Nim-Lang

Compilação

Exemplo:

ola.nim

```
echo "Olá Mundo!"
```

Compilando:

```
# Para compilar
nim c ola.nim
# Para compilar e executar
nim c -r ola.nim # Olá Mundo!
```

> *saída:*
>
> ```
> $ ./ola
> Olá Mundo!
> ```

Comentários

Os comentários começam em qualquer lugar fora de uma string ou literal de caractere com o caractere hash #. Os comentários da documentação começam com ##:

```
# Um comentário.

var myVariable: int ## um comentário de documentação
```

Os comentários da documentação são tokens; eles são permitidos apenas em determinados locais no arquivo de entrada, pois

pertencem à árvore sintática! Esse recurso permite geradores de documentação mais simples.

Os comentários de várias linhas são iniciados com #[e terminados com]#. Os comentários multilinha também podem ser aninhados.

```
#[
você pode ter qualquer texto de código Nim comentado
fora dentro deste sem restrições de indentação.
    yes('Posso fazer uma pergunta sem sentido?')
  #[
     Nota: estes podem ser aninhados!!
  ]#
]#
```

Variáveis e Constantes

```nim
# ---
# Variáveis mutáveis.
# Tipagem estática:
var valor1: int = 10
# ---
# Declarando sem valor a priói:
var valor2: int
# ---
# Atribuindo valor:
valor2 = 20
# ---
# Inferindo o tipo:
var valor3 = 30
# ---
# Declarando em bloco.
# Obs.: No Nim, a indentação por tabulação
# não é permitido mas somente por espaço.
var
  valor4 = -10 # Tipo 'int'
  valor5 = "Olá" # Tipo 'string'
  valor6 = '!' # Tipo 'character'
# Variáveis acima são mutáveis mas seu tipo não,
# logo, a reatribuição: valor5 = 50 causará erro.
# ---
# Nim não faz distinção entre maiúsculas,
# minúsculas e sublinhados portanto as
# variáveis: 'contaRegistros' e
# 'conta_registros' são a mesma.
var contaRegistros: int = 5
echo conta_registros # 5
# ---
# Variáveis imutáveis.
# 'const' e 'let'.
# Na instrução 'const' o valor tem de ser
# conhecido em tempo de compilação.
const pi = 3.14
# Na instrução 'let' o valor não precisa ser conhecido
# em tempo de compilação, mas, uma vez atribuído
# um valor este não muda (nem o seu tipo).
var cem = 100
let porcento = 1 / cem
echo 4 * porcento # 0.04
```

Tipos de dados básicos

Inteiros:

```
# Underline (_) pode ser usado para separar milhares
var dezMilhoes = 10_000_000
echo dezMilhoes # 10000000
let
  a = 11
  b = 4
echo "a + b = ", a + b # a + b = 15
echo "a - b = ", a - b # a - b = 7
echo "a * b = ", a * b # a * b = 44
echo "a / b = ", a / b # a / b = 2.75
echo "a div b = ", a div b # a div b = 2
echo "a mod b = ", a mod b # a mod b = 3
```

Nim tem estes tipos inteiros integrados: *int*, *int8*, *int16*, *int32*, *int64*, *uint*, *uint8*, *uint16*, *uint32*, *uint64*.

O tipo inteiro padrão é int. Literais inteiros podem ter um sufixo de tipo para especificar um tipo inteiro não padrão:

```
let
  x = 0     # x é do tipo `int`
  y = 0'i8  # y is é do tipo `int8`
  z = 0'i32 # z é do tipo `int32`
  u = 0'u   # u é do tipo `uint`
```

Flutuantes:

```
# 2e3 = 2*10³
echo 2e3 # 2000.0
# Operadores 'div' e 'mod' não
# são definidos para flutuadores.
let
  c = 3.5
  d = 2.5
echo "c + d = ", c + d # c + d = 6.0
echo "c - d = ", c - d # c - d = 1.0
echo "c * d = ", c * d # c * d = 8.75
echo "c / d = ", c / d # c / d = 1.4
# multiplicação e divisão têm prioridade
# mais alta do que adição e subtração.
echo 2 + 3 * 4 # 14
echo 24 - 8 / 4 # 22.0
```

Nim tem estes tipos de ponto flutuante embutidos: *float*, *float32*, *float64*.

