Universidade Federal de Santa Maria
Prof. Cesar Tadeu Pozzer
Lógica e Algoritmo – ELC1064
06/06/2022

Lógica de Programação

1. Algoritmo

Algoritmo pode ser definido como um método para a **solução** de um determinado problema. Em termos de programação, este método deve ser **estruturado**, **ordenado** e **formal**, de modo que não apresente ambiguidades, que defina com **clareza** as operações que devem ser realizadas e seja finito.

A **granularidade** das operações vai depender da linguagem de programação que se estiver usando. Linguagens de baixo nível, como assembly, possuem comandos muito elementares, como somar, comparar e desviar. Linguagens como o C e C++ podem ser vistas como linguagens de nível médio, pois combinam elementos de linguagens de alto nível com o assembly, que é de baixo nível. A linguagem de nível mais alto, se existisse, seria igual a linguagem natural. Pode-se dizer que o algoritmo é um passo intermediário entre o problema e o programa para solucioná-lo.

Como exemplo introdutório de um algoritmo, pode-se descrever as operações necessárias para tomar um copo de água na cozinha:

```
Se você não estiver na cozinha então
   vá até a cozinha
fim se
Pegue um copo
encha de água
tome a água
```

Este algoritmo descreve ações complexas, que não podem ser compreendidas por um computador, que somente sabe realizar operações aritméticas e lógicas. Detalhando um pouco mais este exemplo, pode-se definir um novo algoritmo:

```
Se você não estiver na cozinha então
   enquanto não chegar a cozinha repita
      dê um passo em direção a cozinha, desviando de obstáculos
   fim enquanto
fim se
enquanto não encontrar um copo repita
   procure um copo
fim enquanto
mova a mão até o copo
pegue o copo
leve o copo embaixo da torneira
abra a torneira
enquanto o copo não estiver cheio repita
   espere 30 segundos
fim enquanto
feche a torneira
tome a água.
```

Este algoritmo **ainda não pode ser compreendido por um computador**. A tarefa de tomar água, para um computador, é muito complexa e exigiria milhares de linhas de código para ser executada. Geralmente, operações que para os humanos são triviais, para um computador são muito complexas senão impossíveis. Entretanto, cálculos complexos podem ser facilmente compreendidos por um computador.

Para que um algoritmo genérico seja traduzido para um algoritmo computacional, ele deve ser traduzido para **dados e comandos**. Um comando é uma primitiva que pode ser compreendida por um computador ou pela linguagem em que se de deseja implementar o algoritmo. Dados são representações formais de alguma informação, como um número, um caractere, um texto ou combinação destes. Existem várias formas de representação formais de algoritmos: Pseudo-Linguagem (Português Estruturado), fluxograma e diagrama de Nassi-Shneiderman.

Nestas representações, já muito próxima de uma linguagem de programação, porém sem muito formalismo, deve-se definir as variáveis e usar um conjunto limitados de comandos (ler, escrever, comparar, atribuir, operações aritméticas, laços). Elas serão apresentadas ao longo do texto.

1.1 Erros comuns no uso de laços de repetição (por Flávio Borin)

O laço de repetição indica uma ação que deve ser executada várias vezes. Exemplos:

Repita Ação relacionada a condição de parada Enquanto condição de parada	**Enquanto** condição **repita** Ação relacionada a condição de parada Fim enquanto

O laço enquanto não pode ser visto como sinônimo de "ao mesmo tempo" e então receber uma ação como parâmetro.

Errado	**Correto**
Enquanto a água esquenta **repita** Busque o café Fim enquanto	**Enquanto** a água não estiver fervendo **repita** Espere 30 segundos Fim enquanto Busque o café

Laços enquanto precisam que sua condição de parada seja satisfeita em algum momento. Se a condição de parada não for interferida pelas ações dentro do enquanto, o laço jamais parará:

Errado	**Correto**
Enquanto a torneira estiver aberta **repita** Molhe a escova Fim enquanto	**Enquanto** a escova não estiver molhada **repita** Molhe a escova Fim enquanto

O comando Fim enquanto somente pode ocorrer após o comando enquanto (o mesmo vale para o Fim se). E o comando Fim enquanto não deve ser seguido de nada. Exemplos que não fazem sentido:

Va ao banheiro Fim enquanto	**Enquanto** a panela não aquece **repita** Espere 30 segundos Fim enquanto desligue o fogão
Ler (“Arrumar a cama”) **Enquanto** (Hora>=6); **então** Escrever(“Levantar”); Senão (“Continuar deitado”)	**Enquanto** (“Não chegar no banheiro”); então Dê mais um passo para frente Repita até chegar

1.2 Exemplo detalhado: Robô despertador

Vamos considerar o algoritmo de um robô que tem a função de acordar o patrão de manha e após arrumar a sua cama e se recarregar.

Algoritmo	**Comentários**
`Ler hora de acordar`	A pessoa precisa informar ao robô que horas ele deve acordá-la. O robô é quem executa o algoritmo
`Enquanto não for hora de acordar repita`	Verificação se está na hora de acordar
`    Espere 30 segundos`	Se não for a hora de acordar (condição verdadeira), espere 30 segundos até verificar novamente (`horaAtual < horaAcordar`)
`Fim enquanto`	O laço termina quando for hora de acordar o patrão (condição falsa)
`Acorde o patrão`	Alguma ação que faça a pessoa acordar. Pode ser um grito, uma música, um pontapé, etc.
`Enquanto o patrão não levantar repita`	Esperar o patrão sair da cama
`    Espere 30 segundos`	Espere 30 segundos e verifique novamente
`Fim enquanto`	O patrão levantou e liberou a cama
`Arrume a cama`	O robô deve arrumar a cama
`Se a bateria estiver fraca então`	Se a bateria do robô estiver fraca
`   Enquanto não chegar perto da tomada`	Verificação se está perto da tomada
`      Ande em direção a tomada`	Andar em direção a tomada
`   Fim enquanto`	Chegou na tomada
`   Carregue a bateria`	Carrega a bateria
`Fim se`	Bateria carregada

2. Etapas para Solução de Problemas

Para se elaborar um algoritmo, deve-se inicialmente realizar algumas etapas:

1. **Identificar a entrada**: ver como os dados vão ser fornecidos e de onde vão ser lidos. Podem ser lidos do teclado ou de um arquivo, por exemplo. A entrada pode ser uma sequência de números, texto, imagens, arquivos de música, etc. A entrada pode consistir de uma única informação (e.g. um número) ou de um grupo de informações. Existem programas também que não requerem entrada de dados.
2. **Identificar a saída**: quais informações devem ser fornecidas? E para quem? Devem ser impressas na tela do computador, devem ser gravadas em um arquivo ou devem ser enviadas pela internet?
3. **Identificar o que deve ser feito**: Deve-se entender bem o problema para elaborar uma solução apropriada.
4. **Encontrar a melhor estratégia de solução**: uma vez identificado o problema a ser resolvido, deve-se escolher a melhor forma de implementá-lo. Geralmente cada problema tem uma grande gama de soluções. Algumas são mais rápidas, outras exigem maior esforço de programação, outras são de fácil entendimento, outras requerem mais memória, e assim por diante.

Exemplos

1. Calcular a média de uma turma de alunos
   Entrada: uma sequência de valores numéricos variando entre [0, 10]
   Saída: um único valor numérico que representa a média dos n valores lidos
   O que fazer: somar todas as notas e dividir pelo número de notas lidas
   Estratégia: Pode-se ler todas as notas e guardar em um vetor para posterior processamento, ou pode-se, por exemplo, ler a nota, somar e a descartar.

2. Verificar se algum cartão da Mega Sena foi premiado
   Entrada:
   Saída:
   O que fazer:
   Estratégia:

3. Procurar uma palavra dentro de um texto
   Entrada:
   Saída:
   O que fazer:
   Estratégia:

4. Fazer uma consulta de cadastro de cliente em um banco de dados
   Entrada:
   Saída:
   O que fazer:
   Estratégia:

5. Fazer uma jogada no Jogo da Velha
   Entrada:
   Saída:
   O que fazer:
   Estratégia:

6. Criar um personagem autônomo (inimigo) em um jogo de tiro (*First person shooter*)
   Entrada:
   Saída:
   O que fazer:
   Estratégia:

3. Variáveis

Todo o processamento lógico de um algoritmo é baseado em variáveis. As variáveis em algoritmos, semelhante às usadas em equações matemáticas, são usadas para guardar valores (dados). As variáveis podem conter números inteiros, reais, texto, dados mais complexos, etc.

