Функции

Создание функций

В языке **Julia** функция представляет собой объект, который сопоставляет кортеж значений аргументов с возвращаемым значением. Базовый синтаксис определения функций в **Julia** следующий:

[ ]:
```
function f(x,y)
    x + y
end
```

Эта функция принимает два аргумента, x и y, и возвращает значение последнего вычисленного выражения: `x + y`.

Есть и второй, более сжатый синтаксис для определения функции в **Julia**. Традиционный синтаксис объявления функций, показанный выше, эквивалентен следующей компактной форме присваивания:

[ ]:
```
f(x,y) = x + y
```

В форме присваивания тело функции должно быть одним выражением. Короткий синтаксис функций довольно наглядный, он значительно сокращает объем набираемого текста и устраняет визуальный шум.

Вызывается функция всегда с помощью употребления ее имени вместе со списком значений параметров, взятым в круглые скобки:

[ ]:
```
f(2,3)
```

✏Задание 1

Создайте функцию `f(x)`, вычисляющую значение многочлена $2x^2 + 3x + 5$. Затем вызовите функцию `f` со значениями аргумента, равными 4 и 7.

[ ]:
```

```

Решение

[ ]:
```
function f(x)
    2*x^2+3*x+5
end
```

[ ]:
```
f(4)
```

[ ]:
```
f(7)
```

Без скобок выражение f ссылается на функцию как объект и может передаваться, как и любое другое значение:

[ ]:
```
new_func = f
new_func(2,3)
```

Как и в случае с другими именами в **Julia**, в качестве имен функций могут быть использованы любые символы Unicode.

✏Задание 2

Создайте функцию `func(x,y,z)`, вычисляющую синус суммы аргументов этой функции. Затем вызовите функцию `func` со значениями аргументов, равными $\pi/4$, $\pi/3$ и $\pi/5$.

[ ]:
```

```

Решение

[ ]:
```
function func(x,y,z)
    sin(x+y+z)
end
```

[ ]:
```
func(pi/4, pi/3, pi/5)
```

Объявления типов аргументов

Вы можете объявлять типы аргументов функций, добавляя `::TypeName` к имени аргумента. Например, следующая функция рекурсивно вычисляет числа Фибоначчи:

[ ]:
```
fib(n::Integer) = n ≤ 2 ? one(n) : fib(n-1) + fib(n-2)
```

Здесь спецификация `::Integer` означает, что функция `fib` будет вызываемой, только если ее аргумент `n` является подтипом абстрактного типа `Integer`.

Следующие вызовы функции `fib` возвращают значения элементов последовательности Фибоначчи с номерами 3 и 10:

[ ]:
```
fib(3)
```

[ ]:
```
fib(10)
```

Если мы попытаемся в качестве аргумента задать нецелое число, то получим сообщение об ошибке:

```
fib(2.5)
```

```
MethodError: no method matching fib(::Float64)
```

```
Closest candidates are:
  fib(::Integer)
   @ Main In[8]:1

Stacktrace:
 [1] top-level scope
   @ In[13]:1
```

Объявления типов могут быть полезными, если ваша функция возвращает правильные результаты только для определенных типов аргументов. Например, если бы мы опустили типы аргументов и написали

`fib(n) = n ≤ 2 ? one(n) : fib(n-1) + fib(n-2),`

то вызов `fib(2.5)` дал бы нам бессмысленный ответ `2.0`.

Ключевое слово `return`

Значение, возвращаемое функцией, – это значение последнего вычисленного выражения, которое по умолчанию является последним выражением в теле определения функции. В примере функции `f` из раздела **Создание функций** это значение выражения `x + y`. В качестве альтернативы ключевое слово `return` заставляет функцию немедленно выполнить возврат, предоставляя выражение, значение которого возвращается.

Например:

[ ]:
```
function g(x,y)
    return x * y
    x + y
end
```

Вызовем функцию `g` с аргументами 2 и 3:

[ ]:
```
g(2,3)
```

Как мы видим, функция `g` выполнила умножение аргументов 2 и 3, т.е. то действие, которое записано сразу после ключевого слова `return`, а следующее действие (сложение аргументов `x` и `y`) было проигнорировано.

