Capítulo 2

INTRODUÇÃO À CONSTRUÇÃO DE ALGORITMOS

2.1 Definição de Algoritmo

O primeiro passo para a resolução de um problema por meio de um programa de computador é a definição ***precisa*** do problema. Depois desse passo, planeja-se a solução do problema por meio da escrita de um algoritmo. Um **algoritmo** consiste de uma **sequência de passos** (instruções) que recebem alguns valores como **entrada** e produzem alguns valores como **saída**. Quando executadas, as instruções de um algoritmo resolvem um determinado problema. Além disso, essas instruções não devem ser ambíguas e todo algoritmo deve encerrar.

Uma analogia bastante comum que ajuda no entendimento do conceito de algoritmo é aquela entre algoritmo e receita culinária. Numa receita culinária, os ingredientes e utensílios utilizados (por exemplo, ovos, farinha de trigo, assadeira) compõem a entrada e o produto final (por exemplo, um bolo) é a saída. O modo de preparo especifica uma seqüência de passos que informam como processar a entrada a fim de produzir a saída desejada. Apesar de a analogia entre algoritmo e receita culinária ser válida, dificilmente uma receita pode ser considerada um algoritmo de fato, pois, tipicamente, receitas culinárias são imprecisas na especificação de ingredientes (entrada) e na descrição do modo de preparo (processamento). Além disso, receitas culinárias, na maioria das vezes, requerem inferências e tomadas de decisões por partes de quem as implementam e um algoritmo não deve requerer nenhum tipo de inferência ou tomada de decisão por parte do computador que, em última instância, o executará. Computadores simplesmente não possuem tais aptidões.

Raramente, um algoritmo é concebido com o objetivo de receber um conjunto limitado de valores. Isto é, mais comumente, um algoritmo é desenvolvido para lidar com vários **casos de entrada**. Por exemplo, um algoritmo criado para resolver equações do segundo grau pode receber como entrada a equação $x^2 - 5x + 6$ e produzir como saída as raízes 2 e 3. Esse mesmo algoritmo serviria para resolver a equação $x^2 - 4x + 4$, produzindo 2 como saída. Nesse exemplo, as referidas equações são casos de entrada do algoritmo exemplificado. Em linguagem cotidiana, um caso de entrada é referido simplesmente como **entrada**.

Um algoritmo é **correto**, quando, para cada caso de entrada, ele pára com a saída correta. Um algoritmo **incorreto** pode não parar quando ele recebe um dado caso de entrada ou pode parar apresentando um resultado que não é correto.

Pode haver vários **algoritmos funcionalmente equivalentes** que resolvem um mesmo problema. Algoritmos funcionalmente equivalentes podem usar mais ou menos recursos, ter um número maior ou menor de instruções e assim por diante. Novamente, a analogia entre algoritmo e receita culinária é válida aqui: algumas receitas requerem menos esforços e ingredientes do que outras que resultam na mesma iguaria.

É importante ressaltar ainda que nem todo problema possui algoritmo. Por exemplo, não existe algoritmo para o problema de enriquecimento financeiro (lícito ou ilícito). Esse problema não possui algoritmo porque sequer é bem definido. Mas existem problemas que, apesar de serem bem definidos, não possuem algoritmos completos como, por exemplo, jogar xadrez. É interessante notar ainda que problemas que são resolvidos trivialmente por seres humanos, como falar uma língua natural ou reconhecer um rosto, também não possuem solução algorítmica. Por outro lado, problemas relativamente difíceis para seres humanos, como multiplicar dois números inteiros com mais de dez dígitos cada, possuem algoritmos relativamente triviais.

A etapa mais difícil na escrita de um programa é o desenvolvimento de um algoritmo que resolve o problema que o programa se propõe a solucionar. Quer dizer, uma vez que um algoritmo tenha sido corretamente criado, codificá-lo é relativamente fácil, mesmo quando você não tem ainda bom conhecimento sobre a linguagem usada na codificação. Muitas vezes, encontrar um algoritmo adequado é uma tarefa difícil até mesmo para os melhores programadores. Portanto, dedique bastante atenção a este capítulo, que descreve o processo de desenvolvimento de algoritmos.

2.2 Abordagem Dividir e Conquistar

A abordagem mais comum utilizada na construção de algoritmos é denominada **dividir e conquistar**. Utilizando essa abordagem, divide-se sucessivamente um problema em subproblemas cada vez menores até que eles possam ser resolvidos trivialmente. As soluções para os subproblemas são então combinadas de modo a resultar na solução para o problema original. Esse método de resolução de problemas também é conhecido como **refinamentos sucessivos**.

A abordagem dividir e conquistar não é usada especificamente na área de programação ou computação. Ou seja, ela é genérica e freqüentemente utilizada até mesmo na resolução de problemas cotidianos. Por isso, seu uso será exemplificado a seguir por meio da resolução de um problema não computacional.

Suponha que você esteja recebendo amigos para um almoço e deseja servi-los um delicioso camarão ao molho de coco. Então, o algoritmo a ser seguido para resolver esse problema pode ser descrito como a seguir[1]:

Algoritmo Camarão ao Molho de Coco

Ingredientes e equipamentos (Entrada):

- 1kg de camarão sem casca
- 1 litro de leite de coco
- 1 cebola média
- 2 tomates sem pele e sem sementes picados
- ½ pimentão verde médio
- 1 molho de coentro amarrado
- 4 colheres de sopa de azeite
- 2 colheres de sopa de colorau
- Frigideira
- Panela
- Colher de pau ou polipropileno
- etc. (para encurtar o exemplo)

Resultado (Saída):

- Camarão ao molho de coco

Preparo (Passos ou Instruções):

1. Pique a cebola em pedaços miúdos.
2. Remova a pele do tomate e pique-o.
3. Corte o pimentão em pedaços graúdos que permitam a remoção após o cozimento.
4. Obtenha o leite de coco.
5. Faça o azeite colorado.
6. Salteie o camarão e reserve.
7. Refogue a cebola, o pimentão e o tomate.
8. Acrescente o leite de coco e o azeite colorado ao refogado.
9. Quando a mistura ferver, acrescente o camarão.
10. Deixe cozinhar em fogo baixo por cerca de 10 minutos.
11. Remova o coentro e os pedaços de pimentão.

[1] O autor agradece a nutricionista e chef Suzana Brindeiro pela receita e pelos segredos de execução que não são revelados aqui.

Como está na moda recepções na cozinha e seus amigos são íntimos, você convida-os para ajudá-lo na preparação do prato. Agora, suponha que você atribua a tarefa 1 a um de seus amigos e que ele não saiba como executá-la (i.e., a tarefa não é trivial para esse amigo). Então, você terá que especificar essa tarefa em maiores detalhes para que ele seja capaz de executá-la. Em outras palavras, você terá que **refinar** a tarefa 1 do preparo em subtarefas de tal modo que seu amigo saiba como executar cada uma delas. Em programação, a analogia correspondente a esse refinamento de tarefas é que o programador deve refinar (i.e., dividir) os passos de um algoritmo até que cada passo resultante de sucessivas divisões possa ser representado por uma única instrução numa linguagem de alto nível[2].

Prosseguindo com o último exemplo, o passo 1 do algoritmo apresentado pode ser refinado para resultar na seguinte seqüência de subpassos:

1.1 Apare a ponta da cebola.

1.2 Divida a cebola ao meio no sentido do talo.

1.3 Remova a casca da cebola, deixando o talo.

1.4 **Para** cada metade de cebola **faça** o seguinte:

- 1.4.1 Deite a face plana da metade da cebola sobre a tábua de corte.
- 1.4.2 **Se** a metade da cebola for alta, **então**, com a faca na horizontal, aplique dois cortes longitudinais desde a extremidade oposta ao talo até próximo a este, de modo a dividir a metade da cebola em três partes; **senão**, aplique apenas um corte, de modo a dividir a metade da cebola em duas partes.
- 1.4.3 Iniciando com a faca próxima à posição vertical, aplique cortes verticais à metade da cebola da extremidade oposta ao talo até próximo a este, obtendo, assim, tiras finas.
- 1.4.4 Segure a metade de cebola com quatro dedos voltados para dentro, sem incluir o polegar e de modo que o talo da cebola esteja voltado para a palma da mão.
- 1.4.5 Com a faca próxima aos dedos e em posição inclinada, **enquanto** a metade de cebola não estiver cortada em cubinhos até próximo ao talo, aplique cortes transversais.

Note que o passo 1.4.2 apresenta duas ações condicionadas ao formato da cebola (i.e., à entrada do problema) e que apenas uma dessas ações deverá ser executada. A condição do passo 1.4.2 é o fato que imediatamente segue a palavra ***se***, e as ações aparecem após

[2] Um programador com alguma experiência não precisa de tanto refinamento quanto um programador com menos experiência.

as palavras ***então*** e ***senão***. Ações **condicionais** são muito comuns em programação e linguagens de alto-nível provêm facilidades para codificação delas.

Os passos 1.4 e 1.4.5 apresentados acima envolvem ações repetitivas que possuem instruções análogas em programação. Por exemplo, o passo 1.4.5 representa uma estrutura de repetição do tipo: ***enquanto*** *uma dada condição for satisfeita execute repetidamente uma determinada ação*. No exemplo dado, a condição que deve ser satisfeita para que ocorra repetição é o fato de a cebola não estar ainda cortada até a proximidade do talo e a ação a ser repetida é o corte da cebola.

Todos os demais passos do modo de preparo podem ser refinados de acordo com a intimidade culinária de quem irá executá-los[3]. No melhor dos casos, o amigo que irá executar um dos passos é um *chef de cuisine* experiente e não precisa de maiores detalhes para executar a tarefa. Por outro lado, outro amigo que nunca ferveu uma água requer que a tarefa a ser executada seja minuciosamente detalhada. Esses fatos têm correspondência em programação: assim como o *chef*, para criar um programa, um programador experiente precisa de poucos detalhes na descrição de um algoritmo, enquanto que um programador iniciante precisa ter um algoritmo bem mais refinado em detalhes.

Antes de encerrar esta aventura culinária, deve-se notar ainda que os passos sugeridos para o corte de cebola constituem em si um algoritmo; ou, mais precisamente, um **subalgoritmo**, já que ele está subordinado a um algoritmo maior. Nesse subalgoritmo, a entrada consiste em cebola, faca e tábua de corte, e a saída é a cebola picada. Evidentemente, esse subalgoritmo pode ser incorporado sem alteração em outras receitas sempre que cebola picada se fizer necessária. Em programação, **funções** ou **procedimentos** desempenham o papel de subalgoritmos.