O tipo float padrão é float. Na implementação atual, float é sempre de 64 bits.

Literais flutuantes podem ter um sufixo de tipo para especificar um tipo flutuante não padrão:

```
var
  x = 0.0      # x é do tipo `float`
  y = 0.0'f32  # y é do tipo `float32`
  z = 0.0'f64  # z é do tipo `float64`
```

Convertendo floats e inteiros

```
let
  e = 5
  f = 2.6
# echo e + f # error
echo "float(e) = ", float(e) # 5.0
echo "int(f) = ", int(f) # 2 (não faz arredondamento)
echo "---"
echo "e + int(f) = ", e + int(f) # e + int(f) = 7
echo "float(e) + f = ", float(e) + f # float(e) + f = 7.6
```

Conversão de tipo

A conversão entre tipos numéricos é realizada usando o tipo como uma função:

```
var
  x: int32 = 1.int32   # o mesmo que chamar int32(1)
  y: int8  = int8('a') # 'a' == 97'i8
  z: float = 2.5       # int(2.5) arredonda para 2
  sum: int = int(x) + int(y) + int(z) # sum == 100
```

Tipos avançados

Em Nim novos tipos podem ser definidos dentro de uma declaração de tipo:

```
type
  biggestInt = int64      # maior tipo inteiro disponível
  biggestFloat = float64  # maior tipo float disponível
```

Characters

```
# Caracteres são escritos entre aspas simples
let
  h = 'z'
  i = '+'
  j = '2'
  newLine = '\n'
  tab = '\t'
  k = '35' # error
  l = 'xy' # error
```

Strings

```
let
  m = "palavra"
  n = "Esta é uma frase."
  o = "" # String vazia
  p = "32" # Não é um número
  q = "!" # Embora contenha um só caracter é uma string
```

Concatenação de string

```
var
  frase = "Ser ou não ser "
  continuacao = "eis a questão?"
  frase2 = "Vida longa "
  continuacao2 = "ao rei!"
# Integrando o conteúdo de
# 'continuacao' a 'frase'
frase.add(continuacao)
echo "Frase: ", frase # Frase: Ser ou não ser eis a questão?
# Imprimindo 'frase2' e sua
# 'continuacao2'
echo "Concat: ", frase2 & continuacao2 # Concat: Vida longa ao rei!
```

Operadores relacionais

```
let
  n1 = 10
  n2 = 20
echo "n1 > n2: ", n1 > n2 # false
echo "n1 < n2: ", n1 < n2 # true
echo "n1 igual n2: ", n1 == n2 # false
echo "n1 diferente n2: ", n1 != n2 # true
echo "n1 maior igual n2: ", n1 >= n2 # false
echo "n1 menor igual n2: ", n1 <= n2 # true
# Comparação letras e strings
let
  l1 = 'a'
  l2 = 'b'
  s1 = "Fulano"
  s2 = "Cicrano"
echo "l1 < l2: ", l1 < l2 # true
echo "s1 < s2: ", s1 < s2 # false
```

Operadores lógicos

```
echo "true and true: ", true and true # true
echo "true and false: ", true and false # false
echo "false and false: ", false and false # false
echo "---"
echo "true or true: ", true or true # true
echo "true or false: ", true or false # true
echo "false or false: ", false or false # false
echo "---"
echo "true xor true: ", true xor true # false
echo "true xor false: ", true xor false # true
echo "false xor false: ", false xor false # false
echo "---"
echo "not true: ", not true # false
echo "not false: ", not false # true
```

Controle de fluxo

if

```
let
  a = 10
  b = 20
  c = 30
if a < b: # true
  echo "a é menor que b" # a é menor que b
  if 10*a < b: # false
    echo "10*a é menor que b"
if b < c: # true
  echo "b é menor que c" # b é menor que c
  if 10*b < c: # false
    echo "10*b é menor que c"
if a+b == c: # true
  echo "Sim! a+b é igual a c" # Sim! a+b é igual a c
  if a+b <= b+c: # true
    echo "a+b é menor igual a b+c" # a+b é menor igual a b+c
# Nenhum recuo é necessário para instrução de atribuição única:
if x: x = false
```

