ФУНКЦІОНАЛЬНЕ ПРОГРАМУВАННЯ

Лекція 13

ЧИСТО ФУНКЦІОНАЛЬНІ МОВИ ПРОГРАМУВАННЯ

2020

Повний текст лекції буде розміщений на сайті baklaniv.at.ua

Мова програмування MIRANDA

Miranda - функціональна мова програмування, створена в 1985 році Девідом Тернером в якості стандарта функціональної мови. Має строгу полиморфную систему типів, підтримує типи даних користувача.

Як і мова ML, викладається у багатьох університетах. Функціональні об'єкти будуються за допомогою каррінга (часткового застосування) існуючих функцій. Володіє ледачою семантикою. Програма представляє собою безліч визначень.

Наступник мов SASL і Kent Recursive Calculator, який використовує деякі концепції ML і Hope. Мала великий вплив на розробників мови Haskell.

Назва мови походить від імені героїні п'єси «Буря» Вільяма Шекспіра, Міранди. У ній вона вимовляє фразу ***«Прекрасний новий світ!»***, що, за словами творців, має принести ***«чудовий новий світ в функціональне програмування»***. Також героїня зображена на логотипі мови.

Міранда був вперше випущена в 1985 році, в якості швидкого перекладача до мови C для Unix-flavour операційних систем, з подальшими випусками в 1987 і 1989 роках Miranda мала сильний вплив на подальшу розробку - мову Haskell.

Міранда є лінивою, чисто функціональною мовою програмування. Тобто, вона не має побічних ефектів і імперативного програмування функцій.

Програма Miranda (так званий скрипт) являє собою набір рівнянь, що визначають різні математичні функції і алгебраїчні типи даних. Слово набір важливо тут: порядок рівнянь, взагалі кажучи, не має значення, і немає необхідності визначати об'єкт до його використання.

Комментарий вводится в обычные скрипты символами || и продолжается до конца той же линии.

Альтернатива комментирования затрагивает весь файл исходного кода, известный как « грамотный сценарий », в котором каждая строка считается комментарием , если он не начинается с знака >.

Основные Миранды типы данных являются `char`, `num` и `bool`. Строка символов является просто списком `char`, а `num` конвертируются между двумя формами: произвольной точности целых чисел (bignums) по умолчанию, и регулярные с плавающей запятой значения по мере необходимости.

Кортежи являются последовательностями элементов потенциально смешанных типов, аналогичные записи в Паскале-подобных языков, и написаны с разделителями в скобках:

```
  this_employee = ("Folland, Mary", 10560, False, 35)
```

Список наиболее часто используемая структура данных в Миранде. Она написана в квадратные скобки и разделенных запятыми элементов, каждый из которых должен быть того же типа:

```
  week_days =
["Mon","Tue","Wed","Thur","Fri"]
```

Список конкатенация `++`, вычитание `--`, строительство `:`, определение размера `#` и индексации `!`, задается так:

```
 days = week_days ++ ["Sat","Sun"]
 days = "Nil":days
 days!0
⇒ "Nil"
 days = days -- ["Nil"]
 #days
⇒ 7
```

`..` используются для списков, элементы которых образует арифметический ряд, с возможностью задания приращения , отличного от 1:

```
fac n   = product [1..n]
odd_sum = sum [1,3..100]
```

```
squares = [ n * n | n <- [1..] ]
```

Читается: список n в квадрате, где n берется из списка всех положительных целых чисел и ряд, где каждый член является функцией предыдущего, например:

```
powers_of_2 = [ n | n <- 1, 2*n .. ]
```

Поскольку эти два пример предполагает, Miranda позволяет списки с бесконечным числом элементов, простейшие из которых является список всех положительных целых чисел: `[1..]`

Обозначения для применения функции просто соседство, как и в `sin x`.

В Miranda, как и в большинстве других чисто функциональных языках, функции первого класса, который должен сказать , что они могут быть переданы в качестве параметров других функций, возвращается в качестве результата, или включены в качестве элементов структуры данных.

Более того, функция требует два или более параметров может быть «частично параметризованной», или кэррирования , путем подачи меньшего , чем полное число параметров.