Deve-se prestar atenção ao se dar nomes as variáveis. Primeiramente, todo nome de variável deve iniciar com uma letra, e segundo, deve-se dar nomes sugestivos, como mostrado no seguinte exemplo.

```
Var: contador, soma, nome, nota1, nota2, nota3, endereco
contador = 10
nota1 = 10
nome = ler( )
```

```
endereco = "Avenida Presidente Vargas"
soma = contador + 20
```

Para associar um valor a uma variável, deve-se fazer uma atribuição. Pode-se adotar várias sintaxes de atribuição de valores. Em ambos os casos, está sendo atribuído o valor 10 a variável contador.

```
contador = 10
contador ← 10
contador := 10
```

As variáveis podem receber valores de duas formas distintas: informadas pelo usuário, pelo comando `ler()`, ou atribuídas por meio de um operador de atribuição, como visto nos exemplos acima.

Antes de se utilizar uma variável, deve-se declará-la, com o uso da palavra **Var:**. Ao se utilizar uma linguagem de programação, além da declaração, deve-se também definir um tipo, que pode ser inteiro, real, texto, etc. No caso da linguagem C++, os tipos primitivos são: int, float, char, double e bool.

Deve-se tomar o cuidado de atribuir um valor a variável antes de acessar seu conteúdo. No exemplo acima, se a variável **contador** não tivesse sido inicializada com o valor 10, a expressão **soma = contador + 20** seria inválida, visto que não se saberia o valor da expressão.

A variável **endereco** foi inicializada com o texto "**Avenida Presidente Vargas**" e a variável **nome** com uma informação fornecida pelo usuário, através do comando `ler()`. A variável soma foi inicializada com o valor de uma expressão. Neste exemplo, seria inválido fazer a atribuição **soma = endereco + contador**, visto que as variáveis **endereco** e **contador** armazenam valores de tipos diferentes.

4. Comandos

Comandos são usados para **expressar alguma ação** que deve ser desempenhada pelo algoritmo. Os comandos podem ser simples, como por exemplo `ler()`, que indica que um valor deve ser lido (provavelmente do teclado), para alimentar alguma variável. Da mesma fora, o comando `escrever()` ou `imprimir()`, coloca alguma informação na "tela".

Pode-se também escrever comandos complexos como `decidir()`, `desenhar()`, `calcular()`, que podem, por exemplo, descrever ações cuja implementação pode ser muito complexa. Deve-se lembrar que um algoritmo não pode ser executado por um computador. Ele deve ser transcrito em alguma linguagem de programação antes de ser executado pelo computador.

Desta forma, devem ser facilmente compreendidos pelo programador. Comandos como `ler()` e `escrever()` são facilmente mapeados para funções que as linguagens de programação disponibilizam. Por outro lado, um simples comando `verificar_face()`, que verifica se a imagem de uma pessoa casa com a foto da sua carteira de identidade, pode requerer milhares de linhas de código para ser implementada, isso se for possível.

O seguinte exemplo, composto por 5 comandos em alto nível, representa um algoritmo de extrema complexidade de ser implementado, pois os comandos são muito genéricos e não se aproximam de instruções ou funções comumente oferecidas por linguagens de programação.

```
Se você não estiver na cozinha então
   vá até a cozinha
fim se
Pegue um copo
encha de água
```

```
tome a água
```

Por outro lado, o seguinte exemplo pode ser facilmente transcrito para uma linguagem de programação, pois os comandos são facilmente interpretados.

```
Var: Valor
Valor = ler()
Incrementar( Valor )
Imprimir( Valor )
```

Existem várias formas de usar, por exemplo, o comando `ler()`, como visto no seguinte exemplo. As linguagens de programação oferecem alguns meios de realizar a leitura de dados. Em alguns exemplos foi utilizado um operador de atribuição.

```
Valor = ler()
ler( Valor )
ler (v1, v2, v3)
v1, v2 = ler()
v3 ← ler()
```

Com o comando escrever() pode-se exibir informações geradas pelo algoritmo. Pode-se escrever dados de variáveis, mensagens ou combinação de ambos, como mostrado nos seguintes exemplos. Deve-se cuidar para não confundir a impressão de mensagens (entre “ ”) com a impressão de valores.

```
Var: a, b
a = 100
b = “alo mundo”
escrever ( a ) → isso imprime o valor 100
escrever ( “a” ) → isso imprime a letra “a”
escrever ( b ) → imprime a mensagem “alo mundo”
escrever (“o valor de a vale” + a) → concatena a mensagem com o valor de a
```

Pode-se utilizar o comando escrever para dar instruções ao usuário do que deve ser feito.

```
Var: nota
escrever ( “Digite o valor da nota” )
ler( nota )
escrever ( “A nota lida foi: ” + nota )
```

5. Seleção e Operadores

Quando se deseja realizar uma operação aritmética, pode-se fazer uso de operadores aritméticos. Os mais comuns são +, -, *, /, % (resto da divisão inteira), como ilustrado nos seguintes exemplos.

```
a = a + b
b ← a * a
b = a * 12.55 + 66
b = ler() + c * 22
e = a + b * c
```

Existem regras de precedência de operadores. No ultimo exemplo apresentado, na expressão a + b *c, a multiplicação tem precedência sobre a soma, neste caso então será primeiro realizada a multiplicação de b com c para então ser realizada a soma. A forma mais legível e garantida para evitar erros de precedência é com o uso de parênteses, como mostrado no seguinte exemplo.

```
e = a + (b * c)
e = (a + (b * c)) * 4
e = (ler() * 55) + ler()
```

A linguagem C e derivadas oferecem outros operadores aritméticos como ++, --, +=, dentre outros, como nos seguintes exemplos.

```
x++
x--
--x
a = 10
a = a++; //a = 10 pois a eh incrementado após a avaliação. ☹
a = ++a; //a = 11 pois a eh incrementado antes da avaliação. ☹
x += 2
y = x++
a *= 33
```

Todo algoritmo ou programa é baseado em expressões condicionais. É por meio delas que se pode tomar decisões em desvios condicionais e laços de repetição. Uma expressão condicional ou é verdadeira ou falsa. Não existe valor intermediário.

```
Se já é uma hora da tarde então
   Vá para o trabalho
Senão
   Vá assistir televisão
Fim se
```

Expressões condicionais podem também ser compostas de vários argumentos, conectados por operadores lógicos E, OU e NÃO, com no seguinte exemplo.

```
Se (já é uma hora da tarde) E (o dia estiver ensolarado) então
   Vá para o trabalho
Fim se
```

Neste exemplo, a expressão somente será verdadeira se (hora > 13) e (dia ensolarado). Caso qualquer um dos argumentos for falso, toda expressão será falsa. As seguintes tabelas mostram o resultado da expressão para operadores (com sintaxe da linguagem C).

Tipo do Operador	**Exemplo**	**Descrição**
Aritméticos		
+, -, *, /,	`f = a+b;`	Operações aritméticas convencionais
% (módulo)	`c = a%b;`	c recebe o resto da divisão inteira de a por b Ex: `4%2 = 0, 3%2 = 1, -1%2 = -1, -4%2 = 0`
Atribuição		
=	`a = 10;`	Inicializa a com 10
Incremento		
++	`cont++;`	Incrementa cont em 1
--	`cont--;`	Decrementa cont em 1
+=	`cont+=5`	Incrementa cont em 5
-=	`cont-=3`	Decrementa cont em 3. Equivale a `cont = cont - 3`
Relacionais		
=	`a=b`	Testa se a é igual a b
< , <=	`a<=b`	Testa se a é menor ou igual a b
>, >=	`a>b`	Testa se b é menor que a
!=	`a!=b`	Testa se a é diferente de b

Lógicos		
E	`a==1 && b==1`	E (and) lógico
OU	`a==1 || b==1`	Ou (or) lógico
Nao	`a=!a`	Negação (not)
xor		Ou exclusivo

Conjunção E

Condição 1	Condição 2	Resultado Lógico
F	F	F
F	V	F
V	F	F
V	V	V

Conjunção Ou

Condição 1	Condição 2	Resultado Lógico
F	F	F
F	V	V
V	F	V
V	V	V

Conjunção Não

Condição	Resultado Lógico
V	F
F	V

Conjunção Ou Exclusivo

Condição 1	Condição 2	Resultado Lógico
F	F	F
F	V	V
V	F	V
V	V	F

Geralmente os testes lógicos são aplicados sobre variáveis do algoritmo, como nos seguintes exemplos.

```
Inicio
   Var: a, b, c
   a = 10
   escreva("digite um valor")
   b = Ler( )
   c = a + b
   Se (a++ > --b ) então  //não recomendado escrever código bizarro ☹
      Escrever("a > b")
   Senão
      Escrever ( b )
   Fim se
   Se (b = 10 E b < c) então
      Escrever ("b vale", b)
   Fim se
   Fim.
```