else

```
let
  a = 15
  b = 5
if a < 10: # false
  echo "a é menor que 10"
else:
  echo "a é maior que 10" # a é maior que 10
if b < 10: # true
  echo "b é menor que 10" # b é maior que 10
else:
  echo "b é maior que 10"
```

elif

```
let
  a = 3000
  b = 7
if a < 10: # false
  echo "a é menor que 10"
elif a < 100: # false
  echo "a esta entre 10 e 100"
elif a < 1000: # false
  echo "a esta entre 100 e 1000"
else:
  echo "a é maior que 1000" # a é maior que 1000
if b < 1000: # true (entra neste bloco 'if' e ignora o resto)
  echo "b é menor que 1000" # b é menor que 1000
elif b < 100:
  echo "b é menor que 100"
elif b < 10:
  echo "b é menor que 10"
```

case

```
let x = 7
case x
of 5:
  echo "Cinco!"
of 7:
  echo "Sete!" # Sete!
of 10:
  echo "Dez!"
else:
  echo "Número desconhecido"
```

case - Escolha fechada (descartando alternativa de ação)

```
let h = 'y'
case h
of 'x':
  echo "Você escolheu x"
of 'y':
  echo "Você escolheu y" # Você escolheu y
of 'z':
  echo "Você escolheu z"
else: discard
```

case - multiplo

```
let i = 7
case i
of 0:
  echo "i é zero"
of 1, 3, 5, 7, 9:
  echo "i é ímpar" # i é ímpar
of 2, 4, 6, 8:
  echo "i é par"
else:
  echo "i é muito grande"
```

when

A instrução ***when*** é útil para escrever código específico da plataforma, semelhante à construção ***#ifdef*** na linguagem de programação ***C***.

```
when system.hostOS == "windows":
  echo "running on Windows!"
elif system.hostOS == "linux":
  echo "running on Linux!"
elif system.hostOS == "macosx":
  echo "running on Mac OS X!"
else:
  echo "unknown operating system"
```

Loops

for

```
for n in 5 .. 9: # [5, 9]
  echo n # Em cada linha: 5, 6, 7, 8, 9
echo "---"
for n in 5 ..< 9: # [5, 9[
  echo n # Em cada linha: 5, 6, 7, 8
echo "---"
for n in countup(0, 16, 4): # [0, 16] de 4 em 4
  echo n # Em cada linha: 0, 4, 8, 12, 16
echo "---"
for n in countdown(4, 0): # [4, 0]
  echo n # Em cada linha: 4, 3, 2, 1, 0
echo "---"
for n in countdown(-3, -9, 2): # [-3, -9] de 2 em 2
  echo n # Em cada linha: -3, -5, -7, -9
echo "---"
let palavra = "alfabeto"
for letra in palavra:
  echo letra # Em cada linha: a, l, f, a, b, e, t, o
echo "---"
# for incluindo contador (i)
for i, letra in palavra:
  echo "letra ", i, " é: ", letra # letra 0 é: a
                                  # letra 1 é: l
                                  # letra 2 é: f
                                  # letra 3 é: a
                                  # letra 4 é: b
                                  # letra 5 é: e
                                  # letra 6 é: t
                                  # letra 7 é: o
```

for - Esta construção é possivel:

```
# calcula fac(4) em tempo de compilação:
const fac4 = (var x = 1; for i in 1..4: x *= i; x)
```

while

```
var a = 1
while a*a < 10:
  echo "a é: ", a # a é: 1
                  # a é: 2
                  # a é: 3
  inc a # incremento de a, poderia ser:
        # a = a + 1 ou a += 1
echo "valor final de a: ", a # valor final de a: 4
```

break e continue

break

```
var i = 1
while i < 1000:
  if i == 3:
    break
  echo i # 1
         # 2
  inc i
```

continue

```
for i in 1 .. 5:
  if (i == 2) or (i == 4):
    continue
  echo i # 1
         # 3
         # 5
```

block

```
import std/strformat

block tabuadaDeDois:
  var
    x: int = 1
    y: int # Por padrão inicializa com 0
  while true:
    x += 1
    if x == 3: break tabuadaDeDois # Saída do bloco
    while true:
      y += 1
      if y == 11:
        y  =  0
        break # Saída do loop while mais interno
      echo(fmt"{x} x {y} = {x*y}")
```