Это дает еще одну функцию , которая, учитывая остальные параметры, будет возвращать результат. Например:

```
add a b = a + b
increment = add 1
```

дает способ создания функции «приращение», который добавляет один к своему аргументу.

На самом деле, `add 4 7` имеет функцию двух параметров `add`, применяет его к `4` получению функции одного параметра , который добавляет четыре к своему аргументу, то применяется что `7`.

Любая функция принимает два параметра может быть превращена в оператор инфиксного (например, учитывая определение `add` функции выше, термин `$add` во всех отношениях эквивалентных к оператору `+`) , и каждый оператор инфиксным принимает два параметра может быть превращен в соответствующую функцию.

Таким образом:

```
increment = (+) 1
```

это кратчайший способ создать функцию, которая добавляет один к своему аргументу. Аналогичным образом, в

```
half = (/ 2)
reciprocal = (1 /)
```

две функции одного параметра генерируются.

Хотя Миранда строго типизированный язык программирования , он не настаивает на явных типа деклараций . Если тип функции , является явно не объявлен, интерпретатор выводит его из вида его параметров и как они используется в функции.

В дополнение к основным типам (`char`, `num`, `bool`), она включает в себя тип «ничего» , где тип параметра не имеет значения, как и в функции списка реверсирования:

```
rev [] = []
rev (a:x) = rev x ++ [a]
```

которые могут быть применены к списку любого типа данных, для которых явное объявление типа функции будет: `rev :: [*] -> [*]`

Наконец, Миринда имеет механизмы для создания и управления программными модулями , чьи внутренних функции являются невидимыми для программ , вызывающих этих модули.

Haskell

Haskell — стандартизованный **чистый** функциональный язык программирования общего назначения. Другие обсуждаемые функциональные языки Ocaml или Scala — считаются смешанными языками, так как помимо функционального поддерживают и императивный стиль вычислений. Haskell не допускает императивного программирования. Является одним из самых распространённых языков программирования с поддержкой отложенных вычислений. Типизация языка Haskell **строгая**, **статическая**, с автоматическим выводом типов. Строгое отношение к типизации (что не характерно, вообще говоря, для функциональных языков) — ещё одна отличительная черта Haskell. Поскольку язык функциональный, то основная управляющая структура — это функция.

В 1990 г. была предложена первая версия языка, Haskell 1.0. Непосредственно на него оказал очень сильное влияние язык Miranda, разработанный в 1985 г. Дэвидом Тёрнером (Миранда была первым чистым функциональным языком). Но выход Haskell в «широкий свет» начался только в 2003 г. — таким образом, в течение 13 лет этот язык был уделом лабораторий, главным образом математически ориентированных.

Порог вхождения в программирование на Haskell высок. Во-первых, из-за его происхождения из кругов абстрактных математиков и из-за формулирования его понятий в терминах понятий из абстрактной математики (теории категорий). Другая причина, связанная с предыдущей, из-за которой и сложилось устойчивое ложное представление о колоссальной сложности языка Haskell, — это **отсутствие** внятных описаний и руководств. А официальная документация Haskell выкладывается также в виде строгих **формальных определений**, на изучение которых могут уйти месяцы. Например, одно из важных понятий и терминов в языке — «монада» (от греческого μονάς, «единица»), пришедшие в Haskell именно из теории категорий.

С другой стороны, Haskell является языком, строго организующим мышление программиста. В ряде ведущих университетов мира именно Haskell выбран как первый язык обучения студентов 1-го курса «**искусству** программирования» (по определению Д.Кнутта).