O seguinte exemplo mostra um algoritmo para ler e calcular a área de um quadrado e dizer se esta área é maior ou menor que 30. Neste exemplo, base, altura e área são variáveis que armazenam valores numéricos. Deve-se observar a **identação** dos comandos no caso de desvios condicionais.

```
Início
   Var: base, altura, area
   Escrever("digite o valor da base e da altura")
   Ler base, altura
   area = base * altura
   Se area < 30 Então
      Escrever "a área é menor que 30 e vale ", area
   Senão
      Escrever ("a área é maior ou igual a 30")
   Fim se
Fim.
```

No exemplo a seguir são apresentadas duas formas de fazer o mesmo algoritmo, porém a segunda forma é mais compacta e correta que a primeira, visto que evita a replicação de código.

```
Início
   Var: n, resp
   Ler n
   Se n < 10 Então
      resp = n * n
      Escrever resp
   Senão
      resp = n + n
      Escrever resp
   Fim se
Fim.
```

```
Início
   Var: n, resp
   Ler n
   Se n < 10 Então
      resp = n * n
   Senão
      resp = n + n
   Fim se
   Escrever resp
Fim.
```

Pode-se fazer encadeamentos grandes com a estrutura **se-senão**, como no seguinte exemplo. Observe que o último caso não possui teste condicional; este caso é considerado default. Caso nenhum teste seja verdadeiro, será executado este caso. Observe que será impressa uma **ÚNICA** mensagem:

```
Inicio
   Var: media
   Ler media
   Se media > 9 então
      Escreva "conceito A"
   Senão se media > 8 então
      Escreva "conceito B"
   Senão se media > 7 então
      Escreva "conceito C"
   Senão
      Escreva "conceito D – Reprovado"
   Fim se
Fim.
```

É muito comum também o uso de testes condicionais encadeados, como no seguinte exemplo:

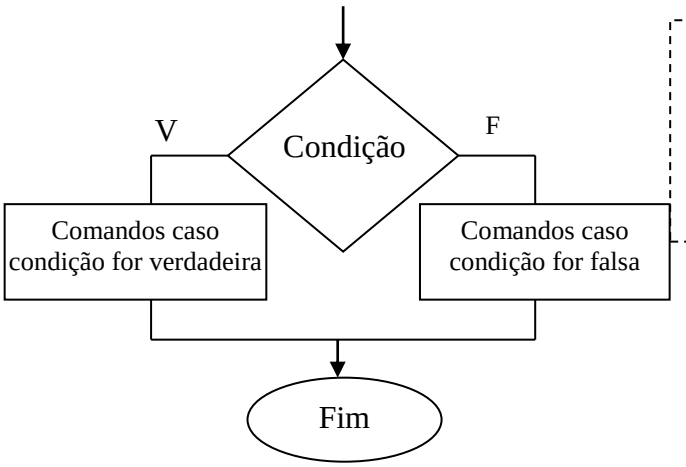
```
Inicio
   Var: A, B, C, D
   A = 10
   Ler B, C, D
   Se A > B então
      Escreva "A é maior que B"
      Se B > C então
         Escreva B    → Neste caso é impresso o valor de B
         Se B = D então
            Escreva "B é igual a D"
         Fim se
      Fim se
      Enquanto A > B repita
         A = A - 1
      Fim enquanto
      Escreva "Agora A é igual a B"
   Fim se
   Escreva "fim do programa"
Fim.
```

O seguinte algoritmo ilustra o cálculo para aprovação em disciplinas da UFSM. O cálculo é feito sobre duas notas (duas avaliações) e uma terceira nota, caso o aluno não seja aprovado por media (pegue exame).

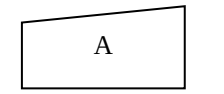
```
Inicio
   Var: n1, n2, media, exame
   Escrever("Digite as duas notas")
   Ler( n1, n2)
   media = (n1 + n2) / 2
   Se ( media >= 7 ) então
      Escrever("aprovado com nota " media)
   Senão
      Escrever("Digite a nota do exame")
      Ler(exame)
      media = (media + exame) /2

      Se media >= 5 então
         Escrever ( "Aprovado com nota" media )
      Senão
         Escrever ( "Reprovado com nota" media )
      Fim se
   Fim se
Fim.
```

5.1 Outras notações gráficas para representação

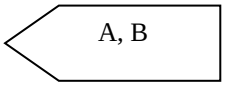
A notação gráfica faz uso de fluxogramas para descrever as ações do algoritmo. Setas indicam em qual direção a execução deve seguir. O caminho a ser seguido é definição por desvios condicionais. Nos seguintes exemplos são apresentados formas de representação de tomada de decisões. Outros elementos desta representação podem ser encontrados em [1].



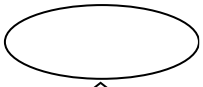

```
Se Condição verdadeira Então
   Executar BlocoVerdadeiro
Senão
   Executar Bloco Falso
Fim Se
```



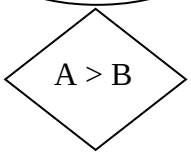
Usado para leitura. Ex: A = ler()



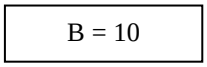
Usado para impressão. Ex: Escreva (A, B)



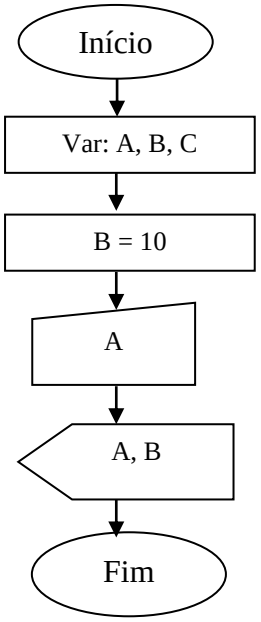
Demarca início e fim de programa



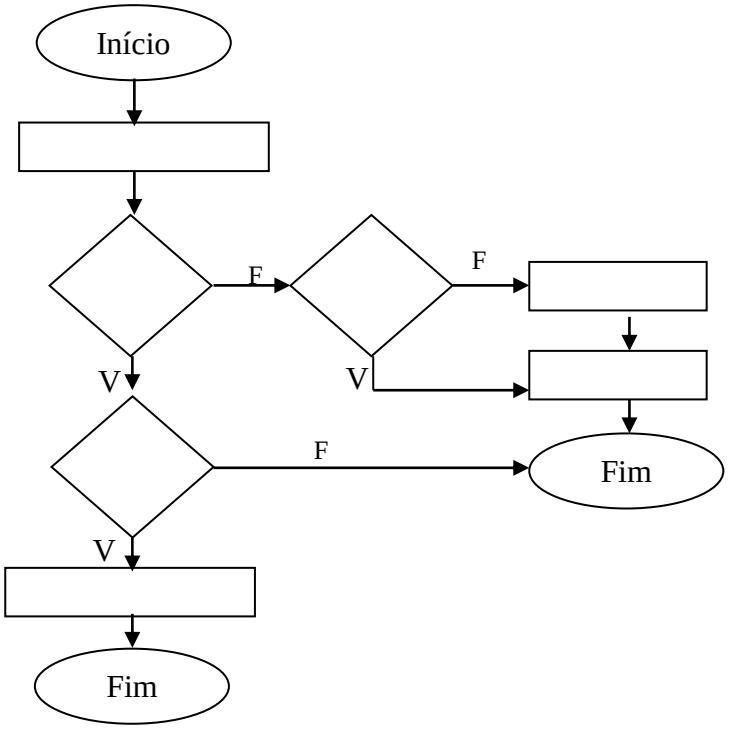
Representa um teste condicional com duas saídas: V ou F



