\func

\func определяет функцию, константу или даже тип:

```
\func zero => 0                                    -- константа 0
\func one : Int => 1                               -- можно явно указать тип
\func id-int (x : Int) => x                        -- принимает x и возвращает x
\func id-string (x : String) : String => x         -- тут тоже можно указать тип
\func const-int (x : Int) (y : Int) : Int => x     -- функция двух аргументов
\func const-int' (x y : Int) : Int => x            -- то же самое, что const-int
\func IntArray : \Type => Array Int
```

Я рекомендую тип результата явно указывать, это упростит жизнь и Вам, и компилятору.

\lam

\lam — это известная нам всем λ:

```
\func id-int : Int -> Int => \lam (x : Int) => x
\func id-int' : Int -> Int => \lam x => x
```

Как видно по второму примеру, тип аргумента можно опустить, если он понятен из контекста.

\Pi

\Pi — это П-тип (тип зависимой функции).

```
\func create-list : \Pi (A : \Type) -> List A => \lam A => nil
```

Если принимать аргументы прямо в \func (без \lam), \Pi писать не требуется:

```
\func create-list' (A : \Type) : List A => nil
```

\Sigma

\Sigma — это Σ-тип (т.е. тип пары, возможно, зависимой пары).

```
\func add (p : \Sigma Nat Nat) : Nat => p.1 Nat.+ p.2
-- Не думайте про = сейчас, думайте про то, что (a, b) --- создание пары
\func increment (x : Nat) : \Sigma (y : Nat) (y = x Nat.+ 1) => (suc x, idp)
```

Первая функция принимает из пару двух чисел и возвращает их сумму. Вторая функция принимает число x и возвращает пару из другого числа и доказательства, что оно на один больше x. Тут Nat.+ — сложение натуральных чисел.

\Type

\Type — это $\star$ из обобщённой системы типов. На самом деле, не совсем, но это обсудим потом.

```
\func id (A : \Type) (a : A) => a
\func IntPair : \Type => \Sigma Int Int
```

Типы данных (\data)

\data помогает определять новые типы данных, которые выражаются через $\vee$ и $\wedge$:

```
\data Point3D
  | point3d Real Real Real -- \import Arith.Real
\data IntOrString
  | left Int
  | right String
```

Point3D — по сути конъюнкция трёх вещественных чисел (Real $\wedge$ Real $\wedge$ Real), а IntOrString — дизъюнкция Int и String (Int $\vee$ String).
Тут point3d, left и right называются конструкторами и по сути являются специальными функциями. Например, если `a : Int`, то `left a : IntOrString`.
Можно сделать и более сложные конструкции. Например, $\lambda \texttt{A}^*.(\texttt{A} \wedge \texttt{A}) \vee (\texttt{A} \wedge \texttt{A} \wedge \texttt{A})$:

```
\data Vector (A : \Type)
  | vector2d A A
  | vector3d A A A
```

То есть тут мы имеем тип с параметром: Vector Int — это тип, а Vector — функция из типа в тип. Также типы могут быть рекурсивными. Например, ниже приведено определение натуральных чисел по Пеано (именно так определяется тип Nat в стандартной библиотеке):

```
\data Nat
  | zero
  | suc Nat
```

Сопоставление с образцом (\case и \elim)

Как мы знаем, для $\alpha \vee \beta$ должно быть правило вывода, которое из $\alpha \to \gamma$, $\beta \to \gamma$ и $\alpha \vee \beta$ сделает γ. В любом функциональном языке такое есть и называется сопоставление с образцом:

```
\func pred (x : Nat) : Nat \elim x
  | zero => zero
  | suc y => y
\func pred' : Nat -> Nat => \lam x => \case x \with {
  | zero => zero
  | suc y => y
}
\func pred'' => \lam (x : Nat) => \case x \return Nat \with {
  | zero => zero
  | suc y => y
}
```

При определении функций я бы рекомендовал первый вариант.
Первый вариант также отличается тем, что во всех рассуждениях Arend заменяет x на zero либо suc y, а остальные не заменяет. Вот пример, который это пояснит:

```
\func example (n : Nat) : 2 Nat.* n = n Nat.+ n => \case n \with {
  | 0 => {?}      -- всё ещё нужно доказать 2 * n = n + n, а не 2 * 0 = 0 + 0
  | suc m => {?} -- всё ещё нужно доказать 2 * n = n + n
}
\func example' (n : Nat) : 2 Nat.* n = n Nat.+ n => \case \elim n \with {
  | 0 => {?}      -- нужно доказать 2 * 0 = 0 + 0
  | suc m => {?} -- нужно доказать 2 * (suc m) = (suc m) + (suc m)
}
\func example'' (n : Nat) : 2 Nat.* n = n Nat.+ n \elim n
  | 0 => {?}      -- то же самое, что в example'
  | suc m => {?} -- то же самое, что в example'
```

При определении функции, лучше использовать последний вариант. В остальных случаях — первый или второй в зависимости от того, что вам нужно.