Существует несколько реализаций Haskell доступных в Linux, но компилятор GHC стал фактическим стандартом в отношении новых возможностей языка:

```
$ yum list ghc
Доступные пакеты
ghc.i686   7.6.3-18.3.fc20     updates
$ sudo yum install ghc
...
Установить  1 пакет (+47 зависимых)
Объем загрузки: 82 M
Объем изменений: 610 M
...
Установлено:
  ghc.i686 0:7.6.3-18.3.fc20
$ ghc --version
The Glorious Glasgow Haskell Compilation System, version 7.6.3
```

Легко видеть, что объём изменений эта инсталляция потянет значительный. В пакете будет установлен одновременно диалоговый интерпретатор Haskell, полезный для отработки конструкций языка, он же может выполнять отдельные приложения:

```
# ghci
GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude> 2+2
4
Prelude> ^D
Leaving GHCi.
```

Но Haskell — это целая своеобразная технология, и в этой технологии есть ещё такая штука как **cabal** (**C**ommon **A**rchitecture for **B**uilding **A**pplications and **L**ibraries). Говорят, что это нечто типа make в мире C/C++ — построитель проектов Haskell. Но cabal придётся устанавливать отдельно:

```
$ yum list cabal*
Загружены модули: langpacks, refresh-packagekit
Доступные пакеты
cabal-dev.i686      0.9.2-2.fc20                    fedora
cabal-install.i686    1.16.0.2-27.fc20            updates
cabal-rpm.i686          0.8.7-1.fc20                updates
$ sudo yum install cabal*
...
Установить  3 пакета (+13 зависимых)
Объем загрузки: 1.7 M
Объем изменений: 9.1 M
...
Выполнено!
```

Кроме того, что это инструмент построения собственных модульных проектов, это мощнейшее средство управления модулями (библиотеками) Haskell:

```
$ cabal --help
...
Commands:
  install      Installs a list of packages.
  update       Updates list of known packages
  list         List packages matching a search string.
  info         Display detailed information about a particular
package.
  fetch        Downloads packages for later installation.
  unpack       Unpacks packages for user inspection.
  check        Check the package for common mistakes
  sdist        Generate a source distribution file (.tar.gz).
  upload       Uploads source packages to Hackage
  report       Upload build reports to a remote server.
  init         Interactively create a .cabal file.
  configure    Prepare to build the package.
  build        Make this package ready for installation.
  copy         Copy the files into the install locations.
  haddock      Generate Haddock HTML documentation.
  clean        Clean up after a build.
  hscolour     Generate HsColour colourised code, in HTML format.
  register     Register this package with the compiler.
  test         Run the test suite, if any (configure with
UserHooks).
  bench        Run the benchmark, if any (configure with
```

С помощью cabal вы можете найти, выбрать и установить любой модуль (библиотеку) из главного мирового репозитория Haskell (его называют Hackage):

```
$ cabal list complex
* complex-generic
    Synopsis: complex numbers with non-mandatory RealFloat
    Default available version: 0.1.1
    Installed versions: [ Not installed ]
    Homepage: https://gitorious.org/complex-generic
    License:  BSD3

* complex-integrate
    Synopsis: A simple integration function to integrate a
complex-valued
              complex functions
    Default available version: 1.0.0
    Installed versions: [ Not installed ]
    Homepage: https://github.com/hijarian/complex-integrate
    License:  PublicDomain

* complexity
    Synopsis: Empirical algorithmic complexity
    Default available version: 0.1.3
    Installed versions: [ Not installed ]
```

```
    License:  BSD3
22
23* storable-complex
24    Synopsis: Storable instance for Complex
25    Default available version: 0.2.1
26    Installed versions: [ Not installed ]
27    License:  BSD3
```

Библиотека Haskell очень обширна (1-я команда выведет полный листинг библиотеки):

```
1$ cabal list >> cabal.lst
2$ ls -l cabal.lst
3-rw-rw-r--. 1 Olej Olej 1273606 мар  9 11:40 cabal.lst
4$ wc -l cabal.lst
541520 cabal.lst
```

Учитывая, что информация о каждом модуле выводится в 6 строк (см. выше) — это даёт объём библиотеки в 6920 единиц компиляции.

Файлы исходного кода Haskell имеют расширения .hs или .lhs. О дним из фундаментальных свойств языка Haskell, которым программисты пугают друг друга, является полное отсутствие в нём **оператора** присваивания. В написании реализации нашей тестовой задачи, в отношении Haskell мы пойдём другим путём, отличающимся от того, как это делалось в других языках: мы не станем вручную компилировать из командной строки файл кода triangle.hs, а создадим проект, пользуясь возможностями cabal по созданию и управлению проектами.