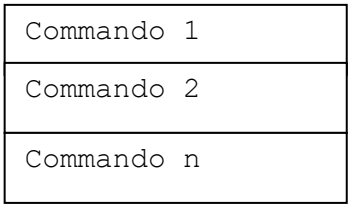
Processamento

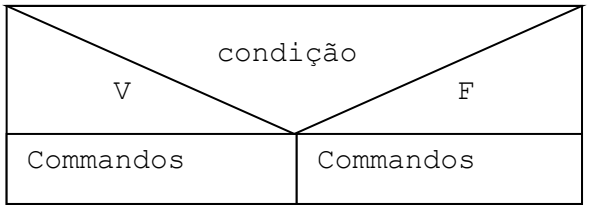


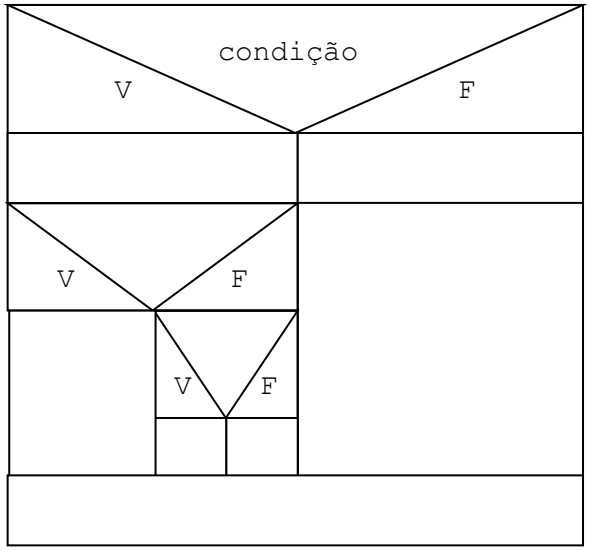
No seguinte exemplo tem-se encadeamentos de testes condicionais, e o fim do programa ilustrado em dois locais distintos.

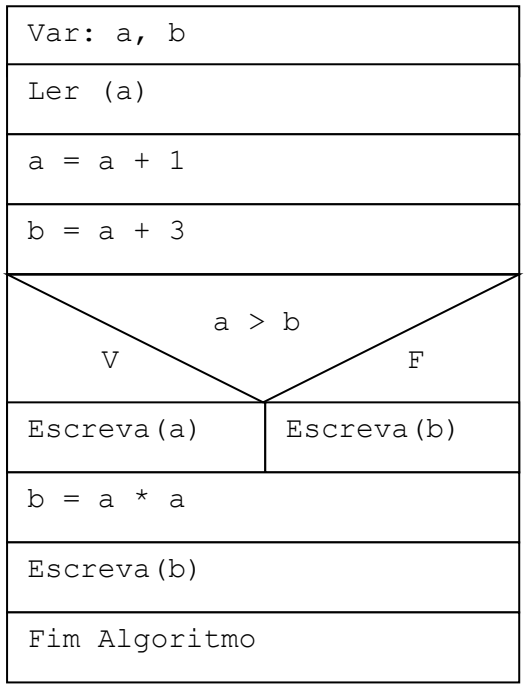


Outra forma de representação é por meio de Diagramas de Nassi-Shneiderman. Esse diagrama permite uma visão estruturada e hierárquica do programa. Faz uso de um único retângulo, que pode ser subdivido em retângulos menores. Permite a representação de comandos, seleção e laços de repetição.

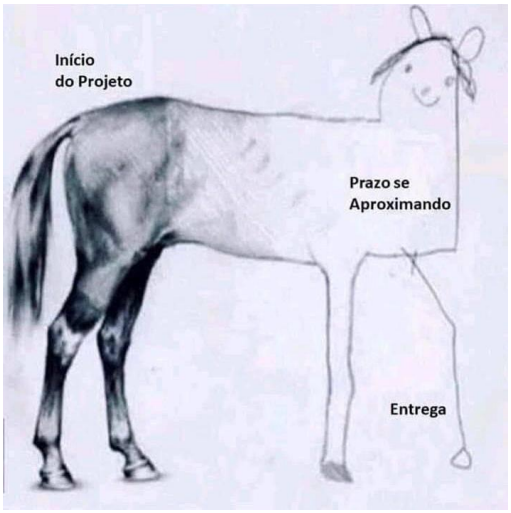






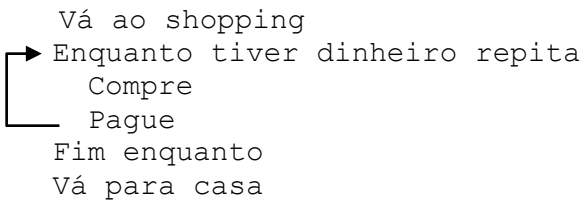


Exercícios: ver lista de exercícios



6. Laços de Repetição

Quando precisamos que alguma operação seja realizada várias vezes, devemos fazer uso de laços de repetição. Da mesma forma como um teste condicional, um laço de repetição está associado a um bloco de comandos que deve ser executado enquanto a condição for verdadeira, com no seguinte exemplo. As operações comprar e pagar são executadas sequencialmente enquanto a pessoa tiver dinheiro. Quando o dinheiro acabar, a pessoa deve ir para casa. O comando usado para laços de repetição é o **enquanto**.



A estrutura básica do laço `enquanto` é:

```
Inicialize variável
Enquanto variável for verdadeira repita
    Incremente variável
Fim enquanto
```

Agora vamos transformar o algoritmo de compras no shopping em uma pseudo-linguagem. Para isso, devemos transformar os dados utilizados em variáveis. De imediato, devemos ter a variável dinheiro, que armazena a quantidade de dinheiro que a pessoa possui. Esse valor deve ser lido de algum lugar (teclado, banco de dados, etc.), pois o programa não pode "adivinhar" quanto dinheiro uma pessoa qualquer possui. Uma primeira versão muito simplificada e incompleta do algoritmo pode ser a seguinte:

```
Inicio
    Ler dinheiro
    Enquanto dinheiro > 0 repita
        Compre item (equivalente a gaste dinheiro)
        Pague pelo item
    Fim enquanto
Fim.
```

Neste exemplo, as operações comprar e pagar não podem ser compreendidas por um programa de computador. A operação pagar deve alterar a quantidade de dinheiro que a pessoa possui. A operação comprar somente pode ser realizada caso a pessoa tiver dinheiro suficiente para comprar o determinado item, senão pode ficar no negativo, o que não é permitido neste problema. Agora, uma versão um pouco mais correta. Observe neste exemplo a identação entre os blocos **se** e **enquanto**.

```
Inicio
    Var: dinheiro, Valor_item
    Escrever "va ao shopping"
    Ler dinheiro
    Enquanto dinheiro > 0 repita
        Ler valor_item
        Se dinheiro >= valor_item então
            Escrever "compre o item"
            dinheiro = dinheiro – valor_item
        Fim se
    Fim enquanto
    Escrever "va para casa"
Fim.
```

Cada vez que algo for comprado, faz-se uma diminuição do valor da variável dinheiro igual ao valor do item a ser comprado, que também deve ser lido de algum lugar. Neste programa, o comando Ler abstrai esta leitura. Este algoritmo está quase correto, pois uma característica de um algoritmo é que ele sempre deve ser finito. O que aconteceria se o valor de todos os itens de uma loja fosse maior que o dinheiro que a pessoa possui, ou se não existisse nenhum item que custasse exatamente o dinheiro que a pessoa possui? Para estes casos, que são muito comuns de ocorrer, o algoritmo não terminaria nunca. Para isso, usa-se a expressão "Entrou em Loop".

Para solucionar este problema, uma estratégia simples seria finalizar o programa após um determinado número de tentativas de compra não realizadas. Eis uma versão desta estratégia, que após 10 tentativas inválidas, finaliza a execução. Observe que a variável contador foi inicializada com valor zero. Outra variação seria a contagem de **10 tentativas sucessivas**. Faça essa modificação.

```
Inicio
    Var: dinheiro, valor_item, contador
    Ler dinheiro
    contador = 0
    Enquanto (dinheiro > 0 E contador < 10) repita
        Ler valor_item
        Se dinheiro >= valor_item então
            Escrever "compre o item"
            dinheiro = dinheiro – valor_item
        Senão
            contador = contador + 1
        Fim se
    Fim enquanto
Fim.
```

Neste exemplo, a variável contador é usada para armazenar o número de vezes que a compra não foi realizada. Quando chegar a 10 ou quando não tiver mais dinheiro, o programa finaliza a execução. Observe que no teste condicional do enquanto é utilizado o operador E. Analise o que aconteceria se fosse usado o operador OU. Como exercício, altere este algoritmo para que ele somente pare após 10 tentativas inválidas consecutivas de compra de algum item, ou seja, entre as tentativas inválidas não pode haver nenhuma compra.