Neste ponto, é oportuno apresentar outra importante analogia entre culinária e programação. Suponha que um dos seus convidados padece de alergia a camarão. Então, você decide que servirá peixe ao molho de coco a esse amigo. Acontece que a receita desse prato é semelhante à receita de camarão apresentada acima, exceto pelos seguintes fatos:

- Ingredientes (entrada): em vez de camarão, deve-se usar um peixe de textura firme (e.g., dourado ou cavala).
- Resultado (saída): peixe ao molho de coco, em vez de camarão ao molho de coco.
- Preparo – os passos 5 e 8 devem ser substituídos por:

[3] De fato, até a obtenção dos ingredientes e equipamentos pode necessitar detalhamento. Por exemplo, as escolhas do camarão e do tipo de coco são críticos nessa receita.

5. Frite o peixe e reserve.

8. Quando a mistura ferver, acrescente o peixe.

Sabiamente, devido à proximidade das duas receitas, você não repetirá os passos que são comuns a elas. Isto é, a melhor maneira de resolver esse novo problema é aproveitar parte do que foi realizado no problema anterior. Nesse caso específico, levando em consideração as alterações descritas acima, todos os demais ingredientes e passos usados na confecção do molho de camarão podem ser reutilizados na criação do novo prato. Portanto, as duas melhores alternativas para o cozinheiro são: reduzir a quantidade de camarão ou aumentar a quantidade de ingredientes usados no molho de coco, para que sobre molho suficiente para ele criar a nova receita. Em qualquer dos casos, o cozinheiro estará reutilizando parte do trabalho que já foi realizado e isso tem um análogo em programação que, infelizmente, não é devidamente explorado no ensino dessa disciplina: **reuso de código**.

Um programador experiente muito raramente começa a escrever um programa a partir do zero. Ou seja, na maioria das vezes, ele sempre encontra um programa que ele já escreveu que pode ter partes reutilizadas na construção de um novo programa.

2.3 Linguagem Algorítmica

Um algoritmo pode ser escrito em qualquer linguagem, como uma língua natural (e.g., Português) ou uma linguagem de programação (e.g., C). Aliás, um programa de computador consiste exatamente de um algoritmo (ou coleção de algoritmos) escrito numa linguagem de programação. Linguagem natural pura raramente é usada na escrita de algoritmos pois ela apresenta problemas inerentes, tais como prolixidade, imprecisão, ambigüidade e dependência de contexto. O uso de uma linguagem de programação de alto nível também não é conveniente para a escrita de um algoritmo, pois o programador precisa dividir sua atenção entre essa tarefa e detalhes sobre construções da linguagem na qual o algoritmo será escrito.

O objetivo aqui é a escrita de algoritmos que, em última instância, possam tornar-se programas. Mas, a escrita de algoritmos numa linguagem de programação impõe sérias dificuldades para aqueles que ainda não adquiriram prática na construção de algoritmos nem conhecem bem a linguagem de programação utilizada. Assim, ao tentar escrever um algoritmo numa linguagem de programação, o aprendiz estaria envolvido em duas tarefas simultâneas: a resolução do problema em questão (i.e., a construção do algoritmo em si) e o uso de uma linguagem que ele ainda não domina. Uma idéia que facilita a vida do programador consiste em usar, na construção de algoritmos, uma linguagem próxima à linguagem natural do programador, mas que incorpore construções semelhantes àquelas encontradas comumente em linguagens de programação. Uma linguagem com essas características é denominada **linguagem**

algorítmica ou **pseudolinguagem**[4]. Para atender à finalidade a que se destina, uma linguagem algorítmica precisa ainda ser bem mais fácil de usar do que qualquer linguagem de programação.

Pseudocódigo é a descrição de um algoritmo escrito numa linguagem algorítmica (pseudolinguagem). Pseudocódigo é dirigido para pessoas, e não para máquinas. Pseudocódigo é usado não apenas por programadores iniciantes, mas também por programadores experientes, embora estes sejam capazes de escrever programas relativamente simples sem o auxílio de pseudocódigo.

Utilizando uma linguagem algorítmica, o desenvolvimento de um programa é dividido em duas grandes fases, cada uma das quais será detalhada mais adiante:

1. **Construção do algoritmo usando linguagem algorítmica**. Nessa fase, o programador deverá estar envolvido essencialmente com o raciocínio que norteia a escrita do algoritmo, visto que, idealmente, a linguagem usada na escrita do algoritmo não deverá impor nenhuma dificuldade para o programador. Por exemplo, se um determinado passo do algoritmo requer a exibição de um valor inteiro a ser armazenado numa variável x, o programador deve escrever:

```
escreva(x)
```

2. **Tradução do algoritmo para uma linguagem de programação**. Aqui, a preocupação do programador não deverá mais ser o raciocínio envolvido na construção do algoritmo. Ou seja, nessa fase, o programador utilizará apenas seu conhecimento sobre uma linguagem de programação para transformar um algoritmo em programa. Por exemplo, nessa fase, a instrução de escrita apresentada acima seria traduzida em C como[5]:

```
printf("%d", x);
```

As próximas seções deste capítulo descrevem uma linguagem algorítmica que leva em consideração o que foi exposto na presente seção. O leitor deve notar que não precisa seguir rigorosamente as especificações dessa linguagem, pois, conforme foi exposto, ela é uma linguagem artificial que tem como propósitos ajudá-lo na escrita de algoritmos e na posterior tradução de cada algoritmo usando uma linguagem de programação. Assim, por exemplo, você pode, se desejar, substituir a instrução de saída:

---

[4] Um recurso alternativo para o uso de pseudolinguagem são os **fluxogramas**. Essa alternativa já foi bastante utilizada mas está em desuso hoje em dia e não será estudada aqui.

[5] Note como seria bem mais complicado para o programador se ele tivesse que escrever o algoritmo em C desde o início do processo de desenvolvimento. Nesse exemplo, a instrução `escreva(x)` é bem mais simples do que `printf("%d", x)`.

```
escreva(x)
```

por:

```
imprima(x)
```

sem que haja nenhum problema. Mas, cuidado com suas personalizações da linguagem algorítmica para que elas não prejudiquem os propósitos da linguagem.

2.4 Variáveis e Atribuições

Uma **variável** (simbólica) em programação representa por meio de um nome o conteúdo de um espaço contínuo em memória (i.e., um conjunto de células vizinhas). Assim, uma variável é caracterizada por três atributos: **endereço**, **conteúdo** (ou **valor**) e **nome** (ou **identificador**), como ilustra a **Figura 1**.

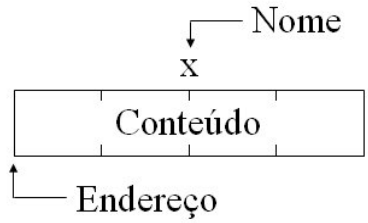


Figura 1: Uma Variável com Quatro Células

Observe que, como ilustra a **Figura 1**, quando uma variável ocupa mais de uma célula de memória, seu endereço corresponde ao endereço da primeira célula.

Em programação, o nome de uma variável representa seu conteúdo. Assim, por exemplo, quando uma variável aparece numa expressão como em:

```
y + 5
```

entende-se que é o seu valor corrente que está sendo adicionado a `5`.

Informalmente, uma **expressão simples** é uma combinação de um **operador**, que representa uma operação a ser efetuada, e **operandos** sobre os quais a operação atua. Um operando pode ser representado por um valor constante ou uma variável. Considerando o último exemplo, o operador é representado pelo símbolo "+" e os operandos são `y` (uma variável) e `5` (uma constante).

Numa expressão mais complexa, os operandos de um operador podem ser também expressões. Por exemplo, na expressão:

```
x + y * 2
```

os operandos do operador "+" são a variável `x` e a expressão `y * 2`. Na **Seção 2.5**, expressões e operadores serão explorados em maiores detalhes.

Uma instrução de **atribuição** representa o ato de uma variável receber o valor de uma constante, o conteúdo de outra variável ou o resultado da avaliação de uma expressão. Uma atribuição em linguagem algorítmica será representada pelo símbolo ←, com a variável que sofre a atribuição à esquerda da seta e o valor atribuído a ela (representado por uma constante, variável ou expressão) à direita. Por exemplo, se for desejado expressar a idéia de que uma variável `x` recebe o valor resultante da soma `y + 2`, onde `y` é outra variável, escreve-se em linguagem algorítmica:

```
x ← y + 2
```

Essa instrução de atribuição lê-se como: *x recebe o valor de y mais dois*.

Considere, agora, o seguinte exemplo de atribuição:

```
x ← x + 2
```

Nesse caso, a variável `x` recebe o valor que tinha ***antes***[6] de a atribuição ocorrer acrescido de `2`.

A maioria das linguagens de programação de alto nível requer que qualquer variável seja declarada antes de seu primeiro uso. **Declarar** uma variável significa simplesmente informar qual é o seu tipo. Isto é realizado precedendo-se o nome da variável com seu respectivo tipo.

Na linguagem algorítmica apresentada aqui, são utilizados três tipos:

- **inteiro** (v. **Seção 2.5.2**)
- **real** (v. **Seção 2.5.2**)
- **booleano** (v. **Seção 2.5.4**)

[6] Isso ocorre porque, numa atribuição na qual uma variável recebe o valor resultante da avaliação de uma expressão, a expressão deve ser avaliada *antes* de a atribuição ocorrer.

Quando o tipo de uma variável não é facilmente deduzido do contexto, é recomendável incluir sua declaração no início do algoritmo no qual a variável é usada. Declarar variáveis num algoritmo também é vantajoso durante a tradução do algoritmo em programa porque evita que o programador esqueça de fazê-lo.

Exemplos de declarações de variáveis num algoritmo:

```
booleano b
inteiro  x
real     y, z
```

Observe que, quando duas ou mais variáveis são de um mesmo tipo, como as variáveis `y` e `z` do exemplo anterior, elas podem ser declaradas de modo resumido separando-as por vírgulas e precedendo-as pelo nome do tipo comum.

2.5 Operadores e Expressões

Existem três tipos básicos de expressões em programação:

- **Expressões aritméticas**. Os operadores representam operações aritméticas usuais, e os operandos e resultados de suas avaliações são valores numéricos inteiros ou reais.
- **Expressões relacionais**. Os operadores representam operações de comparação entre valores numéricos e que resultam num valor **verdadeiro** ou **falso**. Esses valores são denominados **constantes** ou **valores lógicos** ou **booleanos**.
- **Expressões lógicas**. Os operadores representam conectivos lógicos. Os operandos são constantes, variáveis ou expressões lógicas. O resultado de uma expressão lógica é uma constante lógica.