Saída:

```
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
2 x 4 = 8
2 x 5 = 10
2 x 6 = 12
2 x 7 = 14
2 x 8 = 16
2 x 9 = 18
2 x 10 = 20
```

Containers

container

```
let frutas = ["goiaba", "banana", "maça", "pêra", "laranja", "morango",
"jambo"]
for i, fruta in frutas:
  echo "Fruta ", i+1, " é: ", fruta # Fruta 1 é: goiaba
                                    # Fruta 2 é: banana
                                    # Fruta 3 é: maça
                                    # Fruta 4 é: pêra
                                    # Fruta 5 é: laranja
                                    # Fruta 6 é: morango
                                    # Fruta 7 é: jambo
```

Array

```
var
  a: array[3, int] = [5, 7, 9] # Embora correto não é
                               # necessário declarar
                               # tamanho e tipo.
  b = [5, 7, 9] # Tamanho e tipo são inferidos.
  c = [] # error (não há como inferir tamanho e tipo).
  d: array[7, string] # Forma correta de declarar
                      # array vazio.
# ---
# Como o tamanho do array deve ser conhecido em
# tempo de compilação só podemos usar 'const'
const m = 3
let n = 5
var e: array[m, char]
var f: array[n, char] # error (pois 'n' é uma variável
                      # somente conhecida em tempo de
                      # execução).
```

Sequence

```
# Sequências são arrays dinâmicos sendo necessário apenas
# declarar o tipo dos elementos homogênios
var
  elem: seq[int] = @[] # Sequência vazia do tipo int.
  elem2: newSeq[int]() # Outra forma de declarar
                       # sequência vazia.
  f = @["abc", "def"] # Inferindo o tipo da sequência.
```

Adicionando elementos a uma sequência

```
# Lembrar que elementos de sequência
# devem ser do mesmo tipo.
var
  g = @['x', 'y']
  h = @['1', '2', '3']
g.add('z') # Adicionando a letra 'z' a sequência g
echo g # @['x', 'y', 'z']
h.add(g) # Adicionando a sequência g a sequência h
echo h # @['1', '2', '3', 'x', 'y', 'z']
# Obtendo tamanho da sequência
echo "---"
echo "Seq. h tem ", h.len, " elementos." # Seq. h tem 6 elementos.
```

Indexar e fatiar

```
let j = ['a', 'b', 'c', 'd', 'e']
echo j[0] # a (Primeiro elemento da esquerda para direita)
echo j[^1] # e (Último elemento, primeiro da dir. p/ esq.)
# Fatiando
echo j[0 .. 3] # @[a, b, c, d]
echo j[0 ..< 3] # @[a, b, c]
```

Atribuir e modificar valores em containers:

```
var
  k: array[5, int]
  l = @['p', 'w', 'r']
  m = "Tom and Jerry"
echo "---"
for i in 0 .. 4:
  k[i] = 7 * i
echo k # [0, 7, 14, 21, 28]
echo "---"
l[1] = 'q'
echo l # @['p', 'q', 'r']
echo "---"
m[8 .. 9] = "Ba"
echo m # Tom and Barry
       # ||||||||||||
       # 0123456789012
```

Tuplas

```
# Container de dados heterogêneos e tamanho fixo
# Obs.: Os dados são envolvidos por parênteses '()'
let n = ("banana", 2, 'c')
echo n # ("banana", 2, 'c')
```

Tuplas - Rotulando e modificando dados

```
var produto = (nome: "banana", precoPorKilo: 5.50, classificacao: 'fruta')
produto.nome = "abobora" # Modificando pelo rótulo
produto[1] = 4.30 # Modificando pelo indice
produto.classificacao = "legume"
echo produto # (nome: "abobora", precoPorKilo: 4.3, classificacao: "legume")
```

Procedures

Ex1:

```
proc retornaMaior(x: int, y: int): int =
  if x > y:
    return x
  else:
    return y

echo "---"
echo "Maior: ", retornaMaior(2, 5) # Maior: 5
echo "---"
echo "Maior: ", retornaMaior(10, 2) # Maior: 10
```