Um exemplo simples e prático da utilização de laços de repetição é apresentado no seguinte exemplo, que imprime os números inteiros de 0 a N, sendo N informado pelo usuário. Deve-se observar a **identação** dos comandos no caso de laços de repetição.


```
Inicio
    Var: N, cont, incremento, valor = 0
    N = Ler( )
    incremento = Ler()
    cont = 0
    Enquanto cont <= N repita
        Escrever( valor )
        cont = cont + 1
        valor = valor + incremento
    Fim enquanto
Fim.
```

O seguinte programa faz a soma dos N primeiros números inteiros, sendo N informado pelo usuário.

```
Inicio
   Var: N, cont, soma
   N = Ler( )
   cont = 1
   soma = 0
   Enquanto cont <= N repita
   ¦  soma = soma + cont
   ¦  cont = cont + 1
   Fim enquanto
   Escrever( soma )
Fim.
```

Pode-se também utilizar a estrutura de repetição "**para**" de forma similar. Deve-se utilizar este comando quando se sabe de antemão o número de interações que vão ser necessárias. Por convenção, neste texto, no comando `Para` a variável de controle não é incrementada automaticamente em 1. Desta forma deve-se dentro do laço fazer o incremento apropriado da variável.

```
Inicio
   Var: N, cont, soma
   N = Ler( )
   soma = 0
   Para cont = 1 até N repita //enquanto cont <= N
   ¦  soma = soma + cont
   ¦  cont = cont + 1 → cont não é incrementado por default
   Fim para
   Escrever( soma )
Fim.
```

Outra forma para representar laços é por meio da estrutura repita-enquanto. Esta deve ser usada quanto deve-se processar o laço antes de realizar qualquer teste, como mostrado no seguinte exemplo.

```
Inicio
   Var: senha, entrada
   senha = 12345
   Repita
   ¦  escrever( "digite a senha")
   ¦  ler (entrada)
   Enquanto (entrada != senha)
   Escrever( "acesso autorizado" )
Fim.
```

Pode-se também utilizar operadores lógicos dentro de testes condicionais. O seguinte exemplo ilustra um programa para tentar descobrir com 5 tentativas um valor predefinido. O símbolo # é usado para indicar comentários.

```
Inicio
   Var: valor, tentativa, Max_tentativas, Num_tentativas
   Max_tentativas = 5
   Num_tentativas = 0
   valor = 20   #valor que o usuário devera tentar descobrir
   Ler tentativa
   Enquanto tentativa != valor E Num_tentativas < Max_tentativas repita
      Ler tentativa
      Num_tentativas = Num_tentativas + 1
   Fim enquanto
   Se tentativa == valor então
      Escrever ("você acertou o numero")
   Senão
      Escrever ("você não acertou o numero")
   Fim se
Fim.
```

Programas mais elaborados podem requerer laços de repetição aninhados, ou seja, um laço de repetição dentro de outro laço de repetição, como mostrado no seguinte exemplo (versão com **para** e versão idêntica com laço **enquanto**):

```
Inicio
   Var: a, b, N = 4
   Para a = 1 até N repita
      Para b = a até N repita
         Escrever(b)
         b = b + 1
      Fim para
      a = a + 1
   Fim para
Fim.
```

```
Inicio
   Var: a, b, N = 4
   a = 1
   enquanto a <= N repita
      b = a
      enquanto b <= N repita
         Escrever(b)
         b = b + 1
      Fim para
      a = a + 1
   Fim para
Fim.
```

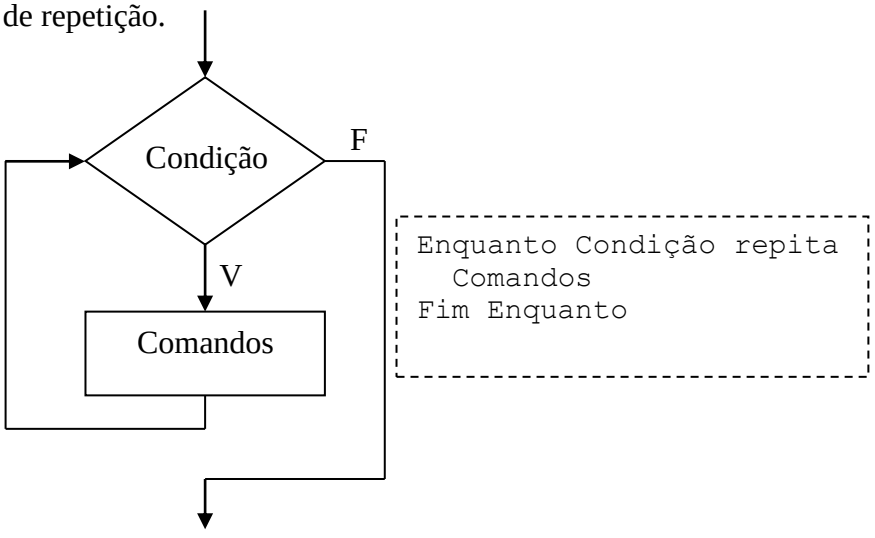
Para N = 4, esse programa imprime a sequência 1 2 3 4 2 3 4 3 4 4.

Deve-se observar com atenção a variação dos valores das variáveis **a** e **b** com o uso do comando **para**.

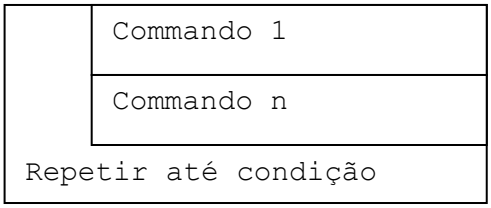
- A variável a é inicializada uma única vez, quando o laço mais externo é executado.
- A variável b é inicializada toda vez que o laço externo executa, mas não quando o laço interno executa.

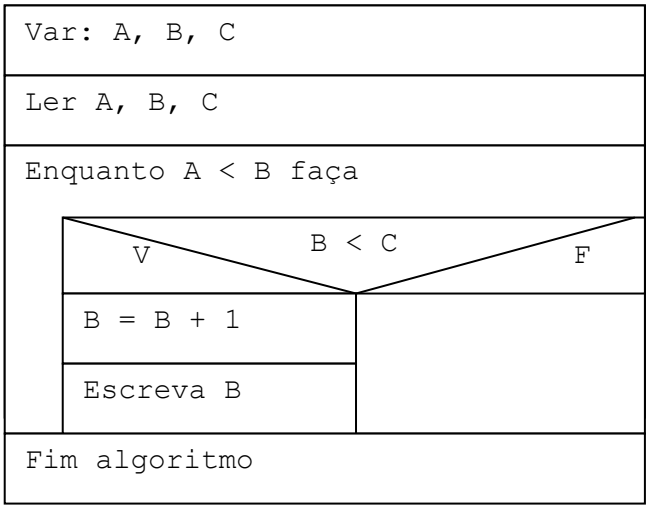
6.1 Outras notações gráficas para representação

Na representação de fluxograma, setas indicam em qual direção a execução deve seguir. O caminho a ser seguido é definição por desvios condicionais. Nos seguintes exemplos são apresentados formas de representação de laços de repetição.



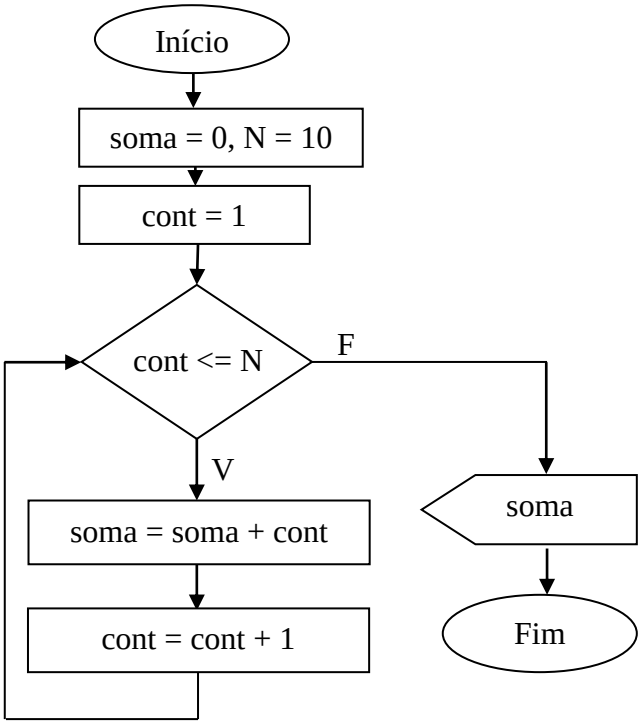
Outra forma de representação é por meio de Diagramas de Nassi-Shneiderman. Esse diagrama permite uma visão estruturada e hierárquica do programa. Faz uso de um único retângulo, que pode ser subdivido em retângulos menores. Permite a representação de comandos, seleção e laços de repetição.