Примечание: Конечно, вы можете откомпилировать полученный ниже файл кода задачи и вручную, предварительно переименовав его в triangle.hs .

Создадим для начала вручную файловую инфраструктуру проекта, например так:

```
$ mkdir triangle
$ cd triangle
$ mkdir src
$ cd src
$ touch Main.hs
$ cd ..
$ tree
.
└── src
    └── Main.hs
```

Теперь, находясь в корне дерева проекта (каталог triangle), выполним построение проекта командой, которая проведёт диалог настройки проекта, вопросы которого достаточно понятны:

```
$ cabal init
Package name? [default: triangle]
Package version? [default: 0.1.0.0]
Please choose a license:
 * 1) (none)
   2) GPL-2
   3) GPL-3
   4) LGPL-2.1
   5) LGPL-3
   6) BSD3
   7) MIT
   8) Apache-2.0
   9) PublicDomain
  10) AllRightsReserved
  11) Other (specify)
Your choice? [default: (none)] 7
Author name?
Maintainer email?
Project homepage URL?
Project synopsis?
Project category:
 * 1) (none)
   2) Codec
   3) Concurrency
   4) Control
```

В каталоге src могут быть созданы дополнительные файлы кода к проекту (.hs) или каталоги (например Utils), содержащие такие файлы. Дополнительные файлы кода будут содержать код **модулей**, которые компонуются в проект. Имена файлов и каталогов в src лучше именовать **с заглавной буквы** — это связано с именованием и импортом модулей в Haskell.

После генерации в файловой иерархии появилось 2 файла: Setup.hs, который нас не интересует, и файл конфигурации проекта triangle.cabal, в котором мы будем неоднократно редактировать строки по ходу развития проекта (сами параметры строк уже записаны в файл в виде комментариев, нам предстоит раскомментировать их и вписать им значения). Прежде всего, нужно (обязательно) определить файл кода с которого стартует приложение (Main.hs, он может иметь произвольное имя):

```
1...
2executable triangle
3  ghc-options:         -W
4  main-is:             Main.hs
5  build-depends:       haskell98 >=2.0.0.2 , exceptions
6...
```

Здесь показаны только строки, подвергшиеся изменению (в порядке, в котором они указаны) в том исходном файле triangle.cabal, который был создан при построении проекта:

- определение включить вывод предупреждений компиляции, не только ошибок;
- определить файл Main.hs как стартовый (на самом деле имена файлов кода могут быть произвольными);

- описать импорт дополнительных стандартных пакетов (библиотек): в данном случае пакет haskell98 содержит модуль Complex для работы с комплексными числами, а пакет exceptions — обработку исключений;

Теперь нам осталось сконфигурировать проект под наши правки (конфигурацию лучше делать **каждый раз** после редактирования triangle.cabal):

```
$ cabal configure
Resolving dependencies...
Configuring triangle-0.1.0.0...
Warning: The 'license-file' field refers to the file 'LICENSE'
which does not exist.
```

Всё! Проект готов. Дальше нам предстоит наполнять смыслом файлы исходного кода (Main.hs) и **компилировать** проект. Вот как может выглядеть код сравниваемой задачи в упрощённой реализации на Haskell (упрощение касается только отсутствия обработки ошибок ввода пользователя — чтобы не перегружать код):

Листинг 6. Реализация задачи на языке Haskell (файл Main.hs каталог triangle):

```
module Main where
import Complex
import Numeric
import IO

```

```
  {- код проверен для версии:
  $ ghc --version
  The Glorious Glasgow Haskell Compilation System, version 7.6.3
  -}

  type Point = Complex Double       -- синоним координатной точки

  get_coord :: String -> Point      -- декодирование строки ввода в
6 координаты x % y
7 get_coord str =
8    f( words str )
9    where
10      f :: [String] -> Point
11      f lst = ( ( read( lst !! 0 ) :: Double ) :+ ( read( lst !!
12 1 ) :: Double ) )
13
14 try_to_input :: IO String        -- ввод строки с ожиданием ^D
15 try_to_input = do
16   line <- hGetLine stdin `catch` (\e -> if IO.isEOFError e then
17 return
18   [] else ioError e)
19   return line
20
21 get_shape :: [Point] -> IO [Point]
22 get_shape shape = do              -- рекурсивный ввод списка
```

Исходный код на Haskell **не является форматно независимым**: его смысл зависит от отступов новых строк, переносов строк и других вещей, связанных с размещением кода. Это достаточно редкий случай для языков программирования, и здесь (только в написании) Haskell близок с Python.