2.5.1 Propriedades de Operadores

Antes de explorar em detalhes os três tipos de expressão mencionados acima, serão descritas propriedades que são comuns a todos os operadores.

Aridade

A **aridade** de um operador é o número de operandos que o operador admite. Em linguagem algorítmica, os operadores são divididos em duas categorias de aridade:

- **Operadores unários** são operadores de aridade um (i.e., eles requerem apenas um operando).
- **Operadores binários** são operadores de aridade dois (i.e., que requerem dois operandos)

Por exemplo, o operador de soma é um operador binário (i.e., ele possui aridade dois).

Resultado

Todo operador, quando aplicado a seus operandos, resulta num valor. Esse valor é o **resultado** do operador. Por exemplo, o resultado do operador de soma é o valor obtido quando seus dois operandos são somados.

Precedência

A **precedência** de um operador determina a ordem relativa com que ele é aplicado numa expressão contendo operadores considerados distintos no que diz respeito a essa propriedade. Ou seja, quando numa expressão, um operador tem maior precedência que outro, o operador de maior precedência é aplicado antes do operador de menor precedência.

Operadores são agrupados em **grupos de precedência**, de tal modo que, em cada grupo de precedência, os operadores têm a mesma precedência. Por outro lado, operadores que fazem parte de grupos de precedência diferentes possuem precedências diferentes. Por exemplo, os operadores de soma e subtração fazem parte de um mesmo grupo de precedência e o mesmo ocorre com os operadores de multiplicação e divisão. Mas, o grupo de precedência que contém multiplicação e divisão possui precedência maior do que o grupo de precedência de soma e subtração. Assim, na expressão:

```
2*5 + 4
```

o operador de multiplicação ("*****") é aplicado antes do operador de soma ("**+**").

Associatividade

Assim como a propriedade de precedência, **associatividade** é usada para decidir a ordem de aplicação de operadores numa expressão. Mas, enquanto precedência é usada com operadores de precedências diferentes, a associatividade é utilizada com operadores de mesma precedência ou com ocorrências de um mesmo operador. Existem dois tipos de associatividade:

- **Associatividade à esquerda** – o operador da esquerda é aplicado antes do operador da direita.
- **Associatividade à direita** – o operador da direita é aplicado antes do operador da esquerda.

Por exemplo, na expressão `8/2/2` o primeiro operador de divisão é aplicado antes do segundo, pois o operador de divisão têm associatividade à esquerda. Nesse caso, o resultado da expressão é `2`. (Se o operador de divisão tivesse associatividade à direita o resultado seria `8`.)

2.5.2 Operadores e Expressões Aritméticos

Os **operadores aritméticos** usados em programação correspondem às operações usuais em matemática (por exemplo, soma, subtração, multiplicação, etc.). Entretanto, nem sempre eles utilizam a mesma notação vista em matemática. Por exemplo, em matemática, o operador de multiplicação pode ser representado por um ponto (por exemplo, `a.b`) ou pela simples justaposição de operandos (por exemplo, `ab`), enquanto que em programação, esse operador é usualmente representado por "*****" (asterisco).

Os operadores aritméticos mais comuns em programação são apresentados na **Tabela 1** com seus respectivos significados.

OPERADOR	SIGNIFICADO
-	menos unário (i.e., inversão de sinal)
+	soma
-	subtração
*****	multiplicação
/	divisão
%	resto da divisão inteira

Tabela 1: Operadores Aritméticos

Os operandos de qualquer operador aritmético devem ser numéricos. Existem dois tipos básicos de números em programação: **inteiros** e **reais**. Com exceção do operador "%", cujos operandos devem sempre ser inteiros, os operandos de qualquer outro operador aritmético podem ser inteiros ou reais. Quando os operandos de um operador aritmético

são de um mesmo tipo, o resultado será desse mesmo tipo. Se um dos operandos for real, o resultado também será desse tipo. Por exemplo:

Expressão	Resultado
`2.5 + 4`	`6.5`
`2.5 + 4.0`	`6.5`
`2 + 4`	`6`
`5 % 2`	`1`
`5 / 2`	`2`
`5.0 / 2`	2.5

A **Tabela 2** a seguir apresenta as propriedades de precedência e associatividade dos operadores aritméticos:

OPERADOR	PRECEDÊNCIA	ASSOCIATIVIDADE
– (unário)	**Alta** (aplicado primeiro)	À direita
*, /, **%**	↓	À esquerda
+, – (binários)	**Baixa** (aplicado por último)	À esquerda

Tabela 2: Precedências e Associatividades dos Operadores Aritméticos

Na **Tabela 2**, operadores numa mesma linha têm a mesma precedência. Portanto, quando tais operadores são encontrados juntos numa mesma expressão aritmética, o operador mais à esquerda é aplicado primeiro, exceto no caso do operador de inversão de sinal que tem associatividade à direita.

Note que o uso de parênteses altera as propriedades de precedência e associatividade dos operadores. Por exemplo, na expressão `(2 + 3)*4` os parênteses fazem com que a operação de soma seja aplicada antes da multiplicação (i.e., a precedência da soma torna-se maior do que a precedência da multiplicação). Outro exemplo: `8/(2/2)` resulta em `8` porque os parênteses aumentam a precedência do segundo operador de divisão.

Existem funções (v. **Seção 2.5.5**) que podem ser utilizadas para compor expressões aritméticas. Por exemplo, a função `sqrt` resulta na raiz quadrada de seu único operando. Quando uma função aparece numa expressão, ela é avaliada antes da aplicação de qualquer operador.

2.5.3 Operadores e Expressões Relacionais

Expressões relacionais são constituídas por **operadores relacionais** e operandos numéricos, e resultam num valor, denominado **valor lógico** ou **booleano**, que pode ser `verdadeiro` ou `falso`. Um operando de um operador relacional pode ser uma constante numérica, uma variável com conteúdo numérico ou uma expressão aritmética.

Os operadores relacionais comumente usados em programação e os respectivos resultados obtidos com suas aplicações são apresentados na **Tabela 3** a seguir[7].

EXPRESSÃO	VERDADEIRA QUANDO
A = B	A é igual a B
A ≠ B	A é diferente de B
A > B	A é maior do que B
A ≥ B	A é maior do que ou igual a B
A < B	A é menor do que B
A ≤ B	A é menor do que ou igual a B

Tabela 3: Operadores Relacionais

Note que uma expressão relacional corresponde a uma questão cuja resposta é *sim* ou *não*. Quando a resposta a essa questão é *sim*, o resultado da expressão é `verdadeiro`; quando a resposta a essa pergunta é *não*, o resultado da expressão é `falso`. Por exemplo, `2 > 3` corresponde à questão: *Dois é maior do que três?* cuja resposta é, obviamente, *não* e, portanto, a expressão `2 > 3` resulta em `falso`.

Conforme foi mencionado antes, um operando de uma expressão relacional pode ser uma expressão aritmética. Isso significa que se podem ter expressões contendo operadores aritméticos e relacionais. Por exemplo, `2 + 3 > 4 * 6` é uma expressão contendo dois operadores aritméticos (`+` e `*`) e um operador relacional (`>`), e é interpretada como: *dois mais três é maior do que quatro vezes seis?* Note que, para essa expressão ter essa interpretação, está implícito que as operações de soma e multiplicação devem ser efetuadas antes da operação relacional *maior do que*. Isto é, os operadores de soma e multiplicação devem ter maior precedência do que o operador *maior do que*.

Em geral, todos os operadores relacionais fazem parte de um mesmo grupo de precedência e a precedência deles é menor do que a precedência de qualquer operador aritmético. Isso significa que numa expressão contendo operadores aritméticos e

[7] Evidentemente, o resultado de um operador relacional é `falso`, quando não é `verdadeiro`.

relacionais, os operadores aritméticos são sempre aplicados antes dos operadores relacionais.

2.5.4 Operadores e Expressões Lógicos

Expressões contendo operadores relacionais constituem exemplos de **expressões lógicas**. Uma expressão lógica (também conhecida como **expressão booleana**) é simplesmente uma expressão que resulta num valor lógico (i.e., `verdadeiro` ou `falso`).

Uma **variável lógica** (ou **variável booleana**) é uma variável que pode assumir apenas um valor lógico. A uma variável booleana pode-se atribuir diretamente uma constante lógica ou o valor resultante da avaliação de uma expressão booleana, como mostram os exemplos a seguir.

```
booleano  bol1, bol2

bol1 ← falso
bol2 ← 2 > 3
```

Constantes, variáveis e expressões booleanas podem ser combinadas entre si por meio de **operadores lógicos**. Existem três operadores lógicos mais comumente usados em programação:

- **Negação** – operador unário representado por `não` em pseudolinguagem
- **Conjunção** – operador binário representado por `e` em pseudolinguagem
- **Disjunção** – operador binário representado por `ou` em pseudolinguagem

Os possíveis resultados das aplicações desses operadores são tipicamente apresentados em tabelas denominadas **tabelas-verdade**. As tabelas-verdade para os operadores `não`, `e`, e `ou` são apresentadas a seguir[8].

Operando1	`não` **Operando1**
`verdadeiro`	`falso`
`falso`	`verdadeiro`

[8] Nessas tabelas, `operando1` e `operando2` podem corresponder a qualquer constante, variável ou expressão booleana.

Operando1	Operando2	Operando1 e Operando2
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	falso
falso	verdadeiro	falso
falso	falso	falso

Operando1	Operando2	Operando1 ou Operando2
verdadeiro	verdadeiro	verdadeiro
verdadeiro	falso	verdadeiro
falso	verdadeiro	verdadeiro
falso	falso	falso

As conclusões a seguir podem ser derivadas de observações das tabelas-verdade acima:

- O resultado da negação de um operando é `verdadeiro` quando o operando é `falso` e vice-versa.
- Em vez de memorizar toda a tabela-verdade do operador `e`, é necessário apenas lembrar que o resultado de `operando1 e operando2` é `verdadeiro` somente quando cada um dos operandos é `verdadeiro`; em qualquer outra situação a aplicação desse operador resulta em `falso`.
- Em vez de decorar toda a tabela do operador `ou`, é necessário somente lembrar que sua aplicação resulta em `falso` apenas quando seus dois operandos resultam em `falso`.