Конечно, в истинно линейном теле функции, например `g`, использование `return` бессмысленно. Так как выражение `x + y` никогда не вычисляется, мы могли бы просто сделать `x * y` последним выражением в функции и опустить `return`. Однако в случае другого потока управления `return` действительно может быть полезен.

Вот, например, функция `quadratic`, которая вычисляет корни квадратного уравнения $ax^2 + bx + c = 0$. Аргументами функции являются коэффициенты `a`, `b` и `c`.

[ ]:
```
function quadratic(a,b,c)
    D = b^2 - 4*a*c
    if D < 0
        return "действительных корней нет"
    elseif D == 0
        return -b/2a
    else
        return ((-b+sqrt(D))/2a, (-b-sqrt(D))/2a)
    end
end
```

Есть три возможные точки возврата из этой функции, возвращающие значения трех разных выражений, в зависимости от значения дискриминанта D. Ключевое слово return в последней строке можно было бы опустить, так как это последнее выражение.

✏Задание 3

Проверьте работу функции quadratic, решив с ее помощью три квадратных уравнения * $x^2 + 5x + 7 = 0$, * $x^2 - 2x + 1 = 0$, * $x^2 + 6x - 16 = 0$.

[ ]:
```

```

Решение

[ ]:
```
quadratic(1,5,7)
```

[ ]:
```
quadratic(1,-2,1)
```

[ ]:
```
quadratic(1,6,-16)
```

Анонимные функции

Функции в **Julia** являются объектами первого класса: их можно присваивать переменным и вызывать с помощью стандартного синтаксиса вызова функции из переменной, которой они были присвоены. Они могут использоваться как аргументы и возвращаться как значения. Их также можно создавать анонимно, без присваивания им имени, с помощью любого из этих синтаксисов:

[ ]:
```
x -> x^2 + 2x - 1
```

[ ]:
```
function (x)
    x^2 + 2x - 1
end
```

Этот код создает функцию, принимающую один аргумент x и возвращающую значение многочлена x^2 + 2x - 1 от этой величины. Обратите внимание, что результат представляет собой универсальную функцию, но с именем, сгенерированным компилятором на основе последовательной нумерации.

Основное использование анонимных функций – их передача в функции, которые принимают другие функции в качестве аргументов. Классический пример – функция `map`, которая применяет функцию к каждому значению массива и возвращает новый массив, содержащий результирующие значения:

[ ]:
```
map(sin, [1.2, 3.5, 1.7])
```

Это отлично работает, если уже существует осуществляющая преобразование именованная функция для передачи в качестве первого аргумента в `map`. Однако часто готовой к использованию именованной функции не существует. В этих ситуациях конструкция анонимной функции позволяет легко создавать объект функции для однократного использования без указания имени:

[ ]:
```
map(x -> x^2 + 2x - 1, [1, 3, -1])
```

Анонимную функцию, принимающую несколько аргументов, можно записать с помощью следующего синтаксиса:

```
(x,y,z) -> 2x + y - z
```

Кортежи

В **Julia** имеется встроенная структура данных, называемая **кортежем**, которая тесно связана с аргументами функций и возвращаемыми значениями. Кортеж – это контейнер фиксированной длины, который может содержать любые значения, но не может быть изменен. Кортежи конструируются так: элементы кортежа перечисляются через запятую, и весь этот перечень берется в круглые скобки.

[ ]:
```
(1, 2, 3)
```

[ ]:
```
x = (0.0, "hello", 6*7)
```

К элементам кортежа можно обращаться с помощью индексирования:

[ ]:
```
x[2]
```

Кортеж из одного элемента должен записываться с запятой: `(1,)`, а `()` представляет собой пустой кортеж (с длиной 0).

Элементам кортежа можно задавать имена, в этом случае конструируется **именованный кортеж**:

[ ]:
```
x = (a=1, b=2)
```

[ ]:
```
x[1]
```

К элементам именованного кортежа можно также обращаться по имени с помощью синтаксиса через точку (`x.a`) помимо обычного синтаксиса индексирования (`x[1]`).

