```
variável
  inteiro: idade
  real: preço

idade ← idade_base
preço ← preço_máximo
```

O valor de uma constante não se altera após a sua definição. Em um algoritmo, constantes nunca aparecerão do lado esquerdo do comando de atribuição. As constantes literais e as identificadas se comportam essencialmente da mesma maneira, razão pela qual não faremos distinção entre elas deste ponto em diante.

2.2.7 Tipo de Dados Definidos no Algoritmo

Em um algoritmo podemos definir um novo tipo de dados a partir de tipos já existentes e dar nome a ele. Fazemos isso através de declarações da forma:

```
tipo
  identificador = expressão
  identificador = expressão
  ...
  identificador = expressão
```

onde `expressão` é ou um tipo pré-existente, uma restrição sobre o intervalo de um tipo inteiro ou caracter, ou uma composição de tipos pré-existentes. Por exemplo, podemos definir um tipo chamado `Booleano` a partir do tipo lógico:

```
tipo
  Booleano = lógico
```

Desta forma, poderemos usar o tipo Booleano como um sinônimo para lógico.

Podemos definir os tipos `dezena` e `eixo` da seguinte forma:

```
tipo
  dezena = 1 até 12
  eixo = 'x' até 'z'
```

Um dado do tipo `dezena` pode assumir valores entre 1 e 12 inclusive. Já um dado do tipo `eixo`, pode assumir valores dentre $\{$'x','y','z'$\}$

Exemplos de composições de tipos básicos para criar um novo tipo aparecerão os próximos capítulos.

2.2.8 Expressões Aritméticas e Lógicas

Em um algoritmo podemos combinar valores através da aplicação de operadores aritméticos, lógicos e relacionais formando expressões, como no exemplo abaixo.

Exemplo 2.13

```
3 + 7 * 2 - 15
verdadeiro e falso ou verdadeiro
3 + 2 < 5
```

O valor de uma expressão é calculado de imediato no algoritmo e pode ser armazenado em uma variável. As expressões também podem envolver o valor de variáveis e constantes como operandos, desde que tenham sido declaradas previamente, como na linha 11 do exemplo abaixo.

Exemplo 2.14

```
1  constante
2    fator = 0,05
3    valor = 1000,00
4    mensalista = falso
5    temporário = verdadeiro
6
7  variável
8    imposto: real
9    segurado: lógico
10
11 imposto ← valor*fator/2 + 15,00
12 imposto ← imposto * 1,05
13 segurado ← mensalista e temporário
```

Uma mesma variável pode, inclusive, aparecer antes e depois do comando de atribuição, como na linha 12 do exemplo anteiror. Nesse caso, primeiramente o valor da expressão é calculado e depois é armazenado na variável. Assim, o valor de `imposto` fica acrescido de 5% depois da execução da linha 12.

Alguns operadores usados em algoritmos são binários, isto é, realizam uma operação sobre dois operandos, como a soma $a + b$. Outros operadores são unários, isto é, realizam uma operação sobre apenas um operando, como o complemento $-a$.

Os operadores aritméticos na tabela abaixo podem ser usados em pseudocódigo.

operador	primeiro operando	segundo operando	resultado	notação
$+$	a	b	$a+b$	$a+b$
$+$	a	—	$+a$	$+a$
$-$	a	b	$a-b$	$a-b$
$-$	a	—	$-a$	$-a$
$*$	a	b	$a \times b$	$a*b$
$/$	a	b	$\frac{a}{b}$	a/b

Em uma expressão alguns operadores têm **precedência** maior que outros, isto é, os operadores de maior precedência são considerados antes em expressões. Operadores de maior precedência são considerados antes em expressões. No caso dos operadores aritméticos a precedência é maior para o multiplicação e para a divisão e menor para a subtração e para a soma, como na tabela abaixo:

	maior
$+ \ -$ unários	$\uparrow$
$* \ /$	precedência
$+ \ -$ binários	$\downarrow$
	menor

Para operadores de mesma precedência, como $+$ e $-$ binários, a ordem de execução é da esquerda para a direita. Por exemplo, consideremos a expressão

$$3 + 7 * 2 - 15$$

Ela é calculada assim:

$$\begin{gathered} 3 + 7 * 2 - 15 \\ 3 + 14 - 15 \\ 17 - 15 \\ 2 \end{gathered}$$