O seguinte exemplo apresenta uma forma detalhada de representar um laço de repetição utilizando-se fluxogramas.

```
Inicio
   Var: cont, soma, N = 10
   soma = 0
   Para cont = 1 até N repita
      soma = soma + cont
      cont = cont + 1
   Fim enquanto
   Escrever( soma )
Fim.
```



Alguns exercícios

1) programa para imprimir os números inteiros de 1 a 1000 usando enquanto e para.
2) programa para imprimir os números inteiros de 1 até um numero fornecido pelo usuário
3) programa para imprimir os números pares inteiros de 1 a N
4) programa para imprimir a sequencia 1, 100, 2, 99, 3, 98, ... 50, 51. Faça uma versão para N sendo ímpar.
5) programa para imprimir os números inteiros entre dois valores fornecidos pelo usuário. A sequência pode ser crescente ou decrescente, dependendo dos valores fornecidos.
6) programa para imprimir a sequencia 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1 **(com 1 e com 2 laços de repetição).**

Mais difíceis

1) Calcular o fatorial de um número
2) Imprimir os primeiros N elementos da sequencia de Fibonacci. 0, 1, 1, 2, 3, 5, 8, 13, etc.
3) Imprimir a sequência: 1 1 2 2 1 3 3 2 1 4 4 3 2 1 5 5 4 3 2 1 6 ...

Exercício Resolvido: programa para imprimir a sequência 1, 10, 2, 9, 3, 8, 4, 7, 5, 6. As seguintes soluções tratam o caso de N ser par, em versões Novato, Expert e Guru.

Novato

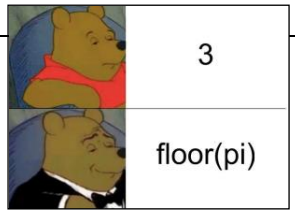
```
Inicio
   Var: N = 10, cont
   Para cont = 1 ate N/2 repita
      Escreva( cont )
      Escreva( N - cont + 1 )
      cont = cont + 1
   Fim para
Fim.
```

Expert

```
void main()
{
   int N = 10;
   int resp = N + 1;
   int i;
   for(i = N; i >= 1; i--)
   {
      resp = resp + pow(-1, !(i & 1)) * i;
      printf("%d\n", resp);
   }
}
```

Guru

```
Inicio
   Var: N = 10, a=1, b=N
   Enquanto a < b repita
      Escreva( a )
      Escreva( b )
      a++
      b--
   Fim enquanto
Fim.
```



7. Funções

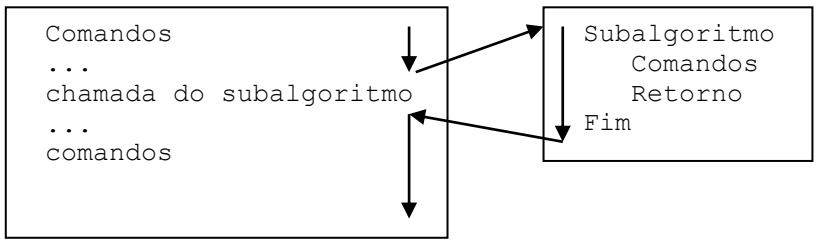
Para melhor estruturar o algoritmo e facilitar reutilização de código, uma boa prática é o uso de subalgoritmos (procedimentos ou funções). Uma função nada mais é que outro algoritmo que é chamado por um algoritmo para desempenhar uma tarefa específica.

Para funções que requerem um maior número de linhas de código a vantagem é mais visível ainda. Além da redução de código, outra característica é mais vantajosa ainda. Considere o caso de um trecho de código extenso, complexo e redefinido várias vezes em um mesmo programa. Se for detectado algum erro nesse código, deve-se fazer a atualização em todas as ocorrências do código, o que toma um tempo enorme e pode ser susceptível a erros ou esquecimentos, acarretando em erros que são extremamente difíceis de serem encontrados.

No seguinte exemplo, define-se a função média, que retorna a média de dois números. Esta função é chamada pela função `principal()`, que é obrigatória em qualquer programa. Para chamar uma função, deve-se escrever o seu nome. Funções podem receber parâmetros ou não. No seguinte exemplo, a função `media()` recebe como argumento dois valores que devem ser somados e divididos por dois. O comando `retorne` é usado para indicar que a função finalizou.

Existem dois tipos de funções: as que retornam e as que não retornam valores. No seguinte exemplo, a função está retornando o valor da variável `M`. Este valor é então atribuído as variáveis `C` e `D`, presentes na função principal, um para cada chamada da função `media()`.

```
Função Media(n1, n2)
   Var: M
   M = (n1+n2)/2
   Retorne M
Fim Função

Início
   Var: A, B
   Ler A, B
   C = Media(A, B)
   D = Media(30, 11)
   Escrever (C, D)
Fim.
```



O comando Retorne é usado para parar a função. Nenhum comando localizado após o `Retorne` é executado, como no seguinte exemplo. Caso esse comando não seja executado, ao final da função sempre existe um retorno implícito, mesmo que o usuário não o coloque.

```
Função f1(valor)
   Se valor < 10 então
      Retorne
   Senão
      Imprime (valor*valor)
   Fim se
Fim Função
```

```
Função f1(valor)
   Se valor < 10 então
      Retorne
   Senão
      Imprime (valor*valor)
   Fim se
   Retorne → comado implicito
Fim Função
```

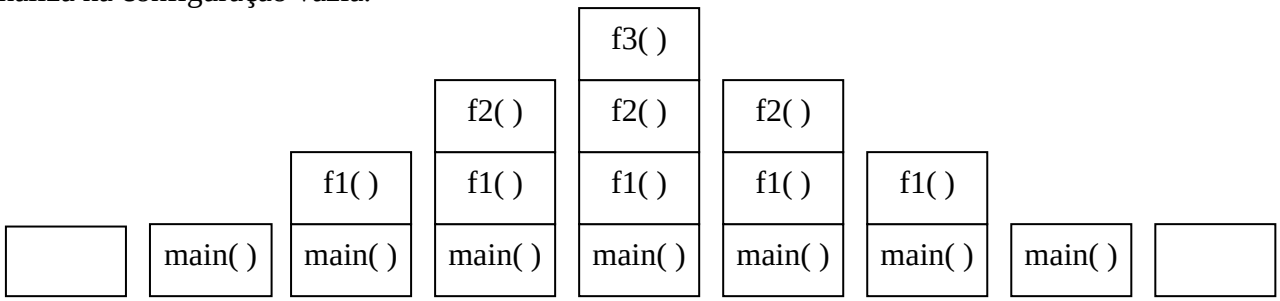
Existem dois tipos de funções: as que retornam valores e as que não retornam. No exemplo anterior, o comando `Retorne` é usado simplesmente para finalizar a função. No seguinte exemplo, a função deve obrigatoriamente retornar algum valor.

```
Função f2(valor)
   Se valor < 10 então
      Retorne valor
   Senão se valor < 20 então
      Retorne (valor*valor)
   Fim se
   Retorne 1
Fim Função
```

Uma função pode chamar outra função sem limite de chamadas, como no seguinte exemplo. Neste exemplo será impresso o resultado `1324`, pois a função principal chama a função `f1()`, que imprime 1. Após, a função `f1()` chama a função `f2()`, que chama a função `f3()`. A função `f3()` imprime 3 e retorna para a função que a chamou, neste caso a função `f2()`. Após a chamada da função `f3()`, a função `f2()` imprime o valor 2 e retorna para a função `f1()`, que escreve o valor 4 e retorna para a função principal. Neste exemplo, as funções não tem valor de retorno. Quando todos os comandos de uma função são executados, a função retorna para o ponto onde foi chamada, mesmo que não exista um comando explícito `retorne`. Neste exemplo, por questões didáticas, foi adicionado o comando retorne ao final de cada função.

```
Função f3()
   Escrever(3)
   retorne
Fim Função

Função f2()
   f3()
   Escrever(2)
   retorne
Fim Função

Função f1()
   Escrever(1)
   f2()
   Escrever (4)
   retorne
Fim Função

Início
   f1()
   retorne
Fim.
```

Chamadas de funções estão associadas a uma pilha (gerenciada pelo sistema operacional), que guarda as funções que foram chamadas para poder restaurar a configuração anterior quando a função retornar. Uma pilha tem a seguinte propriedade: somente pode ser removido o elemento que está no topo e todo novo elemento deve ser adicionado ao topo (LIFO – *Last in, First out*). Além de referências para as funções chamadas, também são armazenadas todas as variáveis locais às funções chamadas. Para o exemplo anterior, tem-se a seguinte evolução da pilha durante a execução do programa. A pilha inicia e finaliza na configuração vazia.