[ ]:
```
x.a
```

Разделенный запятыми список переменных (которые, как вариант, могут заключаться в скобки) может отображаться в левой части присваивания: значение с правой стороны **деструктуризируется** путем итерации по переменным и присваивания значения каждой из них по очереди:

[ ]:
```
a, b, c = (1, 2, 3)
```

[ ]:
```
b
```

Значение справа может быть кортежем, массивом или итератором (например, `1:3`) как минимум той же длины, что и набор переменных слева (любые излишние элементы в правой части игнорируются).

Кортежи используются для возврата нескольких значений из функции. Например, следующая функция возвращает два значения:

[ ]:
```
function func(a,b)
    a+b, a*b
end
```

Если вы вызовете ее без присваивания возвращаемого значения какой-либо переменной, то вы увидите, что возвращен кортеж:

[ ]:
```
func(2,3)
```

Деструктуризирующее присваивание извлекает каждое значение в переменную:

[ ]:
```
(x, y) = func(2,3)
```

Список переменных x и y можно записать и без скобок:

[ ]:
```
x, y = func(2,3)
println(x)
println(y)
```

✏Задание 4

Создайте функцию `pow(x,y)`, возвращающую два значения: x^y и y^x. Затем вызовите эту функцию со значениями аргументов, равными 5 и 7. Присвойте возвращаемые значения переменным a и b. Выведите на экран значения переменных a и b.

[ ]:
```

```

Решение

[ ]:
```
function pow(x,y)
    x^y, y^x
end
```

[ ]:
```
(a, b) = pow(5, 7)
println(a)
println(b)
```

Именованные аргументы

Некоторым функциям требуется большое число аргументов, или у них большое число вариантов поведения. Бывает трудно запомнить, как вызывать такие функции. **Именованные аргументы** могут сделать эти сложные интерфейсы более простыми в использовании и расширить их, разрешив определение аргументов по имени, а не только по положению.

Например, рассмотрим функцию `plot`, которая строит линейный график. У этой функции может быть множество параметров, управляющих стилем линии, толщиной, цветом и т.д. Если она принимает именованные аргументы, возможный вызов может выглядеть как `plot(x, y, width=2)`, в котором мы решили указать только толщину линии. Заметьте, что это служит двум целям. Вызов легче читается, так как мы можем пометить аргумент его значением. Также становится возможным передавать любое подмножество большого числа аргументов в любом порядке.

Функции с именованными аргументами определяются с помощью точки с запятой в списке аргументов:

```
function plot(x, y; style="solid", width=1, color="black")
    ###
end
```

При вызове функции точка с запятой необязательна: можно вызвать `plot(x, y, width=2)` или `plot(x, y; width=2)`, но первый стиль более распространен.

Значения по умолчанию именованных аргументов вычисляются только при необходимости (когда соответствующий именованный аргумент не передан) и в порядке слева направо. Следовательно, выражения по умолчанию могут ссылаться на предыдущие именованные аргументы.

Типы именованных аргументов можно сделать явными следующим образом:

```
function f(;x::Int=1)
    ###
end
```

✏Задание 5

Создайте функцию `cost`, вычисляющую стоимость слитка золота, имеющего форму прямоугольного параллелепипеда. На вход функции должны подаваться 4 именованных аргумента: `length` (длина, см), `width` (ширина, см), `height`

(высота, см) и price (цена, руб. за грамм). Плотность золота примите равной $19,32/^3$. Затем, вызвав эту функцию, вычислите стоимость слитка золота размерами $10 \times 3 \times 1$ см и ценой 6200 руб. за грамм.

[ ]:

Подсказка Т.к. все аргументы функции – именованные, то заголовок описания функции будет выглядеть так (список именованных аргументов располагается после точки с запятой):

```
function cost(; length, width, height, price)
```

Напомним, что объем слитка (^{3}) рассчитывается как произведение длины, ширины и высоты, а масса (г) – как произведение объема (^{3}) на плотность ($/^3$). Стоимость равна произведению массы (г) на цену (руб./г). Полученное значение стоимости нужно округлить до большего целого числа (с помощью функции ceil) и затем преобразовать в целый тип (с помощью функции Int) для предотвращения вывода результата в экспоненциальной форме, которая не используется для денежных сумм.