Пробелы в доказательстве

В любом месте определения можно написать `{?}`, чтобы компилятор подсказал, значение какого типа должно быть вместо `{?}`. Например, в случае ниже компилятор подскажет, что известно `x : A` и нужно значение типа `B -> A`.

```
\func test (A B : \Type) : A -> B -> A => \lam x => {?}
```

Неявные аргументы функций

Аргумент функции можно обернуть в фигурные скобки, тогда его не надо будет указывать, и Arend попытается сам его угадать:

```
\func id {A : \Type} (x : A) : A => x
\func test => id 1          -- выведется A = Nat
\func test' => id {Nat} 1 -- неявный аргумент можно явно указать
```

Рекурсия

Очевидно, произвольная рекурсия помешала бы доказательствам, иначе вот так можно было бы доказать, что любое число равно нулю:

```
\func invalid (x : Nat) : x = 0 => invalid x
```

Поэтому введено такое правило: аргументы рекурсивного вызова должны быть **структурно проще** оригинального аргумента:

```
\func add (x y : Nat) : Nat \elim y
  | 0 => x
  | suc z => suc (add x z) -- y это suc z, z проще, чем suc z
```

Важно что аргумент должен быть именно структурно проще: в памяти Arend хранит просто выражения, и выражение `z` проще, чем выражение `suc z`. А, например, `Nat.div x 2` не проще, чем `x`, поэтому следующий пример не компилируется, хотя рекурсия там всегда конечна:

```
\func bit-length-invalid (x : Nat) : Nat \elim x
  | 0 => 0
  | x => suc (bit-length-invalid (Nat.div x 2))
```

Исправить это можно так: взять число, которое будет точно больше суммарного количества итераций и создать вспомогательную функцию, которая будет принимать это число и делать рекурсивный вызов только если это число не ноль.
Более детально, это можно сделать, например, так: создаём функцию `helper`, которая принимает `x : Nat` (аргумент), `it : Nat` (количество итераций) и `x<=it : x <= it` — доказательство, что `x` меньше либо равен `it`. Функция делает следующее: если `x` равно нулю, вернуть ноль. Если `x` не равно нулю, а `it` равно нулю, получаем противоречие. Если оба не ноль, докажем, что `Nat.div x 2 <= it - 1` и сделаем рекурсивный вызов. Полный код приведён в репозитории.

Инфиксные операторы

Функции определить так, чтобы ими можно было пользоваться как инфиксными операторами при помощи `\infix`, `\infixl` и `\infixr`:

```
\func \infixl 6 plus (n m : Nat) : Nat \elim m
  | 0 => n
  | suc m' => suc (n plus m') -- здесь используется тот факт, что plus инфиксный
\func \infixl 7 mul (n m : Nat) : Nat \elim m
  | 0 => 0
  | suc m' => plus (mul n m') n -- инфиксные функции можно вызывать как обычные
```

Числа после `\infixl` — приоритеты операций: `x plus y mul z` значит `x plus (y mul z)`.
`\infixl` определяет инфиксный оператор с левой ассоциативностью: `x plus y plus z` значит `(x plus y) plus z`, `\infixr` — с правой ассоциативностью: `x ++ y ++ z` значит `x ++ (y ++ z)`, `\infix` — без ассоциативности: `x < y < z` не компилируется.
Обычную функцию (в данном случае `\data`) можно превратить в инфиксную, окружив её обратными кавычками:

```
\func or-left {A B : \Type} (a : A) : A `Or` B => inl a -- \import Data.Or
```

Ещё можно ставить только левую кавычку. Что это будет значит, зависит от контекста. `` `- 1 `` — функция, которая вычитает из аргумента единицу. `` 10 `- `` — функция, которая вычитает аргумент из 10.