Exemplo:

```
booleano bol1, bol2, bol3, bol4, bol5, bol6

bol1 ← 2 = 5
bol2 ← falso
bol3 ← 10 ≤ 2*5
bol4 ← não bol1
bol5 ← bol2 e bol3
bol6 ← bol2 ou bol3
```

Exercício: Quais são os valores assumidos pelas variáveis lógicas `bol1`, `bol2`, `bol3`, `bol4`, `bol5` e `bol6` do último exemplo?

Um operando de uma expressão lógica pode ser uma expressão relacional, pois o resultado de tal expressão é sempre `verdadeiro` ou `falso`. Por outro lado, uma expressão relacional pode ter uma expressão aritmética como operando. Portanto, é possível que se tenha, numa mesma expressão, operadores aritméticos, relacionais e lógicos. Por exemplo:

```
(2 + 4  < 7) e não (x = 10)
```

é uma expressão perfeitamente legal. Assim, é necessário que se defina uma tabela de precedência que leve em consideração todos esses operadores. Essa precedência geral de operadores é apresentada na **Tabela 4**.

OPERADOR	PRECEDÊNCIA
- (unário), `não`	**Alta** (aplicado primeiro)
*, /, **%**	↓
+, - (binários)	↓
operadores relacionais (=, ≠, ≥, etc.)	↓
`e`	↓
`ou`	**Baixa** (aplicado por último)

Tabela 4: Precedência Geral de Operadores

Exercício: Se o valor da variável `x` for `5` no instante da avaliação da expressão `(2 + 4  < 7) e não (x = 10)`, o resultado dessa expressão será **verdadeiro**. Mostre, passo-a-passo, como esse resultado é obtido.

2.5.5 Funções

Além das operações elementares apresentadas, linguagens de alto nível tipicamente oferecem inúmeras funções que efetuam operações mais elaboradas. Os operandos sobre os quais essas funções atuam são comumente denominados **parâmetros**.

Algumas funções com suas denominações mais comumente utilizadas em programação são:

- `sqrt(x)` – calcula a raiz quadrada do parâmetro `x`
- `pow(x, y)` – calcula a exponencial do primeiro parâmetro elevado ao segundo; i.e., essa função calcula x^y

- `rand()` – resulta num número escolhido ao acaso (**número aleatório**) e não tem nenhum parâmetro

Quando uma função aparece numa expressão, sua execução tem prioridade maior do que a aplicação de qualquer operador apresentado antes. Além disso, quando há ocorrência de mais de uma função numa expressão, funções são avaliadas da esquerda para a direita. Por exemplo:

```
sqrt(4)*pow(2, 3) → 2*pow(2, 3) → 2*8 → 16
```

Nesse exemplo, as setas indicam a sequência de avaliação dos termos da expressão aritmética.

Linguagens de programação usualmente provêm o usuário com uma vasta coleção de tais funções que incluem, por exemplo, funções trigonométricas, exponenciais, etc.

2.5.6 Uso de Parênteses

Pode-se modificar precedência e associatividade de operadores por meio do uso de parênteses. Às vezes, o uso de parênteses é obrigatório para dar à expressão o significado que se pretende que ela tenha. Por exemplo, se você pretende que a expressão:

```
bol1 ou bol2 e bol3 ou bol4
```

seja interpretada como uma conjunção de duas disjunções; i.e., tenha o significado:

```
(bol1 ou bol2) e (bol3 ou bol4)
```

você tem que escrevê-la exatamente como na linha anterior (i.e., com os parênteses); caso contrário, a expressão original seria interpretada como:

```
bol1 ou (bol2 e bol3) ou bol4
```

visto que o operador `e` tem precedência maior do que a precedência do operador `ou`.

O uso de parênteses também é recomendado nos seguintes casos:

- Quando se tem dúvida relacionada com a precedência entre dois ou mais operadores.
- Para facilitar a leitura de expressões complexas.

Expressões envolvendo operadores relacionais e lógicos são particularmente vulneráveis e susceptíveis a erros em programação. O uso judicioso de parênteses para melhorar a legibilidade dessas expressões é uma arma preventiva contra erros.

Como conselho final, lembre-se que o uso redundante (mas não excessivo) de parênteses não prejudica o entendimento de uma expressão, mas a falta de parênteses pode resultar numa interpretação que não é a pretendida. Em outras palavras, é melhor ser redundante e ter um algoritmo funcionando do que tentar ser sucinto e ter um algoritmo defeituoso.

2.6 Entrada e Saída

Freqüentemente, um programa de computador precisa obter dados usando algum meio de entrada (por exemplo, teclado, disco). Em linguagem algorítmica, utiliza-se a instrução `leia` para a obtenção de dados para um algoritmo. Essa instrução sempre vem acompanhada de uma lista de variáveis que representam os locais em memória onde os dados lidos serão armazenados. Assim, uma instrução como:

```
leia(x1, x2, ..., xn)
```

indica que n dados serão lidos e então armazenados como conteúdos das variáveis x_1, x_2, ..., x_n.

Uma instrução para saída de dados de um algoritmo em algum meio de saída (por exemplo, tela, impressora) tem o seguinte formato em linguagem algorítmica:

```
escreva(e1, e2, ..., en)
```

Nessa instrução, e_1, e_2, ..., e_n representam a informação que será escrita no meio de saída e cada e_i pode ser:

- Um valor constante, que será escrito exatamente como ele é.
- Uma variável cujo conteúdo será escrito.
- Uma expressão que será avaliada e cujo valor resultante será escrito.
- Uma cadeia de caracteres (v. abaixo).

Uma **cadeia de caracteres** (ou **string**) consiste de uma seqüência de caracteres entre aspas. Por exemplo, `"Boa Sorte"` é uma cadeia de caracteres. Quando uma cadeia de

caracteres aparece numa instrução `escreva`, todos os seus caracteres, exceto as aspas que delimitam a cadeia, são escritos na ordem em que aparecem.

2.7 Estruturas de Controle

O **fluxo de execução** de um algoritmo consiste na seqüência e na freqüência (i.e., número de vezes) com que suas instruções são executadas. No **fluxo natural de execução** de um algoritmo, cada instrução é executada exatamente uma vez e na ordem em que aparece no algoritmo.

Estruturas de controle são instruções que permitem que o programador altere o fluxo natural de execução de um algoritmo. Elas são divididas em três categorias:

- **Desvios condicionais** alteram a seqüência de execução de um algoritmo causando o desvio do fluxo de execução do algoritmo para uma determinada instrução. Esse desvio é condicionado ao valor resultante da avaliação de uma expressão lógica.
- **Desvios incondicionais** são semelhantes aos desvios condicionais, mas os desvios causados por eles independem da avaliação de qualquer expressão.
- **Repetições** (ou **laços de repetição**) alteram a freqüência com que uma ou mais instruções de um algoritmo são executadas.

Algumas estruturas de controle já foram apresentadas informalmente no exemplo de algoritmo da **Seção 2.2**. A seguir serão apresentadas uma instrução de desvio condicional e duas instruções de repetição usadas pela linguagem algorítmica. Desvios incondicionais serão apresentados mais adiante.

2.7.1 Desvio Condicional se-então-senão

A instrução `se`-`então`-`senão` possui o seguinte formato:

```
se (condição) então instruções1 senão instruções2
```

Onde:

- *condição* representa uma expressão lógica que determina para qual instrução o desvio será realizado.

- *instruções1* representa um conjunto de uma ou mais instruções para onde o desvio será efetuado se o valor resultante da avaliação da expressão *condição* for `verdadeiro`.
- *instruções2* representa um conjunto de uma ou mais instruções para onde o desvio será efetuado se o valor resultante da avaliação da expressão *condição* for `falso`.

É importante salientar que a parte `senão` da instrução condicional é opcional. Quando essa parte da instrução estiver ausente, ocorrerá desvio apenas se o resultado da expressão condicional for `verdadeiro`.

Exemplo:

```
leia(x)

se (x < 0) então
   escreva("O número é negativo")
senão
   escreva("O número é positivo ou zero")
```

Nesse exemplo, se o valor lido para `x` for `-5`, a instrução:

```
escreva("O número é negativo")
```

será executada, mas não será o caso da instrução:

```
escreva("O número é positivo ou zero")
```

2.7.2 Desvio Condicional selecione-caso

A instrução `selecione-caso` é uma instrução que permite **desvios condicionais múltiplos** e útil quando existem várias ramificações possíveis a ser seguidas num algoritmo. Nesse caso, o uso de instruções `se-então-senão` aninhadas torna o programa difícil de ser lido.

A sintaxe da instrução `selecione-caso` é:

```
selecione (expressão-inteira)
   caso constante1
      instruções1
   caso constante2
      instruções2
   ...
   caso constanteN
      instruçõesN
   padrão
      instruçõesp
```

A expressão entre parênteses que segue imediatamente a palavra *selecione* deve resultar num valor inteiro. Então, quando o valor resultante da avaliação dessa expressão coincide com o valor de uma das constantes que acompanham as palavras *caso*, as instruções correspondentes à respectiva constante são executadas.

A parte da instrução selecione-caso que inicia com a palavra *padrão* é opcional e suas instruções são executadas quando o valor resultante da avaliação da expressão não coincidir com o valor de nenhuma constante precedida por *caso*.

Exemplo:

```
inteiro opção

escreva("Escolha uma das cinco opções: ")
leia(opção)

selecione (opção)
   caso 1
      escreva("Você escolheu a opção 1")
   caso 2
      escreva("Você escolheu a opção 2")
   caso 3
      escreva("Você escolheu a opção 3")
   caso 4
      escreva("Você escolheu a opção 4")
   caso 5
      escreva("Você escolheu a opção 5")
   padrão
      escreva("Você não escolheu uma opção válida")
```

2.7.3 Estrutura de Repetição enquanto-faça

A instrução enquanto-faça tem o seguinte formato:

```
enquanto (condição) faça instruções
```

Onde:

- *condição* representa uma expressão lógica que controla a repetição.
- *instruções* representa um conjunto de uma ou mais instruções, denominado **corpo do laço**, que poderá ser executado várias vezes.

O laço `enquanto`-`faça` funciona da seguinte maneira: a expressão *condição* é avaliada; se o resultado dessa avaliação for `verdadeiro`, o corpo do laço é executado; caso contrário, o laço é encerrado. Esse procedimento é repetido até que o resultado da expressão *condição* seja `falso`.