Ex2:

```
proc imprimeMelhorLinguagem(language: string) =
  case language
  of "Nim", "nim", "NIM":
    echo language, " é a melhor linguagem!"
  else:
    echo language, " pode ser uma segunda melhor linguagem."

echo "---"
imprimeMelhorLinguagem("nim") # nim é a melhor linguagem!
echo "---"
imprimeMelhorLinguagem("C#") # C# pode ser uma segunda melhor linguagem.
```

Ex3:

```
proc yes(question: string): bool =
  echo question, " (s/n)"
  while true:
    case readLine(stdin)
    of "s", "S", "sim", "Sim": return true
    of "n", "N", "não", "Não": return false
    else: echo "Por favor, seja claro: sim ou não"

if yes("Devo excluir todos os seus arquivos importantes?"):
  echo "Sinto muito, temo que não posso fazer isso."
else:
  echo "Acho que você sabe qual é o problema tão bem quanto eu."
```

> ***Saída:***
>
> ```
> Devo excluir todos os seus arquivos importantes? (s/n)
> s
> Sinto muito, temo que não posso fazer isso.
> ```

Para mudar valor de argumento:

```nim
# Informando a instrução 'var' antes do tipo.
proc acrescentaCinco(argumento: var int) =
  argumento += 5

# Para que isso funcione a variável que
# é passada como argumento também deve
# ser declarada como 'var'.
var valor = 10
acrescentaCinco(valor)
echo valor # 15
acrescentaCinco(valor)
echo valor # 20
# Aqui o argumento é passado antes
# do nome do procedimento ligado pelo
# conector '.' e podemos suprimir os
# parênteses '()'.
valor.acrescentaCinco
echo valor # 25
```

Variação do exemplo acima:

```
proc acrescentaValor(arg: var int, valorDeAcrescimo: int) =
  arg += valorDeAcrescimo

var valor = 10
# Aqui o 1º argumento é passado antes
# do nome do procedimento ligado pelo
# conector '.' e o 2º argumento dentro
# dos parênteses do procedimento.
valor.acrescentaValor(4)
echo "---"
echo valor # 14
valor.acrescentaValor(6)
echo "---"
echo valor # 20
```

Parâmetros com valores padrão

```
proc endereco(rua, num, bairro: string; cidade = "Belo Horizonte", estado =
"MG"): string =
  result.add("Rua: ")
  result.add(rua)
  result.add(", ")
  result.add(num)
  result.add(" - ")
  result.add(bairro)
  result.add(" - ")
  result.add(cidade)
  result.add("(")
  result.add(estado)
  result.add(")")

echo endereco("Capivara", "101", "Pampulha")
```

> ***Saída:***
>
> ```
> Rua: Capivara, 101 - Pampulha - Belo Horizonte(MG)
> ```

Também é possivel usar variáveis e/ou constantes declarados fora do procedimento:

```nim
var x = 100
const unidade = 1

proc echoX() =
  echo x
  x += unidade
  echo x

echoX() # 100
        # 101
```

Retorno sem *'return'*:

Ex1:

```nim
proc olaMundo(): string =
  "Olá, Mundo!"
```

Ex2:

```nim
proc encontrarMaior(a: seq[int]): int =
  for number in a:
    if number > result: # 'result' é incializado por
                        # padrão com 0 do tipo 'int'.
      result = number

let d = @[3, -5, 11, 33, 7, -15]
echo encontrarMaior(d) # 33
```

Ex3:

```nim
proc encontrarImpares(a: seq[int]): seq[int] =
  # result é inicializado por padrão
  # com um sequência vazia '@[]'.
  for number in a:
    if number mod 2 == 1:
      result.add(number)

let f = @[1, 6, 4, 43, 57, 34, 98]
echo encontrarImpares(f) # @[1, 43, 57]
```

return - sem parâmetro:

```nim
proc somaAteAcharNegativo(x: varargs[int]): int =
  for i in x:
    if i < 0:
      return
    result = result + i

echo somaAteAcharNegativo() # ecôa 0
echo somaAteAcharNegativo(3, 4, 5) # ecôa 12
echo somaAteAcharNegativo(3, 4 , -1 , 6) # ecôa 7
```