Pares de parênteses podem ser usados para alterar a precedência dos operadores, como por exemplo:

$$\begin{gathered} (3 + 7) * (2 - 15) \\ 10 * (2 - 15) \\ 10 * -13 \\ -130 \end{gathered}$$

Os operadores **e**, **ou** e **não** lógicos podem ser usados em algoritmos. O operador **e** é um operador binário que opera sobre valores lógicos (constantes,

variáveis ou expressões), resultando nos valores indicados abaixo:

verdadeiro	**e**	verdadeiro	=	verdadeiro
verdadeiro	**e**	falso	=	falso
falso	**e**	verdadeiro	=	falso
falso	**e**	falso	=	falso

O operador **ou** é um operador binário que opera sobre valores lógicos (constantes, variáveis ou expressões), resultando nos valores indicados abaixo:

verdadeiro	**ou**	verdadeiro	=	verdadeiro
verdadeiro	**ou**	falso	=	verdadeiro
falso	**ou**	verdadeiro	=	verdadeiro
falso	**ou**	falso	=	falso

O operador **não**, por sua vez, é um operador unário que opera sobre valores lógicos (constantes, variáveis ou expressões), resultando nos valores indicados abaixo:

não	verdadeiro	=	falso
não	falso	=	verdadeiro

A precedência dentre os operadores lógicos segue a ordem **não**, **e** e **ou**, sendo o **não** o operador de maior precedência e o **ou** o operador de menor precedência. Pares de parênteses podem ser usados também para alterar a precedência dos operadores lógicos.

Além dos operadores aritméticos e lógicos, os operadores relacionais

$$=, <, >, \leq, \geq \text{ e } \neq$$

também podem ser usados em expressões nos algoritmos. O resultado de uma expressão que contém um operador relacional é do tipo lógico, isto é, verdadeiro ou falso. Os operadores aritméticos, lógicos e relacionais podem ser combinados em uma única expressão, como no exemplo abaixo.

Exemplo 2.15

```
idade ← 28
valor ← 1000,00
fator ← 0,05

segurado: lógico

segurado ← idade < 30 e valor*fator leq 500,00
```

A expressão do Exemplo 2.15 é resolvida da seguinte forma:

idade $<$ 30 **e** valor*fator $\leq$ 500,00
idade $<$ 30 **e** 50 $\leq$ 500,00
verdadeiro **e** 50 $\leq$ 500,00
verdadeiro **e** verdadeiro
verdadeiro

Portanto, o resultado final da variável `segurado` é verdadeiro.

A precedência dos operadores que vimos até agora pode ser resumida na tabela abaixo. Acrescentamos a atribuição à tabela para enfatizar que apenas depois do calculo de toda a expressão seu valor final é guardado na variável:

$+$ $-$ unários	maior
$*$ $/$	$\uparrow$
$+$ $-$ binários	precedência
não e ou	$\downarrow$
$= < > \leq \geq \neq$	menor
$\leftarrow$	

Funções Pré-definidas Além dos operadores aritméticos, em pseudo-código há funções pré-definidas para outras operações matemáticas. O conceito de uma função em uma linguagem de programação será estudado em profundidade no Capítulo 4. Por hora, é suficiente saber que uma função é um algoritmo que recebe zero ou mais dados (chamados parâmetros), faz algumas operações e devolve um único dado. As funções aritméticas pré-definidas que usaremos em nossos algoritmos estão na tabela abaixo.

Função	Tipo dos Parâmetros	Resultado
raiz(x,n)	x: real, n: real	A n-ésima raiz de x
seno(x)	x: real	O seno de x dado em graus
cosseno(x)	x: real	O cosseno de x dado em graus
tangente(x)	x: real	A tangente de x dado em graus
exp(x)	x: real	e^x
abs(x)	x:real	o valor absoluto de x
arredonda(x)	x:real	aproxima para o inteiro mais próximo

O Exemplo 2.16 fornece alguns resultados das funções apresentadas acima.

Exemplo 2.16 *Resultados de Funções*

raiz(144,2) = *12*
seno(30) = *0,5*
cosseno(90) = *0,0*
tangente(45) = *1,0*

Funções podem ser utilizadas em expressões, ou como parâmetros de outras funções, conforme exemplificado a seguir.