É importante observar o seguinte:

- Se a primeira avaliação expressão *condição* resultar em `falso`, o corpo do laço não é executado nenhuma vez.
- Se a expressão *condição* nunca resultar em `falso`, o laço nunca terminará e a estrutura é denominada **laço infinito**.

Exemplo 1:

```
leia(x)

enquanto (x < 10) faça
     x ← x + 1
```

Exercício: O que ocorreria se o corpo do laço desse exemplo fosse `x ← x - 1` e o valor lido fosse menor do que 10?

Exemplo 2:

```
inteiro soma, cont

soma ← 0
cont ← 1

enquanto (cont < 10) faça
   soma ← soma + cont
   cont ← cont + 1

escreva("Resultado: ", soma)
```

Exercício: O que escreve o algoritmo do último exemplo?

2.7.4 Estrutura de Repetição faça-enquanto

O laço **faça-enquanto** tem o seguinte formato:

```
faça instruções enquanto (condição)
```

A única diferença entre os laços faça-enquanto e enquanto-faça diz respeito a *quando* a expressão condicional é avaliada. Na instrução enquanto-faça, a expressão condicional é avaliada *antes* do corpo do laço; na instrução faça-enquanto, a expressão condicional é avaliada *depois* do corpo do laço. Como conseqüência, o corpo de um laço enquanto-faça pode não ser executado nenhuma vez, mas o corpo de um laço faça-enquanto sempre é executado pelo menos uma vez.

Exemplo:

```
leia(x)

faça
   x ← x + 1
enquanto (x < 10)
```

Compare esse último exemplo com o primeiro exemplo da **Seção 2.7.3**. Observe que a única diferença entre eles é que aquele exemplo usa um laço enquanto-faça e esse exemplo utiliza um laço faça-enquanto. Agora, supondo que o valor lido e armazenado na variável x fosse 12 nos dois casos, o corpo do laço enquanto-faça do exemplo anterior não seria executado nenhuma vez e o corpo do laço faça-enquanto do presente exemplo seria executado exatamente uma vez.

2.7.5 Desvio Incondicional pare

A instrução pare é usada para encerrar um laço de repetição, fazendo com que o fluxo de execução seja desviado para a próxima instrução que segue o respectivo laço Por exemplo, o algoritmo a seguir calcula a soma dos valores numéricos introduzidos até que seja introduzido o valor zero:

```
soma ← 0

enquanto (verdadeiro) faça
   leia(x)
```

```
        se (x = 0) então
           pare

        soma ← soma + x

    escreva("Soma dos valores: ", soma)
```

Observe que a expressão condicional que acompanha a instrução `enquanto-faça` é representada pelo valor constante `verdadeiro`. Portanto, a única maneira de encerrar o laço é por meio de uma instrução `pare`.

2.7.6 Blocos e Endentações

Em programação, **endentação** é um pequeno espaço em branco horizontal que indica subordinação de uma instrução em relação a outra. Todos os exemplos de estruturas de controle apresentados usam endentação para indicar subordinação de instruções às estruturas de controle a que pertencem.

Num algoritmo, um **bloco** é uma seqüência de instruções que não apresentam dependências entre si. Assim, instruções que pertencem a um mesmo bloco não apresentam endentações entre si. Por exemplo, no trecho de algoritmo a seguir:

```
x ← 1

enquanto (x <10) faça
   x ← x + 1
   escreva(x)

escreva("Bye, bye")
```

as instruções `x ← x + 1` e `escreva(x)` estão num mesmo nível de endentação e, portanto, fazem parte de um mesmo bloco. Esse bloco está subordinado ao laço `enquanto (x <10) faça` e, por isso, ele está endentado em relação a esse laço. A instrução `escreva("Bye, bye")`, que não está endentada, não pertence ao bloco que constitui o corpo do referido laço.

Exercício: Quantos blocos existem no último exemplo e quais são as instruções que fazem parte de cada bloco?

Linguagens de programação modernas consideram espaços em branco múltiplos, como aqueles usados em endentações, como um único espaço. Portanto, nessas linguagens, outros mecanismos são usados para indicar os limites de um bloco. Em C, por exemplo, um bloco é delimitado pelos caracteres **{** e **}**. Contudo, na linguagem algorítmica usada

aqui, que será lida apenas por pessoas, endentação e alinhamento são suficientes para delimitar blocos.

2.8 Comentários e Legibilidade de Algoritmos

Legibilidade é uma propriedade altamente desejável de um algoritmo. Um algoritmo bem legível é mais fácil de entender, testar e refinar do que um algoritmo com legibilidade sofrível.

As seguintes recomendações devem ser seguida na construção de algoritmos legíveis:

- **Incorpore no algoritmo comentários em Português claro**. Comentários são inseridos num algoritmo entre /* (abertura) e */ (fechamento). O objetivo de comentar um algoritmo é torná-lo mais legível e o melhor momento para tal é durante a concepção do algoritmo (e não depois de uma semana, por exemplo). Comentários devem acrescentar alguma coisa além daquilo que pode ser facilmente apreendido. Por exemplo, na instrução:

  ```
  x ← 2  /* x recebe o valor 2 */
  ```

  o comentário é completamente redundante e irrelevante, pois um programador com um mínimo de conhecimentos básicos sobre algoritmos conhece o significado dessa instrução sem a necessidade de qualquer comentário.

- **Utilize nomes de variáveis que sejam representativos**. Em outras palavras, o nome de uma variável deve sugerir o tipo de informação armazenado nela. Por exemplo, num algoritmo para calcular médias de alunos numa disciplina, os nomes de variáveis `nome`, `nota1`, `nota2` e `média` serão certamente muito mais significativos do que simplesmente `x`, `y`, `w` e `z`.

- **Grife todas as palavras-chave** (por exemplo, `leia`, `se`, `então`) utilizadas no algoritmo[9].

- **Use endentação coerentemente**. Não existe nenhum formato considerado mais correto para endentação, embora alguns formatos de

[9] Palavras-chave são escritas muitas vezes em **negrito** em livros-textos e outros materiais impressos, mas esse recurso não está disponível, ou pelo menos não é muito prático, em algoritmos manuscritos.

endentação sejam mais recomendados do que outros. Por exemplo, alguns programadores preferem alinhar as partes de uma instrução se-então-senão como:

```
se (condição) então
   instruções
senão
   instruções
```

enquanto outros podem preferir:

```
se (condição)
   então
      instruções
   senão
      instruções
```

Nenhuma das duas formas de endentação acima é mais correta ou mais legível do que a outra. Isto é, usar um ou outro formato é uma questão de gosto pessoal. Mas, uma vez que um formato de endentação tenha sido escolhido, ele deve ser consistentemente mantido na escrita dos algoritmos.

- **Use espaços em branco verticais judiciosamente**. O uso de espaços verticais pode melhorar a legibilidade de algoritmos da mesma forma que a divisão em parágrafos melhora a legibilidade de textos comuns. Em especial, use espaços em branco verticais para separar declarações de variáveis e o corpo do algoritmo. Separe também grupos de instruções que têm funções diferentes. Como com endentação, o uso de espaços em branco para melhorar a legibilidade de um algoritmo não possui regras fixas: simplesmente use o bom senso e não exagere nem para mais nem para menos.

- **Use espaços em branco horizontais para enfatizar precedência de operadores**. Por exemplo:

```
5*3 + 4
```

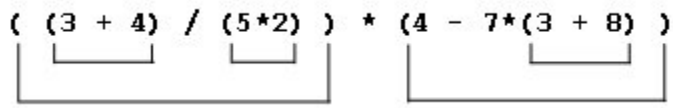
é melhor do que:

```
5 * 3 + 4
```

ou:

```
5*3+4
```

- **Use parênteses em expressões para torná-las mais legíveis** (ver **Seção 2.5.6**). Não se preocupe em ser redundante e lembre-se que para cada parêntese aberto deve haver um parêntese de fecho. Isso significa que o número de parênteses de abertura deve ser igual ao número de parênteses de fechamento. Uma forma de checar as correspondências entre parênteses numa expressão contendo muitos parênteses é traçando-se linhas conectando cada par de parênteses, como mostra o exemplo a seguir[10].



  Esse modo de checar expressões parentéticas é muito mais eficiente do que simplesmente contar o número de parênteses abertos e comparar para ver se ele casa com o número de parênteses fechados.

- **Nunca escreva mais de uma instrução por linha**. O que separa instruções da linguagem algorítmica utilizada aqui é quebra de linha. Isso significa que, mesmo que você tenha algumas instruções suficientemente pequenas, não as escreva numa mesma linha para não prejudicar a legibilidade do algoritmo.

- **Quebre linhas de instrução muito extensas**. Quando uma instrução for muito extensa para caber numa única linha, quebre a linha e use endentação na linha seguinte para indicar a continuação da instrução. Por exemplo, é melhor escrever a instrução[11]:

```
escreva("Esta é uma instrução muito looooooooooooooonga")
```

  assim:

```
escreva("Esta é uma instrução muito
        looooooooooooooooooooooonga")
```

2.9 Como Construir um Programa I: Projeto de Algoritmo

[10] Isso é apenas um artifício de verificação. Obviamente, essas linhas não deverão fazer parte do algoritmo.
[11] Suponha que a instrução do exemplo seja realmente bem mais extensa.

A construção de um programa *de pequeno porte* em qualquer linguagem algorítmica convencional segue, normalmente, a seqüência de etapas mencionadas a seguir:

1. Análise do problema (v. **Seção 2.9.1**)
2. Refinamento do algoritmo (v. **Seção 2.9.2**)
3. Teste do algoritmo (v. **Seção 2.9.3**)
4. Codificação do algoritmo (v. **Seção 3.16.1**)
5. Construção do programa executável (v. **Seção 3.16.2**)
6. Teste do programa (v. **Seção 3.16.3**)

As três primeiras etapas serão exploradas em profundidade a seguir, enquanto que as demais serão estudadas no **Capítulo 3**.

Antes de prosseguir, é importante salientar que um computador faz apenas aquilo que você é capaz de instruí-lo a fazer. Assim, a principal premissa de programação pode ser enunciada como:

> ***O computador faz apenas aquilo que é instruído a fazer (o que nem sempre corresponde àquilo que você deseja que ele faça).***

Portanto, quando escrever um algoritmo, tente *pensar* como o computador irá executá-lo; i.e., sem fazer nenhuma inferência, conjectura ou suposição, pois o computador não possui essa capacidade.