Quando o "computador" inicia, a pilha está vazia. Quando o programa começa a executar, é carregada na pilha a função principal (`main()`). Quando a função `f1()` é chamada, esta é empilhada na pilha, e assim sucessivamente. Quando uma função finaliza, esta é removida da pilha e toma-se a função logo abaixo como sendo a função em atual execução, com suas variáveis locais restauradas com o valor antes da chamada da função. Quando a pilha esvaziar, significa que o programa finalizou a execução.

As variáveis presentes nas funções são consideradas locais. Isso significa que duas funções podem ter variáveis com os mesmos nomes, porém serão consideradas variáveis distintas. Somente podem ser acessadas variáveis da função que estiver em execução.

Sabemos que uma função retorna para a função que a chamou, porém uma questão que deve ser bem observada é saber qual vai ser a próxima instrução executada após o seu retorno. No seguinte exemplo, é executado o próximo comando após a chamada, ou seja, o comando `escrever("fim")`.

```
Início
   f()
   escrever("fim")
Fim.
```

```
Função f()
   Escrever("função f")
Fim Função
```

No próximo exemplo, pelo fato da função retornar um valor, o próximo comando a ser executado será a atribuição do valor retornado a uma variável, neste caso o comando `a = 30`.

```
Início
   Var: a
   a = f()
   escrever(a)
Fim.
```

```
Função f()
   Retorne 30
Fim Função
```

Caso o retorno de uma função não for tratado, ele é ignorado. No seguinte exemplo, será impresso o valor 4, pois o retorno da função não foi atribuído a variável `x` local a função principal.

```
Início
   Var: x
   x = 4
   duplica(x)
   escrever(x)
Fim.
```

```
Função duplica(x)
   Retornar (x * 2)
Fim Função
```

No seguinte código, será impresso o valor 8, pois o retorno da função foi atribuído à variável `x`.

```
Início
   Var: x
   x = 4
   x = duplica(x)
   escrever(x)
Fim.
```

Funções podem ou não receber argumentos. Quando uma função não recebe argumentos, na declaração e na chamada basta apenas declarar a função.

Início
   f()
Fim.
	Função f()
	   Escrever ("função f")
	Fim Função

Quando uma função recebe argumentos, deve-se declarar uma variável para cada argumento passado. Cada variável declarada para receber argumento é considerada local a função. No seguinte exemplo, a função `f()` recebe 3 argumentos. Na chamada da função deve-se passar também 3 argumentos. O primeiro argumento passado é associado à primeira variável declarada na função, o segundo associado a segunda e assim por diante. Para o exemplo, o valor de `x` é atribuído à variável `a`.

Início
   Var: x = 1, y = 2, z = 3
   f(x, y, z)
Fim.
	Função f(a, b, c)
	   Escrever (a, b, c)
	Fim Função

Não se pode acessar variável definidas dentro de uma função em outra função. Considerando-se o exemplo anterior, seria um erro usar, por exemplo, a variável `a` dentro da função principal como usar `x` dentro da função `f()`.

Início
   Var: x = 1, y = 2, z = 3
   a = 100 → erro: a não definido
   f(x, y, z)
Fim.
	Função f(a, b, c)
	   Escrever (a, b, c)
	   y = a + b → erro: y não definido
	Fim Função

Pode-se ter exemplos mais complexos com o uso de funções. A função `soma()` está sendo chamada 3 vezes. A primeira chamada é a `soma(1,2)`. O retorno é usado como parâmetro para a chamada `soma(3,3)` e posteriormente a chamada para `soma(6,4)`. O resultado desta última chamada é usado como argumento para a função `escrever(10)`. A execução ocorre desta forma, pois uma função somente pode ser chamada quando se conhece seus argumentos.

Início
   escrever(soma(soma(soma(1,2), 3), 4))
Fim.
	Função soma(a, b)
	   Retorne (a + b)
	Fim Função

Passo a passo, esta chamada se resume a:

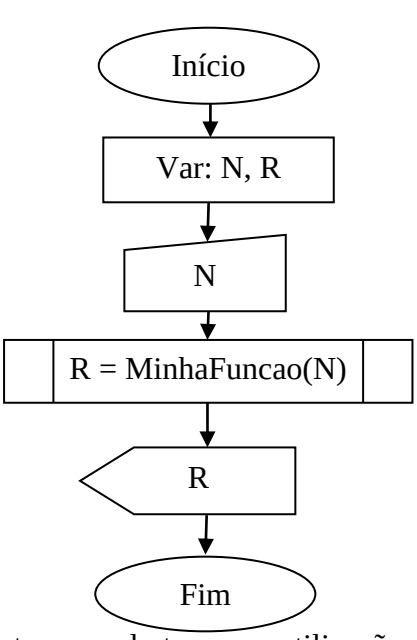
```
escrever( soma( soma( soma(1,2), 3), 4) )
escrever( soma( soma(3, 3), 4) )
escrever( soma(6, 4) )
escrever( 10 )
```

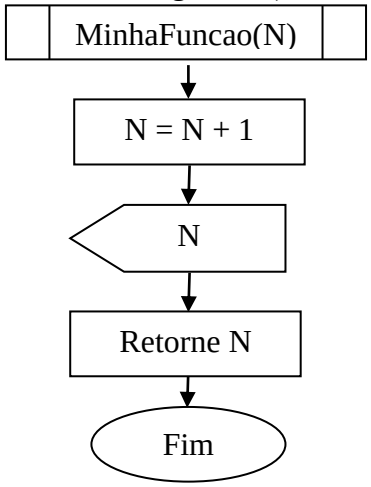
Este algoritmo pode também ser reescrito das seguintes formas. Qual delas é a melhor?

```
Início
   Var: a, b, c
   a = soma(1,2)
   b = soma(a,3)
   c = soma(b,4)
   escrever( c )
Fim.
```

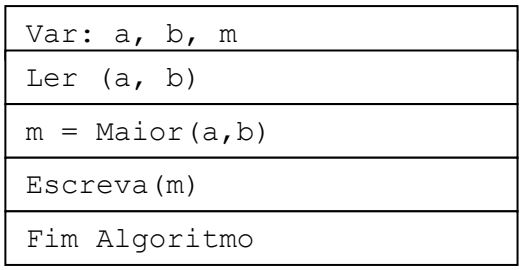
```
Início
   Var: a, b, c
   a = soma(1,2)
   b = soma(3,4)
   c = soma(a,b)
   escrever( c )
Fim.
```

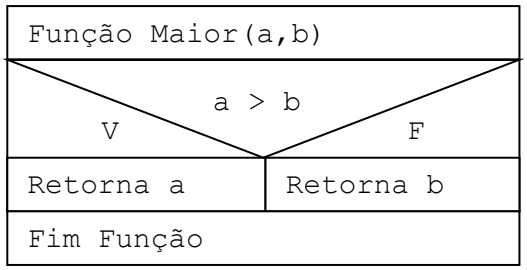
No seguinte exemplo tem-se a representação de funções em fluxograma (chamada e definição da função).





No seguinte exemplo tem-se a utilização de diagramas para representação de funções





Exercício em Aula: Implemente uma função para ler dois inteiros i e f, sendo i < f, e uma indicação de impressão dos números impares ou pares. A função deve chamar a função imprime() que escreve na tela apenas os números pares ou impares da sequência, conforme escolha do usuário. A determinação se os números são pares ou impares deve estar dentro de outra função.

7.1 Funções Recursivas

Uma função é dita **recursiva** quando ela a chama de forma direta ou indireta. Soluções que fazem uso de laços de repetição podem ser implementadas de forma recursiva, como no seguinte exemplo, que imprime os números inteiros de 1 a 10.