Exemplo 2.17

```
valor ← raiz(a,2)*2 + n
```

No exemplo acima, o valor da raiz quadrada de `a` será multiplicado por 2, em seguida somado com o valor da variável `n` e o resultado dessa expressão será atribuído à variável `valor`.

Funções têm precedência sobre todas as demais operações vistas até agora. Os parâmetros passados para funções podem ser variáveis, constantes, expressões ou resultados de outras funções.

2.2.9 Entrada e Saída

Um algoritmo pode receber dados através de dispositivos como teclado, mouse, discos e placas de rede, e pode enviar dados para o monitor de vídeo, discos e outros. Este tipo de operações em que dados são recebidos por um algoritmo ou são enviados por um algoritmo para um dispositivo são chamados de **operações de entrada e saída**.

Nosso pseudo-código para algoritmos contém funções pré-definidas para ler dados do teclado e para escrever caracteres no monitor de vídeo. Tais funções aparecem na tabela abaixo.

Função	Parâmetros	Resultado
leia(x1,x2,...)	variáveis de qualquer tipo básico	Os valores digitados no teclado são armazenados em x1, x2, ...
escreva(y1,y2,...)	variáveis, constantes ou expressões de qualquer tipo básico	Os valores de y1, y2, ... são escritos no monitor.

Em geral as linguagens de programação também provêm funções para receber e para enviar dados para outros tipos de dispositivos. Algumas provêm funções para entrada e saída de outros tipos de dados, como imagens e som, dentre outros.

Nos algoritmos deste livro vamos usar essencialmente entrada de dados proveniente do teclado, saída para o monitor de vídeo e entrada ou saída em arquivos em disco, conforme será visto em capítulo posterior.

2.2.10 Comentários, Linhas em Branco e Indentação

Comentários, linhas em branco e indentação[2] são elementos de algoritmos que não são comandos, mas servem para documentar o algoritmo e melhorar a sua legibilidade.

Comentários são usados para descrever o algoritmo, indicar o significado de variáveis e constantes e esclarecer trechos do código. Sua função primordial é apoiar a compreensão de quem lê o programa posteriormente, que pode ser a mesma pessoa que o escreveu ou outra pessoa diferente.

Linhas em branco e indentação têm função parecida: melhorar a legibilidade do programa. A adição das linhas em branco e da identação delimita blocos de comandos do algoritmo, deixando claro quais comandos serão selecionados por uma alternativa `se`, por exemplo. Se olharmos para o algoritmo 2.7 (Página 18) veremos que é muito mais fácil compreendê-lo daquela forma do que se estivesse escrito como aparece abaixo.

Exemplo 2.18

```
Algoritmo Raízes
variável a,b,c: real
delta: real
x1,x2: real
leia(a,b,c)
delta ← b*b - 4*a*c
se delta < 0 então escreva('Esta equação não possui raízes reais.')
senão x1 ← (-1*b - raiz(delta,2)) / 2*a
x2 ← (-1*b + raiz(delta,2)) / 2*a
escreva(x1,x2)
fim se
fim
```

Em pseudo-código os comentários são delimitados por pares de chaves, como nos exemplos 2.19 e 2.20.

[2]Esta palavra vem do inglês *indent*.

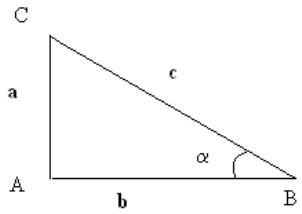

Figura 2.4: Triângulo retângulo e parâmetros

2.2.11 Exemplos de Algoritmos

A seguir apresentamos um algoritmo para calcular os dois lados de um triângulo retângulo, dado um dos seus ângulos, α, e a hipotenusa c (ver Figura 2.4). Pelas regras básicas de geometria $cosseno(\alpha) = \frac{b}{c}$ e $seno(\alpha) = \frac{a}{c}$, sendo a o lado oposto ao ângulo, e b o lado adjacente ao ângulo α. O exemplo abaixo apresenta para calcular os lados adjacente e oposto ao ângulo α.

Exemplo 2.19

```
Algoritmo lados_triângulo
  {Este algoritmo calcula o valor dos lados de um triângulo
  retângulo, dados um de seus ângulos menores e a hipotenusa}

  variável
    lado_oposto, lado_adjacente: real
    hipotenusa, alfa: real

  leia(alfa)
  leia(hipotenusa)
  lado_oposto ← seno(alfa)*hipotenusa
  lado_adjacente ← cosseno(alfa)*hipotenusa
  escreva(lado_oposto)
  escreva(lado_adjacente)
fim
```

O exemplo a seguir calcula o salário de um vendedor que possui um salário base e um acréscimo baseado em comissão.