2.9.1 Etapa 1: Análise do Problema

O primeiro passo a ser seguido na construção de um algoritmo consiste na análise precisa do problema. Esse passo é fundamental e, portanto, não deve ser negligenciado.

Nessa etapa, o enunciado do problema deve ser analisado palavra a palavra até que os dados de entrada e saída possam ser devidamente identificados. Feito isso, tenta-se descrever um procedimento em língua portuguesa que mostre como obter o resultado desejado (saída) usando os dados disponíveis (entrada). O quadro a seguir resume essa etapa.

1. Leia e reflita cuidadosamente sobre o problema, e responda às seguintes questões:
   1.1 Que dados iniciais do problema estarão disponíveis (entrada)? Escreva a resposta a essa questão precedida pela palavra *Entrada*.

1.2 Qual é o resultado esperado (saída)? Escreva a resposta precedendo-a por *Saída*.

1.3 Que tipo de processamento (algoritmo) é necessário para obter o resultado esperado a partir dos dados de entrada? Escreva um algoritmo que faça isso. Não se preocupe, por enquanto, com que o algoritmo seja bem detalhado. Você pode, inclusive, escrevê-lo em Português, em vez de usar a linguagem algorítmica.

As respostas às questões 1.1 e 1.2 apresentadas no quadro acima são necessárias para identificar os dados de entrada e saída do programa. Ao responder essas questões, use nomes significativos (não necessariamente em português) para representar dados de entrada e saída (e.g., *matriculaDoAluno*). Certifique-se de que essas respostas são bem precisas antes de prosseguir para a **Etapa 1.3**.

Na **Etapa 1.3**, você deve encontrar uma conexão entre os dados de entrada e o resultado desejado que permita determinar quais são os passos do algoritmo que conduzem ao resultado. A seguir são apresentadas algumas recomendações para ser bem sucedido na escrita desse esboço de algoritmo[12]:

- Construa um exemplo previsto de execução do programa resultante e use-o até o final do processo de desenvolvimento. Um exemplo de execução é útil não apenas na fase de construção do algoritmo como também serve como caso de teste do próprio algoritmo e do programa que resultará ao final do processo de desenvolvimento (v. exemplo adiante).
- Desenhe diagramas que auxiliem seu raciocínio. Em particular, use um retângulo para representar cada variável e acompanhar eventuais alterações de seu conteúdo representado pelo interior do retângulo, como mostram vários exemplos apresentados neste livro.
- Se encontrar dificuldades para criar um algoritmo preliminar nessa etapa, procure um problema mais simples que seja parecido com aquele em questão, mas que seja considerado mais fácil. Resolver um problema análogo mais simples pode fornecer pistas para resolução de um problema mais complexo.

Como exemplo de realização dessa etapa, suponha que seu algoritmo se propõe a resolver equações de segundo grau (i.e., $ax^2 + bx + c = 0$). Então, nessa etapa de construção do algoritmo, você deverá obter o seguinte:

[12] Algumas das sugestões apresentadas a seguir foram adaptadas do livro *How to Sove It* do matemático húngaro George Pólya (v. **Bibliografia**).

Entrada: a, b, c (valores reais que representam os coeficientes da equação)

Saída:

- "Os coeficientes não constituem uma equação do segundo grau" (ou, equivalentemente, "O valor de 'a' não pode ser zero")
- "Não há raízes reais"
- x1, x2 (as raízes reais da equação, se elas existirem)

Algoritmo:

1. Leia os valores dos coeficientes e armazene-os nas variáveis a, b, c.
2. Se os coeficientes não constituírem uma equação do segundo grau, relate o fato e encerre.
3. Calcule o discriminante (Δ) da equação.
4. Se o valor do discriminante for menor do que zero, informe que não há raízes reais e encerre.
5. Calcule os valores das raízes
6. Apresente as raízes

Exemplos previstos de execução:

Exemplo 1:

```
Coeficiente quadrático (a): 0
Resultado: O valor de a não pode ser zero
```

Exemplo 2:

```
Coeficiente quadrático (a): 1
Coeficiente linear (b): -2
Coeficiente constante (c): 4

Resultado: Não há raízes reais
```

Exemplo 3:

```
Coeficiente quadrático (a): 1
Coeficiente linear (b): -5
Coeficiente constante (c): 6

Resultado: As raízes são: 3 e 2
```

Exemplo 4:

```
Coeficiente quadrático (a): 1
Coeficiente linear (b): -2
Coeficiente constante (c): 1

Resultado: As raízes são: 1 e 1
```

Apesar da relativa simplicidade do problema exemplificado, dificilmente um programador iniciante consegue, na primeira tentativa, ser bem sucedido nessa etapa. No algoritmo em questão, o erro mais comum entre iniciantes é supor que a única saída do algoritmo são as raízes da equação. Ou seja, freqüentemente, os iniciantes esquecem que pode ser que não haja raízes ou mesmo que os coeficientes lidos sequer constituem uma equação do segundo grau. Por isso, é importante que haja uma profunda reflexão sobre o problema para que se determine com exatidão quais são os dados de entrada e saída de um algoritmo antes de prosseguir com sua escrita. É impossível que um algoritmo seja escrito corretamente quando suas entradas e saídas não são bem especificadas.

2.9.2 Etapa 2: Refinamento do Algoritmo

A segunda etapa de construção de um programa consiste no refinamento do algoritmo preliminar obtido na primeira etapa:

> 2. Subdivida cada passo do algoritmo esboçado na **Etapa 1.3** que não tenha solução trivial.

Nessa etapa, a abordagem dividir e conquistar (v. **Seção 2.2**) deve ser aplicada sucessivamente até que cada subproblema possa ser considerado trivial. O nível de detalhamento de cada instrução resultante depende do grau de conhecimento do programador relativo ao problema e à linguagem de programação para a qual o algoritmo será traduzido. Tipicamente, programadores novatos precisam de muito mais detalhes do que programadores experientes. Se o pseudocódigo resultante dessa etapa for difícil de ler ou traduzir numa linguagem de programação, deve haver algo errado com o nível de detalhamento adotado.

O algoritmo preliminar para resolução de equações do segundo grau apresentado na **Seção 2.9.1** seria refinado nessa etapa como:

```
real a, b, c, x1, x2, delta

leia(a)

se (a = 0) então
   escreva("O valor de a não pode ser zero")
senão
   leia(b, c)
```

```
delta = b*b - 4*a*c

se (delta < 0)
   escreva("Não há raízes reais")
senão
   x1 = (-b + sqrt(delta))/(2*a)
   x2 = (-b - sqrt(delta))/(2*a)
   escreva("As raízes são: ", x1, x2)
```

Observações:

- Note que a variável `delta` no algoritmo não representa nem entrada nem saída do algoritmo; i.e., ela é usada como variável auxiliar no processamento.
- O algoritmo apresentado acima não é a *única* solução para o problema proposto. Normalmente, existem muitos algoritmos *funcionalmente equivalentes* que resolvem um determinado problema. Talvez, nem todos eles sejam equivalentes em termos de eficiência, mas, por enquanto, não se preocupe com esse aspecto.
- Não espere obter rapidamente um refinamento como o apresentado acima. Isto é, pode ser que você precise fazer vários refinamentos intermediários antes de obter um algoritmo satisfatório (i.e., um algoritmo no qual todos os passos são considerados *triviais*). Como exercício, refine o esboço de algoritmo apresentado na **Seção 2.9.1** e tente obter um algoritmo equivalente ao apresentado acima. O resultado que você obterá não precisa ser igual ao último algoritmo apresentado, mas deve ser funcionalmente equivalente a ele.

2.9.3 Etapa 3: Teste do Algoritmo

Após obter um algoritmo refinado, você deve testá-lo:

3. Verifique se o algoritmo realmente funciona conforme o esperado. Ou, mais precisamente, teste o algoritmo com alguns casos de entrada que sejam qualitativamente diferentes e verifique se ele produz a saída desejada para cada um desses casos. Se o algoritmo produz respostas indesejáveis, volte à **Etapa 1**.

Considerando o exemplo de equações de segundo grau apresentado acima, se você testar seu algoritmo apenas com casos de entrada que resultem sempre em $\Delta > 0$, independentemente da quantidade de testes, eles não serão qualitativamente diferentes, e portanto, não serão suficientes para testar seu algoritmo. Testes qualitativamente diferentes devem verificar todas as saídas possíveis de um algoritmo.

Note ainda que, caso você tenha que voltar à **Etapa 1**, como recomendado no último quadro, não precisa desfazer tudo que já fez até aqui. Pode ser, por exemplo, que você tenha apenas esquecido de levar em consideração algum dado de entrada (**Etapa 1.1**), ou um dos passos do algoritmo preliminar (**Etapa 1.3**) ou seu refinamento (**Etapa 2**) seja inadequado.

Para testar um algoritmo, você deve fazer papel tanto de computador quanto de usuário. Isto é, você deverá executar o algoritmo manualmente, como se fosse um computador, e também deverá fornecer dados de entrada para o algoritmo, como se fosse um usuário. Por exemplo, considere o algoritmo para resolução de equações do segundo grau apresentado acima. O primeiro passo desse algoritmo é a instrução:

```
leia(a)
```

Durante os testes, você deverá introduzir um valor para a variável `a`, exercendo, assim, o papel de usuário, e ler o respectivo valor, fazendo o papel de computador. Suponha que você introduz (como usuário) e lê (como computador) o valor zero. Então, após a execução manual dessa instrução o valor da variável `a` passa a ser zero.

O início da próxima instrução do algoritmo é:

```
se (a = 0) então
   escreva("O valor de a não pode ser zero")
```

Como o valor da variável `a` neste instante da execução do algoritmo é zero, de acordo com a interpretação da instrução `se`-`então`-`senão`, a instrução:

```
escreva("O valor de a não pode ser zero")
```

será executada. Assim, o resultado é a escrita de:

```
O valor de a não pode ser zero
```

Após a escrita dessa frase, o algoritmo encerra porque a parte `senão` da referida instrução `se`-`então`-`senão` não é executada e não há mais nenhuma outra instrução que possa ser executada no algoritmo. Agora, observando o primeiro exemplo previsto de execução, esse era realmente o resultado esperado do algoritmo. Portanto, o algoritmo é aprovado no teste do caso de entrada no qual o coeficiente `a` é igual a zero.