Chamando procedimento dentro de procedimento

```
proc ehDivisivelPor3(x: int): bool =
  x mod 3 == 0 # O mesmo que: return x mod 3 == 0

proc filtraMultiplosDe3(a: seq[int]): seq[int] =
  for i in a:
    if i.ehDivisivelPor3():
      result.add(i)

let
  g = @[2, 6, 5, 7, 9, 0, 5, 3]
  h = @[5, 4, 3, 2, 1]
  i = @[626, 45390, 3219, 4210, 4126]

# As formas de enviar parâmetros na 'procedure'
echo filtraMultiplosDe3(g) # @[6, 9, 0, 3]
echo h.filtraMultiplosDe3 # @[3]
echo filtraMultiplosDe3 i # @[45390, 3219]
```

Assinatura de 'procedure' e a utilização destas 'procedures' antes da sua implementação

Ex1:

```
# Assinatura da 'procedure'.
proc plus(x, y: int): int

# Usando a 'procedure' antes de sua implementação.
echo 5.plus(10) # 15

# Implementando a 'procedure'.
proc plus(x, y: int): int =
  x + y
```

Ex2:

```nim
# 'impar()' depende de 'par()' e vice-versa
# por isso iniciamos com a assinatura de 'par()'
# para que não ocorra problema em 'impar()'.
proc par(n: int): bool

proc impar(n: int): bool =
  assert(n >= 0) # garante que não encontremos recursão negativa
  if n == 0: false
  else:
    n == 1 or par(n-1)

proc par(n: int): bool =
  assert(n >= 0) # garante que não encontremos recursão negativa
  if n == 1: true
  else:
    n == 0 or impar(n-1)

echo "4 é par: ", par(4) # ecôa: 4 é par: true
echo "7 é ímpar: ", impar(7) # ecôa: 7 é ímpar: true
```

Procedimentos sobrecarregados

```nim
proc toString(x: int): string =
  result =
    if x < 0: "negativo"
    elif x > 0: "positivo"
    else: "zero"

proc toString(x: bool): string =
  result =
    if x: "sim"
    else: "não"

# chama o proc toString(x: int)
echo toString(13) # ecôa positivo
# chama o proc toString(x: bool)
echo toString(true) # ecôa sim
```

Operadores são 'procedures':

```
# if 3 + 4 == 7: echo "true" # ecôa true
if `==`( `+`(3, 4), 7): echo "true" # ecôa true
```

Interadores (*Iterators*) - Uma palavra

Exemplo de um contador:

```
echo "Contando até dez:"
for i in countup(1, 10):
  echo i
```

Se quisermos fazer nosso próprio ***countup()*** com uma ***'procedure'*** para usar no código acima, temos esta primeira abordagem:

```
proc colecao(a, b: int): int =
  var res = a
  while res <= b:
    return res
    inc(res)

echo "Contando até dez:"
for i in colecao(1, 10):
  echo i
```

Para o objetivo pretendido a ***'procedure'*** acima não funcionaria, deve ser feito desta forma:

```
iterator colecao(a, b: int): int =
  var res = a
  while res <= b:
    yield res
    inc(res)

echo "Contando até dez:"
for i in colecao(1, 10):
  echo i
```

> ***Obs.:*** Os iteradores são muito semelhantes aos procedimentos, mas existem várias diferenças importantes:
>
> - Iteradores só podem ser chamados de ***loops for***.
> - Iteradores não podem conter uma instrução ***return*** (e ***procs*** não podem conter uma instrução ***yield***).
> - Os iteradores não têm variável de resultado implícita.
> - Iteradores não suportam recursão.
> - Iteradores não podem ser declarados para frente, porque o compilador deve ser capaz de interar inline. (Esta restrição desaparecerá em uma versão futura do compilador.)

Módulos

Módulos Nim mais usados

- ***strutils*** - funcionalidade adicional ao lidar com strings
- ***sequtils*** - funcionalidade adicional para sequências
- ***math*** - funções matemáticas (logaritmos, raíz quadrada, ...), trigonometria (sen, cos, ...)
- ***times*** - medir e lidar com o tempo

Importando um módulo

Ex1:

```
import strutils
let
  a = "Minha string com espaço em branco."
  b = '!'
echo a.split() # @["Minha", "string", "com", "espaço", "em", "branco."]
echo a.toUpperAscii() # MINHA STRING COM ESPAçO EM BRANCO.
echo b.repeat(5) # !!!!!
```