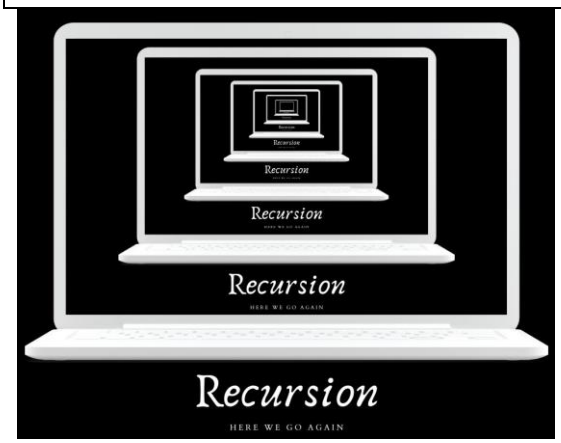
```
Início
   imprimeRec( 1 )
Fim.
```

```
Função imprimeRec(valor)
   Se( valor > 10 ) entao
      Retorne
   Fim se
   escrever ( valor)
   imprimeRec( valor + 1 )
Fim Função
```

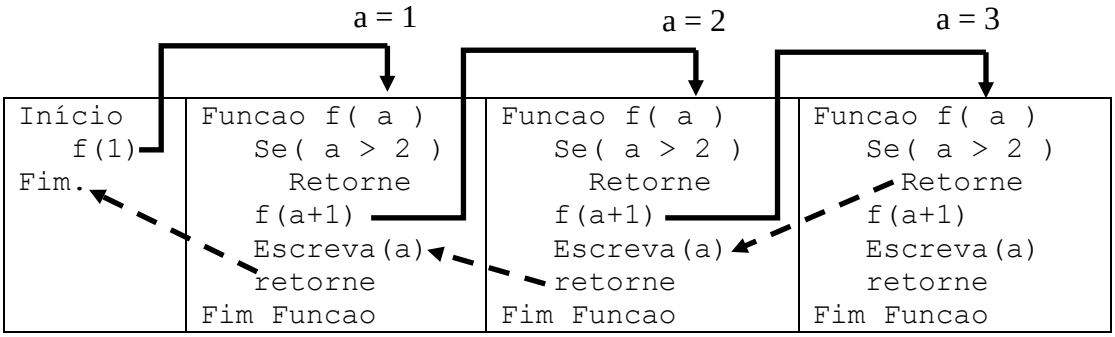
Para imprimir os números em ordem decrescente, basta mudar a posição do comando escrever.

```
Início
   imprimeRec( 1 )
Fim.
```

```
Função imprimeRec(valor)
   Se( valor > 10 ) entao
      Retorne
   Fim se
   imprimeRec( valor + 1 )
   escrever ( valor)
Fim Função
```



O seguinte diagrama ilustra o funcionamento de uma função recursiva que imprime os números inteiros decrescentes de 2 a 1. As linhas sólidas representam chamadas e as linhas pontilhadas o retorno. Por questões didáticas, é feita uma cópia da função cada vez que ela é chamada. Como exercício, faça diagramas de execução semelhantes para os demais exemplos apresentados nesta seção. Sempre que uma função retorna, a execução volta para o próximo comando após a chamada da função. Para o exemplo, após a chamada recursiva da função, o próximo comando é Escreva(a).



Toda função recursiva deve ter um **teste de parada**. No exemplo anterior o teste (valor>10) faz com que a função retorne quando o argumento passado ultrapassar 10. Observe que a cada chamada o valor do argumento é aumentado em 1. A função retorna para o local onde foi feita a última chamada, que pode ser a própria função.

Pode-se também elaborar algoritmos complexos com o uso de funções recursivas, como no seguinte exemplo. Faça uma depuração e diga qual será a resposta impressa por este algoritmo.

```
Início
   imprime( 1 )
Fim.
```

```
Função imprime(valor)
   Se valor = 5 então
      Retorne
   Fim se
   Escrever(valor)
   imprime(valor + 1)
   imprime(valor + 1)
Fim Função
```

Como qualquer outra função, pode-se passar um número qualquer de argumentos para uma função, como no seguinte exemplo que imprime os números inteiros de 1 a N, sendo N lido.

```
Início
   Var: N
   Ler N
   rec( 1, N )
Fim.
```

```
Função rec(valor, limite)
   Se valor > limite então
      Retorne
   Fim se
   Escrever(valor)
   imprime(valor + 1, limite)
Fim Função
```

Os casos mais complexos de recursão ocorrem quando a função deve retornar valores. No seguinte exemplo, calcula-se a soma dos números de 1 a 10 de forma recursiva. Deve-se sempre armazenar o valor retornado pela chamada recursiva. Neste exemplo, a variável **res** contem os seguintes valores em ordem cronológica: 10, 10+9, 10+9+8, etc. De forma semelhante, pode-se implementar uma função para calcular o fatorial de um número. Observe que com o uso de funções recursivas pode-se evitar o uso de laços de repetição. Porém, geralmente a solução que faz uso de laços é mais eficiente que a solução recursiva, visto que não necessita alterações na pilha do sistema. Funções recursivas são geralmente utilizadas em algoritmos para manipulação de árvores, como será visto na disciplina de Estruturas de Dados.

Versão recursiva

```
Função Soma(valor)
   Var: res = 0
   Se valor = 10 então
      Retorne 10
   Fim se
   res = res + Soma( valor + 1 )
   Retorne res
Fim Função
```

Não recursiva

```
Função Soma(valor)
   Var: res, cont
   res = 0
   Para cont = 1 até 10 repita
      res = res + cont
      cont = cont + 1
   Fim Para
   Retorne res
Fim Função
```

```
Início
   Escrever( Soma(1) )
Fim.
```

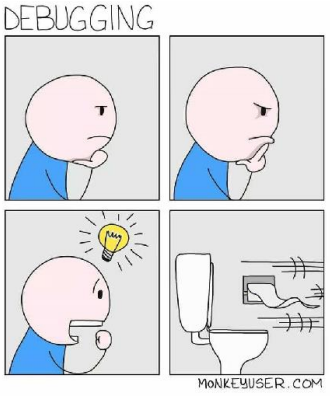
Funções recursivas que retornam valores devem retornar alguma coisa em todos os casos. O seguinte exemplo hipotético ilustra um erro (destacado em negrito).

```
Função F(valor)
   Se (valor == 10 ) entao
      Retorna 10
   Fim se
   Se (valor > 0 ) entao
      Retorna F( valor + 1 )
   Fim se
   Retorna
Fim Função
```

Pode-se também fazer uma solução que incrementa a media que a recursão é chamada. Assim que a recursão parar, o resultado é impresso. Observe que essa função não tem retorno.

```
Início
   Soma(1, 2) )
Fim.
```

```
Função Soma(valor, cont)
   Var: resp = valor + cont
   Se cont = 10 então
      Imprime(resp)
   Fim se
   Soma( resp, cont + 1)
Fim Função
```



Exercícios sobre funções recursivas (em ordem crescente de complexidade)

1. Imprimir os números inteiros de 1 a 20
2. Imprimir os números pares de 2 a 10
3. Imprimir os números pares de 10 a 2
4. Imprimir a sequência 1, 2, 3, 4, 5, 6, 5, 4, 3, 2, 1
5. Imprimir a sequência 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 8, 6, 4, 2
6. Função para calcular a somatória de N termos. S = 1 + 2 + 3 + ... + N
    a. Com dois argumentos na função
    b. Com 1 argumento na função
7. Função para o cálculo do fatorial de um número, em várias versões:
    a. F = 5*4*3*2*1 (multiplicação após retorno da função)
    b. F = 1*2*3*4*5
    c. A impressão (ou retorno) do valor deve ocorrer no momento da parada da recursão.
8. Função fatorial modificada para N termos: F=1*2*3*4*5*5 (o último número repete)
9. Função para calcular:
    a. S = 5*1 + 4*2 + 3*3 (para N impar)
    b. S = 6*1 + 5*2 + 4*3 (para N par)
    c. Solução para qualquer valor de N
10. Função F = 1 + 3 + 5 + 7 + 9 + 8 + 6 + 4 + 2 (somados nesta ordem)
11. Função F = (1*2*3*4*5*6*7*8*...*N) – N – (N–2) – ... – 6 – 4 – 2
12. Função fatorial modificada para N termos. Não podem ser criadas funções auxiliares. Elabore soluções com 1 e 2 chamadas recursivas. Neste exemplo N=5.
    a. F = 1 * 2*2 * 3*3*3 * 4*4*4*4 * 5*5*5*5*5.
    b. F = 5*5*5*5*5 * 4*4*4*4 * 3*3*3 * 2*2 * 1.

8 – Referências Bibliográficas

[1] Celes, W., Cerqueira, R., Rangel, J. Introdução a Estruturas de Dados. Ed. Campus, 2004.

[2] Mokarzel, F., Soma, N. Introdução a Ciência da Computação. Editora Campus. 2008, 429p.

[3] Guimarães, A. M., Lages, N. A. C. Algoritmos e estrutura de dados. Livros Técnicos e científicos. RJ. 216p. 1985.

[4] Veloso, P., et al. Estruturas de Dados. Editora Campus, 1983, 228p.