Os quatro exemplos previstos de execução apresentados na **Seção 2.9.1** representam casos de entrada qualitativamente diferentes. Portanto, todos eles devem ser testados. Acima, foi mostrado como testar o algoritmo com o primeiro caso de entrada (i.e., quando `a` é igual a zero). É deixado como exercício, testar os demais casos usando o mesmo raciocínio empregado acima.

2.9.4 Implementação

Comumente, as *últimas* etapas de construção de um programa são coletivamente denominadas **implementação** e consistem em edição do programa-fonte, construção do programa executável e subseqüentes testes. Na realidade, essas etapas só podem ser consideradas *derradeiras* se o programa resultante for absolutamente correto, o que raramente ocorre. O mais comum é que ele contenha falhas e, conseqüentemente, seja necessário repetir o processo de construção a partir de alguma das etapas anteriores.

O que ocorre a partir da **Etapa 4** (inclusive) de construção de um programa é objeto de estudo do próximo capítulo. Mas, apenas a título de ilustração e para não deixar incompleto o exemplo de equações do segundo grau já iniciado, a codificação em C do algoritmo que resolve esse problema é apresentada a seguir:

```
   /* Permite usar funções de biblioteca */
#include "leitura.h" /* Função LeReal() */
#include <stdio.h>   /* Função printf() */
#include <math.h>    /* Função sqrt()   */

int main(void)
{
   /* >>> A tradução do algoritmo começa a seguir <<< */

      /* double é o equivalente em C ao tipo */
      /* real da linguagem algorítmica       */
   double a, b, c, /* Coeficientes da equação */
          x1, x2,  /* As raízes */
          delta;   /* Discriminante da equação */

      /* Solicita ao usuário e lê o coeficiente 'a' */
   printf("\nDigite o coeficiente quadratico (a): ");
   a = LeReal();

      /* Se o valor de 'a' for igual a zero, não */
      /* se terá uma equação do segundo grau     */
   if (a == 0) {
         /* Apresenta o resultado do programa */
      printf("\nO valor de a nao pode ser zero\n");
   } else {
         /* Solicita ao usuário e lê o coeficiente 'b' */
      printf("\nDigite o coeficiente linear (b): ");
      b = LeReal();

         /* Solicita ao usuário e lê o coeficiente 'c' */
      printf("\nDigite o coeficiente constante (c): ");
      c = LeReal();

      delta = b*b - 4*a*c; /* Calcula o discriminante */

         /* Verifica se há raízes reais */
      if (delta < 0) { /* Não há raízes reais */
            /* Informa o resultado do programa */
         printf("\nNao ha' raizes reais\n");
      } else { /* Há raízes reais */
```

```
            /* Calcula as raízes da equação */
         x1 = (-b + sqrt(delta))/(2*a);
         x2 = (-b - sqrt(delta))/(2*a);

            /* Exibe o resultado do programa */
         printf( "\nAs raizes da equacao sao: %f e %f\n",
                 x1, x2 );
      }
   }

      /* Encerra o programa informando o sistema */
      /* operacional que não houve erro          */
   return 0;
}
```

Exemplos de execução do programa:

Exemplo 1:

Digite o coeficiente quadratico (a): 0

O valor de a nao pode ser zero

Exemplo 2:

Digite o coeficiente quadratico (a): 1

Digite o coeficiente linear (b): -2

Digite o coeficiente constante (c): 4

Nao ha' raizes reais

Exemplo 3:

Digite o coeficiente quadratico (a): 1

Digite o coeficiente linear (b): -5

Digite o coeficiente constante (c): 6

As raizes da equacao sao: 3.000000 e 2.000000

Exemplo 4:

Digite o coeficiente quadratico (a): 1

Digite o coeficiente linear (b): -2

Digite o coeficiente constante (c): 1

As raizes da equacao sao: 1.000000 e 1.000000

Mesmo que você ainda não tenha sido formalmente apresentado à linguagem C, leia o programa e tente encontrar correspondências entre ele e o algoritmo apresentado na **Seção 2.9.2**.

2.10 Exemplos de Programação

2.10.1 Troca de Valores entre Variáveis

Problema: Escreva um algoritmo que lê valores para duas variáveis, troca os valores dessas variáveis e exibe seus conteúdos antes e depois da troca. Em outras palavras, se as variáveis são `x` e `y`, ao final `x` terá o valor lido para `y`, e `y` terá o valor lido para `x`.

Solução:

Entrada: x, y
Saída: x, y

Algoritmo Preliminar:

```
1. Leia os valores de x e y
2. Escreva os valores de x e y
3. Guarde o valor de x numa variável auxiliar
4. Atribua o valor de y a x
5. Atribua o valor da variável auxiliar a y
6. Escreva  os valores de x e y
```

Algoritmo Refinado:

```
leia(x, y)
escreva("Valores de x e y antes da troca", x, y)

   /* Se o valor de x não for    */
   /* guardado, ele será perdido */
auxiliar ← x
x ← y
y ← auxiliar

escreva("Valores de x e y depois da troca", x, y)
```

2.10.2 Tabela de Potências

Problema: Escreva um algoritmo que calcule e apresente uma tabela apresentando as primeiras 100 potências de 2 (i.e., 2^n). (**NB**: Seu algoritmo não deve conter 100 instruções `escreva`!)

Solução:

Entrada: não há
Saída:

```
  0 1
  1 2
  2 4
  ...
100 2^99
```

Algoritmo Preliminar:

```
1. Inicie o expoente com 1
2. Inicie a potência com 1
3. Escreva 0 e 1
4. Enquanto o expoente for menor do que 100, faça o seguinte:
   4.1 Atribua à potência seu valor corrente multiplicado por 2
   4.2 Escreva o valor do expoente seguido da potência
   4.3 Atribua ao expoente seu valor corrente somado com 1
```

Algoritmo Refinado:

```
expoente ← 1
potência ← 1

   /* Exibe a primeira linha da tabela */
escreva("0 1")

   /* Apresenta as demais linhas da tabela */
enquanto (expoente < 100) faça
   potência ← potência * 2
   escreva(expoente, potência)
   expoente ← expoente + 1
```

2.10.3 Soma de um Número Indeterminado de Valores

Problema: Escreva um algoritmo que calcule e escreva a soma de um conjunto de valores. O algoritmo deve terminar quando for lido o valor zero.

Solução:

Entrada: conjunto de valores
Saída: a soma dos valores

Algoritmo Preliminar:

```
1. Leia o primeiro valor
2. Atribua o primeiro valor à variável que acumulará a soma
3. Enquanto o valor lido for diferente de zero, faça o seguinte:
   3.1 Leia um valor
   3.2 Atribua à variável soma seu valor corrente somado ao valor
   lido
4. Escreva a soma dos valores lidos
```

Algoritmo Refinado:

```
leia(valor) /* Lê o primeiro valor */

soma ← valor /* Inicia a soma com o valor lido */

   /* Enquanto o valor lido for diferente */
   /* de zero faça o seguinte             */
enquanto (valor ≠ 0) faça
   leia(valor)
   soma ← soma + valor

escreva(soma)
```

2.10.4 Seqüência de Fibonacci

Preâmbulo: Uma **seqüência de Fibonacci** é uma seqüência de números naturais, cujos dois primeiros termos são iguais a 1, e tal que cada número (exceto os dois primeiros) na seqüência é igual a soma de seus dois antecedentes mais próximos[13]. Isto é, a seqüência de Fibonacci é constituída da seguinte forma:

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...
```

Problema: Escreva um algoritmo que gera a seqüência de Fibonacci até o n-ésimo termo.

Solução:

Entrada: número de termos da seqüência.
Saída:

- Os termos da seqüência, se o número de termos for maior do que ou igual a 2.
- Relato de irregularidade se o número de termos for menor do que 2.

[13] Números de Fibonacci ocorrem misteriosamente na natureza. Por exemplo, uma flor de girassol possui duas espirais de sementes: uma com 21 sementes e a outra com 34 sementes. Acontece que 21 e 34 são dois números de Fibonacci consecutivos. Para quem gosta de numerologia...

Algoritmo Preliminar:

```
1. Leia o número de termos da seqüência
2. Se o número de termos introduzido for menor do que dois,
   informe que a seqüência não existe e encerre.
3. Escreva os dois primeiros termos da seqüência
4. Gere e apresente os termos da seqüência a partir do terceiro
   termo até o n-ésimo termo.
   4.1 Enquanto o n-ésimo termo não for gerado e escrito:
       4.1.1 Calcule o termo atual como sendo a soma dos dois
             termos antecedentes
       4.1.2 Escreva o termo atual
       4.1.3 Atualize os termos antecedentes
             4.1.3.1 O primeiro antecedente passa a ser o segundo
             4.1.3.2 O segundo antecedente passa a ser o atual
```

Algoritmo Refinado:

```
inteiro nTermos, antecedente1, antecedente2, atual, i

leia(nTermos)

se (nTermos < 2) então
   escreva("O numero de termos não pode ser menor do que 2")
senão
   antecedente1 ← 1
   antecedente2 ← 1

        /* Exibe os dois primeiros termos da série */
   escreva(antecedente1, antecedente2)

      /* Gera e Exibe os termos da seqüência a partir */
      /* do terceiro termo até o n-ésimo termo        */

      /* i é o índice do próximo termo */
   i ←  3

   enquanto (i <= nTermos) faça
      atual ←  antecedente1 + antecedente2
      escreva(atual)

         /* Atualiza os termos antecedentes */
      antecedente1 ←  antecedente2
      antecedente2 ←  atual

         /* Incrementa o índice do próximo termo */
      i ←  i + 1
```

Em C, esse algoritmo poderia ser codificado assim:

```
   /* Permite usar funções de biblioteca */
#include <stdio.h>   /* Função printf()    */
#include "leitura.h" /* Função LeInteiro() */

int main(void) /* Aqui começa a execução do programa */
```

```
{
      /* Declarações de variáveis */
   int antecedente1, antecedente2,
       atual, nTermos, i;

      /* Lê o número de termos da sequência */
   printf( "\nDigite o numero de termos da sequencia "
           "(pelo menos 2): " );
   nTermos = LeInteiro();

      /* Verifica se o número de */
      /* termos é menor do que 2 */
   if(nTermos < 2) {
      printf( "O numero de termos nao pode ser menor "
              "do que 2" );
   } else {
         /* Inicia os dois primeiros termos da série */
      antecedente1 = 1;
      antecedente2 = 1;

         /* Exibe os dois primeiros termos da série */
      printf("\n%d, %d", antecedente1, antecedente2);

         /* Gera e exibe os termos da seqüência a  */
         /* partir do terceiro até o n-ésimo termo */