Решение

[ ]:
```
function cost(; length, width, height, price)
    V = length * width * height
    m = V * 19.32
    return cost = Int(ceil(m * price))
end
```

[ ]:
```
cost(length=10, width=3, height=1, price=6200)
```

Если именованному аргументу не присвоено значение по умолчанию в определении метода, тогда он является обязательным:

[ ]:
```
function f(x; y)
    ###
end
```

[ ]:
```
f(3, y=5)
```

Если вызывающий объект не присваивает значение именованному арументу, то будет выдано исключение UndefKeywordError:

```
f(3)

UndefKeywordError: keyword argument `y` not assigned

Stacktrace:
 [1] f(x::Int64)
```

```
  @ Main ./In[7]:1
 [2] top-level scope
  @ In[11]:1
```

Можно также передавать выражения `key => value` после точки с запятой. Например:

```
plot(x, y; :width => 2)
```

эквивалентно

```
plot(x, y, width=2)
```

Математические функции

В **Julia** существует большое количество стандартных математических функций. Рассмотрим наиболее часто используемые из них.

Абсолютные значения и корни

- `abs(x)` – абсолютное значение числа x.
- `abs2(x)` – квадрат числа x.
- `sqrt(x)` – квадратный корень из числа x.
- `isqrt(x)` – целочисленный квадратный корень из целого числа x.
- `cbrt(x)` – кубический корень из числа x.

Примеры:

[ ]:
```
abs(-10)
```

[ ]:
```
abs2(-10)
```

[ ]:
```
sqrt(1000)
```

[ ]:
```
cbrt(729)
```

Степени и логарифмы

- `exp(x)` – e в степени x.
- `exp2(x)` – 2 в степени x.
- `exp10(x)` – 10 в степени x.
- `expm1(x)` – e в степени x $-$ 1 (точно).
- `ldexp(x,n)` – $x \times 2^n$ (n должно быть типа `Float`).
- `exponent(x)` – наибольшее целое число y, такое, что $2^y \leq |x|$.
- `log(x)` – натуральный логарифм x.
- `log2(x)` – логарифм x по основанию 2.
- `log10(x)` – десятичный логарифм x.
- `log(n,x)` – логарифм x по основанию n.
- `log1p(x)` – натуральный логарифм 1 + x (точно).

- `significand(x)` – двоичная значащая часть (мантисса) числа с плавающей запятой x.

Примеры:

[ ]:
```
exp(3)
```

[ ]:
```
exp2(10)
```

[ ]:
```
exp10(6)
```

[ ]:
```
exponent(17)
```

[ ]:
```
log(234)
```

[ ]:
```
log2(1024)
```

[ ]:
```
log(3,81)
```

✏Задание 6

С помощью встроенных математических функций вычислите значения следующих выражений: * $\sqrt{7396}$, * e^5, * $\ln 10$, * $\log_2 100$, * $\log_{10} 365$.

[ ]:
```

```

Решение

[ ]:
```
sqrt(7396)
```

[ ]:
```
exp(5)
```

[ ]:
```
log(10)
```

[ ]:
```
log2(100)
```

[ ]:
```
log10(365)
```

Тригонометрические функции

Если аргумент x задан в радианах, то

- `sin(x)` – синус x .
- `sinpi(x)` – синус πx.
- `cos(x)` – косинус x.
- `cospi(x)` – косинус πx.
- `tan(x)` – тангенс x.
- `cot(x)` – котангенс x.
- `sec(x)` – секанс x.

- `csc(x)` – косеканс x.
- `asin(x)` – арксинус x.
- `acos(x)` – арккосинус x.
- `atan(x)` – арктангенс x.
- `acot(x)` – арккотангенс x.
- `asec(x)` – арксеканс x.
- `acsc(x)` – арккосеканс x.
- `sinc(x)` – кардинальный синус x: $\text{sinc}(x) = \begin{cases} \frac{\sin(\pi x)}{\pi x} x \neq 0 \\ 1 x = 0 \end{cases}$
- `cosc(x)` – производная от кардинального синуса x: $\text{cosc}(x) = \begin{cases} \frac{\cos(\pi x)}{x} - \frac{\sin(\pi x)}{\pi x^2} x \neq 0 \\ 0 x = 0 \end{cases}$

Для вычисления тригонометрических функций с использованием градусов (а не радиан) добавьте к имени функции окончание d: `sind(x)`, `cosd(x)`, `tand(x)`, `cotd(x)`, `secd(x)`, `cscd(x)`, `asind(x)`, `acosd(x)`, `atand(x)`, `acotd(x)`, `asecd(x)`, `acscd(x)`.