Exemplo 2.20

```
Algoritmo salário
```

```
{Este algoritmo calcula o valor do salário de um funcionário
dados o valor total de suas vendas e sua porcentagem de comissão}

constante
  salário_base = 240,00;

variável
    salário: real
    comissão: real
    valor_vendido: real

    leia(comissão,valor_vendido)
    salário ← salário_base + comissão/100*valor_vendido
    escreva(salário)

fim
```

2.2.12 Considerações Finais

Como vimos ao longo deste capítulo, um algoritmo é uma forma de organizar as idéias com o objetivo de construir um programa.

Quando o problema a resolver é um pouco mais complexo, a pessoa que está construindo o algoritmo provavelmente terá problemas se tentar escrevê-lo ao longo de várias páginas. Provavelmente será mais fácil chegar a um boa solução se a estrutura do problema e a estrutura do próprio algoritmo forem analisadas e se o algoritmo for *quebrado*. Quando quebramos um algoritmo estamos fazendo o que chamamos de **modularização**. Quando modularizamos diminuímos a complexidade com que temos que lidar ao tentar resolver o problema original. Podemos pensar em cada subproblema separadamente e depois de resolvê-los, construir a solução para todo o problema sem pensar nos detalhes de cada subproblema. Uma boa modularização é aquela que divide um algoritmo em partes (módulos) de tal forma que: (1) cada parte resolve um subproblema bem definido e (2) uma parte não interfere na outra.

Um efeito colateral de uma boa modularização é que fica fácil reutilizar os módulos na solução de outros problemas. Algumas vezes é óbvio que um certo subproblema deve ser

Outro efeito colateral da modularização adequada é a facilidade de modificar o algoritmo quando for necessário. Se os módulos resolvem um problema bem definido e são independentes, os efeitos da modificação de um módulo

provavelmente ficarão restritos a uma parte pequena do programa. O assunto de modularização será tratado novamento num capítulo posterior.

Mas quando pensamos em problemas realmente complexos, que são resolvidos por algoritmos e sistemas de software grandes, teremos equipes de pessoas trabalhando em conjunto. Para a construção de sistemas de grande porte ser eficiente ela deve ser bem planejada. Tanto o planejamento quanto a execução da construção de tais sistemas são complexos. Para auxiliar essa tarefa há metodologias desenvolvidas pelo ramo da computação chamado de **Engenharia de Software**, que se preocupa com questões humanas, de planejamento e de qualidade relacionadas à construção de sistemas de software.

Outro aspecto da importante na construção de algoritmos é o tratamento de situações excepcionais. Uma situação excepcional é algum evento que não faz parte da operação padrão de um algoritmo. Por exemplo, vamos considerar o exemplo do algoritmo para calcular os lados de um triângulo. A operação padrão é ler os dados do teclado, calcular os lados e imprimir o resultado. Mas o que aconteceria se na digitação das entradas dados, o usuário cometesse um erro de digitação e o ângulo fosse igual a 90 graus? O algoritmo produziria um resultado errado. E no exemplo do cálculo do salário, se o valor vendido for igual a zero, então teremos uma divisão por zero que causa uma falha do programa.

Extrapolando, podemos imaginar os transtornos e riscos se as situações excepcionais não forem previstas e tratadas adequadamente nos programas de gerenciamento de contas bancárias, sinais de trânsito, metrô, tráfego aéreo... O que fazer em relação às situações excepcionais depende dos mecanismos que a linguagem de programação oferece e da forma escolhida para detectar e contornar essas situações.

Sugestão 1 *Procure sempre desenvolver um algoritmo em etapas. Entenda o problema, elabore uma solução escrevendo instruções gerais, e em seguida detalhe cada uma das instruções gerais individualmente.*

Sugestão 2 *Procure usar nomes de variáveis significativos, que expressem o significado real da variável, mesmo que eles fiquem longos.*

Sugestão 3 *Revise seu algoritmo, tentando identificar se os passos individuais são suficientemente independentes um dos outros.*

Sugestão 4 *Revise seu algoritmo em busca de possíveis erros e exceções que possam ser tratados.*