Ex2:

```
import math
let
  c = 30.0 # graus
  cRadians = c.degToRad() # Convertendo graus em radianos.
echo cRadians # 0.5235987755982988
echo sin(cRadians).round(2) # 0.5
                            # Mostra o seno do valor em
                            # radianos arredondando p/
                            # duas casas decimais.
# O módulo 'math' contém operador de potências (^).
echo 2^5 # 32
```

Criando nosso próprio módulo

primeiroArquivo.nim

```
proc plus*(a, b: int): int = # O asterisco (*) torna o
                             # procedimento disponível
                             # na importação por outro
                             # arquivo.
  return a + b

proc minus(a, b: int): int = # Este procedimento não esta
                             # disponível na importação
                             # por outro arquivo por não
                             # conter o asterisco após seu
                             # nome.
  return a - b
```

segundoArquivo.nim

```
import primeiroArquivo
echo plus(5, 10) # 15
# echo minus(10, 5) # error
```

Importação de vários arquivos inclusive em subdiretórios

```
.
├── Subdir
│   └── terceiroArquivo.nim
├── outroSubdir
│   ├── quartoArquivo.nim
│   └── quintoArquivo.nim
├── primeiroArquivo.nim
└── segundoArquivo.nim
```

Agora importando os arquivos acima

segundoArquivo.nim

```
import primeiroArquivo
import Subdir/terceiroArquivo
import outroSubdir/[quartoArquivo, quintoArquivo]
```

Interagindo com a entrada do usuário

Lendo de um arquivo

pessoas.txt:

```
Fulano
Cicrano
Beltrano
```

lendoDoArquivo.nim:

```
import strutils
let contents = readFile("pessoas.txt")
echo contents # Fulano
              # Cicrano
              # Beltrano
              # (linha em branco)
let people = contents.splitLines()
echo people # @["Fulano", "Cicrano", "Beltrano", ""]
```

Obs.: Nas duas saídas do código acima notamos que na primeira ocorre uma linha em branco e na segunda ocorre uma string nula na sequência.

Refatorando o código acima:

lendoDoArquivo.nim:

```
import strutils
# Acrescentando a instrução 'strip()' eliminamos
# de cada linha: espaço, novas linhas, espaços,
# tabulações, etc.
let contents = readFile("pessoas.txt").strip()
echo contents # Fulano
              # Cicrano
              # Beltrano
let people = contents.splitLines()
echo people # @["Fulano", "Cicrano", "Beltrano"]
```

Lendo a entrada do usuário

```
echo "Qual seu nome?" # Qual seu nome?
let name = readLine(stdin) # Fulano
echo "Olá ", name, "!" # Olá Fulano!
```

Lidando com números

```
# Faz-se necessário a importação
# do módulo 'strutils' para o uso
# das 'procedures': 'parseInt()',
# 'parseFloat', etc.
import strutils
echo "Entre com o ano de nascimento:"
let anoDeNasc = readLine(stdin).parseInt() # Ex.: 1972
let idade = 2022 - anoDeNasc
echo "Você tem ", idade, " anos." # Você tem 50 anos.
```

Lendo números de um arquivo

Obter a soma e média desses números

numbers.txt

```
27.3
98.24
11.93
33.67
55.01
```

obterSomaMedia.nim

```
import strutils, sequtils, math

let
  strNums = readFile("numbers.txt").strip().splitLines()
  nums = strNums.map(parseFloat)
let
  somaNums = sum(nums)
  media = somaNums / float(nums.len)

echo "A soma da seq.: ", nums, " é:" # A soma da seq.: @[27.3,
98.23999999999999, 11.93, 33.67, 55.01] é:
echo somaNums # 226.15
echo "E a média é:"
echo media # 45.23
```

Nim - Programação Orientada a Objetos

Herança

```
type
  # Para habilitar a herança o objeto precisa herdar de RootObj.
  Pessoa = ref object of RootObj
    nome*: string  # o * significa que `nome` é acessível a partir de outros
módulos
    idade: int     # sem * significa que o campo está oculto de outros
módulos

  Estudante = ref object of Pessoa # Estudante herda de Pessoa
    id: int                        # com um campo id

var
  estudante: Estudante

# construção do objeto:
estudante = Estudante(nome: "Antônio", idade: 5, id: 2)
echo estudante[] # ecôa (id: 2, nome: "Antônio", idade: 5)
```