- `rad2deg(x)` – преобразовать угол x из радиан в градусы.
- `deg2rad(x)` – преобразовать угол x из градусов в радианы.

Примеры:

[ ]:
```
sin(2.31)
```

[ ]:
```
cos(pi)
```

[ ]:
```
tan(pi/4)
```

[ ]:
```
atan(1)
```

[ ]:
```
sinc(0.5)
```

[ ]:
```
rad2deg(atan(1))
```

[ ]:
```
deg2rad(180)
```

✏Задание 7

С помощью встроенных математических функций вычислите значения следующих выражений: * $\sin \pi/12$, * $\cos 30\pi$, * $\text{tg} 89^\circ$, * $\arcsin 0.5$.

[ ]:
```

```

Подсказка Для того чтобы вычислить тангенс аргумента, заданного в градусах, воспользуйтесь функцией `tand`.

Решение

[ ]:
```
sin(pi/12)
```

[ ]:
```
cos(30*pi)
```

[ ]:
```
tand(89)
```

[ ]:
```
asin(0.5)
```

Гиперболические функции

- `sinh(x)` – гиперболический синус x.
- `cosh(x)` – гиперболический косинус x.
- `tanh(x)` – гиперболический тангенс x.
- `coth(x)` – гиперболический котангенс x.
- `sech(x)` – гиперболический секанс x.
- `csch(x)` – гиперболический косеканс x.

Соответствующие им обратные гиперболические функции: `asinh(x)`, `asoch(x)`, `atanh(x)`, `acoth(x)`, `asech(x)`, `acsch(x)`.

Примеры:

[ ]:
```
sinh(2)
```

[ ]:
```
cosh(0)
```

[ ]:
```
tanh(1)
```

Комбинаторные функции

- `factorial(x)` – факториал числа x.
- `binomial(x,y)` – число сочетаний из a по b (биномиальный коэффициент).

Примеры:

[ ]:
```
factorial(10)
```

[ ]:
```
binomial(6,4)
```

Функции округления

- `round(x)` – округляет x до ближайшего целого числа.
- `ceil(x)` – округляет x до ближайшего большего целого числа.
- `floor(x)` – округляет x до ближайшего меньшего целого числа.
- `trunc(x)` – отбрасывает дробную часть числа x.

```
[ ]: round(15.64)
```

```
[ ]: ceil(15.64)
```

```
[ ]: floor(15.64)
```

```
[ ]: trunc(15.64)
```

✏Задание 8

С помощью встроенных математических функций выполните округление числа 3,49 всеми четырьмя рассмотренными выше способами.

```
[ ]:
```

Решение

```
[ ]: round(3.49)
```

```
[ ]: ceil(3.49)
```

```
[ ]: floor(3.49)
```

```
[ ]: trunc(3.49)
```

Функции деления

- `div(x,y)` – целочисленное деление с усечением. Частное округляется “к нулю”.
- `fld(x,y)` – целочисленное деление с округлением в меньшую сторону. Частное округляется в сторону минус бесконечности.
- `cld(x,y)` – целочисленное деление с округлением в большую сторону. Частное округляется в сторону плюс бесконечности.
- `rem(x,y)` – остаток от целочисленного деления x на y; удовлетворяет условию `x == div(x,y)*y + rem(x,y)`. Знак соответствует x.
- `mod(x,y)` – остаток от целочисленного деления x на y; удовлетворяет условию `x == fld(x,y)*y + mod(x,y)`. Знак соответствует y.
- `mod2pi(x)` – остаток от целочисленного деления x на 2π.