         /* Inicia o índice do próximo termo da seqüência */
      i = 3;

      while (i <= nTermos) {
            /* Atualiza o termo corrente com a */
            /* soma dos seus dois antecedentes */
         atual = antecedente1 + antecedente2;

            /* Exibe o termo corrente */
         printf(", %d", atual);

            /* Atualiza os termos antecedentes */
         antecedente1 = antecedente2;
         antecedente2 = atual;

            /* Incrementa índice do próximo termo */
         i =  i + 1;
      }
   }

      /* Encerra o programa informando o sistema */
      /* operacional que não houve erro          */
   return 0;
}
```

Exemplo de execução do programa:

```
Digite o numero de termos da sequencia (pelo menos 2): 5

1, 1, 2, 3, 5
```

Novamente, mesmo que você ainda não conhece a linguagem C, leia o programa e tente encontrar alguma correspondência entre o algoritmo apresentado antes e esse programa.

2.11 Exercícios de Revisão

Definição de Algoritmo (Seção 2.1)

1. O que é um algoritmo?

2. (a) Em que aspectos o conceito de algoritmo é análogo ao conceito de receita culinária? (b) Quando essa analogia deixa de ser válida?

3. O que é um caso de entrada?

4. (a) O que é um algoritmo correto? (b) O que é um algoritmo incorreto?

5. O que são algoritmos funcionalmente equivalentes?

6. Cite três exemplos de problemas que não possuem algoritmos.

Abordagem Dividir e Conquistar (Seção 2.2)

7. Descreva a abordagem dividir e conquistar utilizada na construção de algoritmos.

8. De que depende o grau de detalhamento de instruções obtidas por meio de refinamentos sucessivos?

9. O que é reuso de código?

Linguagem Algorítmica (Seção 2.3)

10. (a) O que é linguagem algorítmica? (b) Por que linguagem algorítmica também é denominada *pseudolinguagem*?

11. Qual é a vantagem do uso de pseudolinguagem na construção de algoritmos em detrimento ao uso de linguagem natural (e.g., português)?

12. Por que uma linguagem de programação de alto nível não é conveniente para a escrita de algoritmos?

13. (a) O que pseudocódigo? (b) Existe tradutor, como aqueles descritos no **Capítulo 1**, para algoritmos escritos em pseudocódigo?

Variáveis e Atribuições (Seção 2.4)

14. Quais são os atributos que caracterizam uma variável?

15. O que é uma instrução de atribuição?

16. Qual é o significado da seguinte instrução:

```
x ← x + 2
```

Operadores e Expressões (Seção 2.5)

17. Descreva as seguintes propriedades de operadores:
    (a) Aridade
    (b) Resultado
    (c) Precedência
    (c) Associatividade

18. (a) O que é um operador unário? (b) O que é um operador binário?

19. O que é um grupo de precedência?

20. (a) O que é associatividade à esquerda? (b) O que é associatividade à direita?

21. Como os operadores aritméticos são agrupados de acordo com suas precedências?

22. (a) Para que servem os operadores relacionais? (b) Quais são os operadores relacionais?

23. Quais são os possíveis valores resultantes da avaliação de uma expressão relacional?

24. Descreva os operadores lógicos de negação, conjunção e disjunção.

25. O que é tabela-verdade?

26. Para que servem parênteses numa expressão?

27. Escreva a tabela-verdade correspondente à expressão booleana: `A ou B e C`, onde `A`, `B` e `C` são variáveis lógicas. (**Dica**: lembre-se que o operador `e` possui precedência maior do que o operador `ou`.)

28. Se `A = 150`, `B = 21`, `C = 6`, `L1 = falso` e `L2 = verdadeiro`, qual será o valor produzido por cada uma das seguintes expressões booleanas?

    (a) `não L1 e L2`

    (b) `não L1 ou L2`

    (c) `não (L1 e L2)`

    (d) `L1 ou não L2`

    (e) `(A > B) e L1`

    (f) `(L1 ou L2) e (A < B + C)`

Entrada e Saída (Seção 2.6)

29. Para que serve uma instrução de entrada de dados?

30. Descreva o funcionamento da instrução `leia`.

31. Para que serve uma instrução para saída de dados?

32. Descreva o funcionamento da instrução `escreva`.

33. Uma instrução `escreva` pode incluir expressões. Por que o mesmo não é permitido para uma instrução `leia`?

34. O que é uma cadeia de caracteres?

Estruturas de Controle (Seção 2.7)

35. (a) O que é fluxo de execução de um algoritmo? (b) Como é o fluxo natural de execução de um algoritmo?

36. O que são estruturas de controle?

37. Como estruturas de controle são classificadas?

38. Descreva o funcionamento da instrução `se-então-senão`.

39. Qual é a diferença entre as instruções `enquanto-faça` e `faça-enquanto` em termos de funcionamento?

40. (a) O que bloco? (b) O que é endentação? (c) Qual é a relação entre bloco e endentação em pseudolinguagem?

41. Considere o seguinte algoritmo, onde `i1`, `i2`, `i3`, `i4` e `i5` representam instruções:

```
booleano b1, b2, b3

se (b1) então
   i1
senão
   se (b2) então
      se (b3) então
         i2
      senão
         i3
         i4
i5
```

Agora, suponha que `V` representa `verdadeiro` e `F` representa `falso` e responda as seguintes questões:

(a) Que instruções serão executadas quando `b1 = V`, `b2 = V` e `b3 = F`?

(b) Que instruções serão executadas quando `b1 = F`, `b2 = V` e `b3 = F`?

(c) Que instruções serão executadas quando `b1 = F`, `b2 = V` e `b3 = V`?

(d) Que valores devem assumir `b1`, `b2` e `b3` para que apenas a instrução `i5` seja executada?

42. Qual será o valor da variável `resultado` após a execução do algoritmo a seguir?

```
booleano b1, b2, b3
real x, y
inteiro resultado

b1 ← falso
b2 ←  verdadeiro
b3 ←  falso

x ←  1.5
y ←  3.5
x ←  x + 1
```

```
se (b3 ou ((x + y > 5) ou (não b1 e b2))) então
   resultado ← 0
senão
   resultado ← 1
```

43. Duas instruções são funcionalmente equivalentes se elas produzem o mesmo efeito em quaisquer circunstâncias. Verifique se as instruções 1 e 2 abaixo são funcionalmente equivalentes:

    Instrução 1: `L ← X = Y`

    Instrução 2:
    ```
    se (X = Y)
       então L ← verdadeiro;
       senão L ← falso;
    ```

Comentários e Legibilidade de Algoritmos (Seção 2.8)

44. Cite cinco práticas recomendáveis na construção de algoritmos que favorecem a legibilidade.

45. O que é um nome de variável representativo?

46. (a) O que é comentário em construção de algoritmos? (b) Qual é a importância da incorporação de comentários num algoritmo?

47. (a) O que é endentação coerente? (b) Por que é aconselhável usar endentação coerentemente na construção de algoritmos?

Como Construir um Programa I: Projeto de Algoritmo (Seção 2.9)

48. Quais são as etapas envolvidas na escrita de um algoritmo?

49. Como um problema cuja solução algorítmica é desejada deve ser analisado?

50. (a) Como um algoritmo deve ser testado? (b) Como um programa deve ser testado?

51. (a) O que são casos de entrada qualitativamente diferentes? (b) Qual é a importância do uso de casos de entrada qualitativamente diferentes em testes de algoritmos?

52. Como exemplos de execução de um algoritmo auxiliam em seu processo de desenvolvimento?

53. Por que, mesmo que um algoritmo tenha sido testado e considerado correto, é necessário testar um programa derivado dele?

54. É sempre necessário declarar as variáveis de um algoritmo?

55. Por que a última etapa de construção de um programa nem sempre é *exatamente* a *última*?

2.12 Exercícios de Programação

2.12.1 Fácil

EP2.1) Escreva um algoritmo que recebe o raio de um círculo como entrada, calcula sua área e exibe o resultado. **Dado**: área de um círculo = πr^2, onde `r` é o raio do círculo. [**Dica**: para obter r^2, simplesmente calcule `r*r`.]

EP2.2) Escreva um algoritmo que lê um valor supostamente representando uma medida em polegadas e converte-o em centímetros. [**Dica**: 1 polegada corresponde a 2.54cm.]

EP2.3) Escreva um algoritmo que leia três valores inteiros que serão armazenados nas variáveis `x`, `y` e `z`. Então, o algoritmo calcula e exibe a soma e o produto desses valores.

EP2.4) Escreva um algoritmo que recebe dois números como entrada e exibe o menor deles. Se os números forem iguais, não haverá diferença em relação a qual deles será apresentado.

EP2.5) Escreva um algoritmo ligeiramente diferente daquele do exercício anterior que apresente uma mensagem (por exemplo, *Os números são iguais*) quando os números forem iguais.

EP2.6) Escreva um algoritmo que lê três inteiros e informa qual deles é o maior.

2.12.2 Moderado

EP2.7) Escreva um algoritmo que, repetidamente, lê valores reais que representam raios de círculos. Então, para cada valor lido, o algoritmo deve calcular a área do círculo respectivo e exibir o resultado. O algoritmo deve encerrar quando for lido um valor igual a zero.

EP2.8) Escreva um algoritmo que lê uma quantidade indeterminada de valores e exibe o menor deles. A entrada de valores encerra quando zero for lido. [**Sugestão**: Use uma variável para armazenar o menor valor lido. Inicialmente atribua a essa variável o primeiro valor lido. Então, a cada valor lido, compare-o com o valor da variável que deverá conter o menor. Se um novo valor lido for menor do que aquele correntemente armazenado nessa variável, atribua esse valor à variável.]

EP2.9) Escreva um algoritmo que calcule e apresente o fatorial de um dado número inteiro não negativo. O fatorial de um número inteiro $n \geq 0$ é dado por:

$$\begin{cases} 1, & \text{se } n = 0 \\ n * (n - 1) * (n - 2) * \ldots * 2 * 1, & \text{se } n > 0 \end{cases}$$

EP2.10) Edite o programa apresentado na **Seção 2.9.4** usando o conhecimento obtido na **Seção 1.6.3**. Então, construa um programa executável, conforme foi ensinado na **Seção 1.6.4**, e execute-o, conforme você aprendeu na **Seção 1.7**.

EP2.11) Repita o exercício **EP2.10** utilizando o programa apresentado na **Seção 2.10.4**, em lugar do programa da **Seção 2.9.4**.