Примеры:

```
[ ]: div(20,3)
```

```
[ ]: fld(-20,3)
```

```
[ ]: cld(20,3)
```

```
[ ]: rem(20,3)
```

[ ]:
```
mod(-20,3)
```

Другие полезные функции

- `gcd(x,y)` – наибольший положительный общий делитель чисел x, y.
- `lcm(x,y)` – наименьшее общее кратное чисел x, y.
- `min(x,y)` – минимальное значение из списка (для произвольного количества чисел в списке).
- `max(x,y)` – максимальное значение из списка (для произвольного количества чисел в списке).
- `minmax(x,y)` – минимальное и максимальное значения из двух чисел x, y; результат в форме кортежа.
- `muladd(x,y,z)` – вычисляет значение x*y+z.
- `hypot(x,y)` – гипотенуза прямоугольного треугольника с катетами x и y.
- `eval(x)` – вычислить значение выражения x.
- `real(x)` – вещественная часть числа x.
- `imag(x)` – мнимая часть числа x.
- `reim(x)` – возвращает вещественную и мнимую части x (в виде кортежа).
- `conj(x)` – число, комплексно-сопряженное к числу x.
- `sign(x)` – знак числа x, возвращает -1, 0 или +1.
- `modf(x)` – кортеж, содержащий дробную и целую части числа x. Обе части имеют тот же знак, что у аргумента.
- `digits(x)` – массив десятичных цифр, образующих целое число.
- `isapprox()` – позволяет сравнивать два числа с заданным уровнем погрешности `atol`:

```
isapprox(1.0, 1.05; atol = 0.1) (true)
```

```
isapprox(1.0, 1.1; atol = 0.05) (false)
```

Примеры:

[ ]:
```
gcd(24,32)
```

[ ]:
```
lcm(24,32)
```

[ ]:
```
min(24, 33, 51, 22)
```

[ ]:
```
hypot(6,8)
```

[ ]:
```
reim(2+3im)
```

[ ]:
```
conj(2+3im)
```

[ ]:
```
sign(-5)
```

[ ]:
```
modf(432.657)
```

```
[ ]: isapprox(12.47, 12.58, atol=0.01)
```

✏Задание 9

С помощью встроенных математических функций вычислите: * результат целочисленного деления 100 на 6, * остаток от целочисленного деления 100 на 6, * наибольший общий делитель чисел 64 и 48, * наименьшее общее кратное чисел 64 и 48, * вещественную и мнимую части комплексного числа 4-6i.

```
[ ]:
```

Решение

```
[ ]: div(100,6)
```

```
[ ]: rem(100,6)
```

```
[ ]: gcd(64,48)
```

```
[ ]: lcm(64,48)
```

```
[ ]: reim(4-6im)
```

Специальные функции

Библиотека SpecialFunctions.jl содержит множество дополнительных специальных математических функций. Среди них гамма-функция, бета-функция, интегральный синус, интегральный косинус, функция ошибок, функции Эйри, функции Бесселя, эллиптические интегралы, дзета-функция Римана и др.

Область видимости переменных

Под **блоком кода** будем понимать фрагмент кода, заключенный в синтаксическую конструкцию `function` (функция) или `let … end`. Внутри таких блоков кода можно объявлять локальные переменные, которые будут видны только внутри этого блока и не будут конфликтовать с переменными в других частях кода (например с глобальными переменными).

Область видимости определяет, где в коде вы можете использовать переменную. Так, переменные бывают локальные и глобальные.

Локальная (внутренняя) переменная – переменная, объявленная и определенная внутри своего родительского элемента, например функции, объекта, блока кода и т.д. Локальные переменные видны только внутри той области видимости, в которой они определены, и недоступны за ее пределами. Рассмотрим локальную переменную на примере:

[ ]:
```
function func()
    local_var = 10
    println(local_var)
end
```

Здесь `function func()` определяет функцию с именем `func`, содержащую локальную переменную `local_var` в локальной области видимости.

Вызов функции командой `func()` выведет значение переменной `local_var`, равное 10:

[ ]:
```
func()
```

После выполнения функции локальная переменная `loval_var` выходит из области видимости и становится недоступной за пределами функции. Это означает, что переменная существует только в пределах блока, в котором она была определена.

Попробуем вызвать переменную `local_var` вне локальной области с помощью следующего кода:

```
println(local_var)
```

```
UndefVarError: `local_var` not defined
```

Мы видим, что внешний код не имеет доступа к локальной переменной и выдает сообщение об ошибке о том, что локальная переменная `local_var` не определена.

Глобальная (внешняя) переменная – переменная, определенная вне функций и блоков кода с локальной областью видимости. Глобальная переменная доступна из любой части кода и находится в глобальной области видимости. Рассмотрим глобальную переменную на примере:

[ ]:
```
global_var = 20
function func()
    println(global_var)
end
```

Этот код задает глобальную переменную `global_var` и определяет функцию `func()`, которая выводит значение глобальной переменной. Вызов функции командой `func()` выведет значение глобальной переменной `global_var`, равное 20:

[ ]:
```
func()
```

Глобальную переменную `global_var` можно использовать за пределами функции:

[ ]:
```
println(global_var)
```

Затенение переменных – это ситуация, когда внутри определенной области видимости (например, внутри функции или блока кода) используется переменная с тем же именем, что и в глобальной области. В результате переменная внутри области видимости затеняет (перезаписывает) переменную с тем же именем из внешней области.

Будьте внимательны при задании переменных с одинаковыми именами – из-за затенения переменная внутри ограниченной области может иметь иное значение или тип, чем переменная с тем же именем в других частях программы.

Для примера рассмотрим код с созданием двух переменных без затенения с одинаковыми именами и разными значениями:

[ ]:
```
x = 10  # создание глобальной переменной

function func()
    x = 20  # создание локальной переменной (внутри функции) с тем же␣
  ↪именем, что и глобальная переменная
    println(x)  # будет использована локальная переменная внутри␣
  ↪функции
end

func()  # вызов функции
println(x)  # глобальная переменная не была изменена внутри функции,␣
  ↪поэтому выводится ее значение
```

Внутри функции создается локальная переменная с тем же именем x, что и глобальная. Это не изменяет глобальную переменную, а создает новую локальную переменную, видимую только внутри функции. В результате выводятся значения обеих переменных, равные 20 (глобальная) и 10 (локальная) соответственно. Таким образом, создание переменных в разных областях видимости помогает избежать проблем с перезаписью переменных с одинаковыми именами.

В явном виде область видимости переменной можно задать с помощью одного из двух ключевых слов: `global` (глобальная) или `local` (локальная).

Например, определим функцию `func`, в которой задается значение переменной x. Чтобы эта переменная было доступна за пределами функции, сделаем ее глобальной, написав ключевое слово `global` перед именем переменной.

[ ]:
```
function func()
    global x = 30
end
```

Вызов функции `func()` выведет значение переменной x, равное 30.

[ ]:
```
func()
```

Переменная x определена как глобальная, поэтому ее можно вызвать за пределами функции:

```
[ ]: println(x)
```

✏Задание 10

Создайте функцию `func`, принимающую два аргумента a и b. В теле функции задайте переменную x, равную сумме переменных a и b. Затем вызовите функцию `func` со значениями аргументов, равными 2 и 3, и выведите на экран переменную x.

```
[ ]:
```

Подсказка Поскольку переменная x должна быть доступна за пределами функции `func`, то x нужно определить как глобальную переменную (`global x`).

Решение

```
[ ]: function func(a,b)
         global x = a + b
     end
```

```
[ ]: func(2,3)
```

```
[ ]: println(x)
```

Оператор `let` создает **блок со строгой областью**. Это означает, что если локальной переменной y еще не существует и присваивание происходит внутри конструкции, образующей строгую область (т.е. внутри блока, ограниченного операторами `let` и `end`), то создается новая локальная переменная с именем y в области присваивания.

Рассмотрим пример:

```
[ ]: let y = 1
         let y = 2
         end
         y
     end
```

Здесь во внешнем блоке `let` создается локальная переменная y со значением 1. Во вложенном в него (внутреннем) блоке `let` создается своя локальная переменная y со значением 2. Вызов переменной y из внешнего блока выводит значение, присвоенное ей в этом блоке, т.е. 1. Если попытаться вызвать переменную y за пределами внешнего блока `let`, то возникнет ошибка, т.к. переменная y является локальной.

Можно использовать оператор `let` без аргументов, чтобы просто ввести новый блок области, не создавая новые привязки сразу же.

Следующий код полностью эквивалентен предыдущему:

[ ]:
```
let
    local y = 1
    let
        local y = 2
    end
    y
end
```

Тест для получения сертификата

Пройти тест по теме **Функции**.