В показанном фрагменте кода есть достаточно много: и лямбда-определения функций, и сопоставления с образцом, и обработка исключений (в определении конца ввода, ситуации EOF), и использование рекурсии.

Теперь мы готовы **компилировать** полученный **проект**:

```
$ cabal build
Building triangle-0.1.0.0...
Preprocessing executable 'triangle' for triangle-0.1.0.0...
[1 of 1] Compiling Main ( src/Main.hs,
dist/build/triangle/triangle-tmp/Main.o )
src/Main.hs:3:1: Warning:
    The import of `Numeric' is redundant
      except perhaps to import instances from `Numeric'
    To import instances alone, use: import Numeric()
src/Main.hs:42:1: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for `show_shape': Patterns not matched: []
src/Main.hs:50:10: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for `summa': Patterns not matched: [] _
src/Main.hs:55:1: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for `square': Patterns not matched: []
src/Main.hs:62:10: Warning:
    Pattern match(es) are non-exhaustive
    In an equation for `triang': Patterns not matched: _ []
Linking dist/build/triangle/triangle ...
```

Почему так много предупреждений? Не знаю... Могу предположить по смыслу, что все они (связанные только с сопоставлением с образцом) используют синтаксические конструкции, заимствованные из

описаний прежних версий (Haskell 98). А очень свежий компилятор (Haskell 2010) хотел бы более современных определений образцов. Предоставим читателям возможность и право самостоятельно улучшить код в этом направлении.

После правок, конфигураций и компиляций дерево проекта имеет структуру:

```
$ tree
.
├── dist
│   ├── build
│   │   ├── autogen
│   │   │   ├── cabal_macros.h
│   │   │   └── Paths_triangle.hs
│   │   └── triangle
│   │       ├── triangle
│   │       └── triangle-tmp
│   │           ├── Main.hi
│   │           └── Main.o
│   ├── package.conf.inplace
│   └── setup-config
├── Setup.hs
├── src
│   └── Main.hs
└── triangle.cabal
```

Здесь поддерево dist и есть, собственно, каталогом сборки. Запускать откомпилированное приложение на тестирование мы можем прямо из каталога проекта:

```
$ ./dist/build/triangle/triangle
координаты вершин в формате: X Y
вершина № 1 : 1.00001 1.00003
вершина № 2 : 1.00003 2.0003
вершина № 3 : 2.0004 1.00005
вершин 3 : [2.00,1.00] [1.00,2.00] [1.00,1.00]
периметр = 3.42
площадь = 0.50
--------------------------------
координаты вершин в формате: X Y
вершина № 1 : ^C
```

Как видим, оно не сильно отличается от того, что мы видели в реализациях на других языках программирования.

После построения проекта, симметрично, очищаем следы его создания:

```
$ cabal clean
cleaning...
```

Для того, чтобы не быть голословным относительно **ручной** компиляции отдельных файлов Haskell кода, о которой упоминалось выше, продемонстрируем его на простейшем приложении, которое заодно

проверит, как реализация языка ведёт себя с Unicode, кодировкой UTF-8 и русским текстом (это всегда нужно делать для начала). Само «приложение»:

Листинг 7. Простейшее приложение на языке Haskell:

```
module Main where
import System.Environment

main :: IO ()
main = do              -- сама программа
     args <- getArgs  {- вложенный комментарий -}
     putStrLn( "Привет от Haskell, " ++ args !! 0 )
```

Его ручная компиляция

```
$ ghc -o hello_hs hello_hs.hs
[1 of 1] Compiling Main             ( hello_hs.hs, hello_hs.o )
Linking hello_hs ...
```

Ручное выполнение:

```
$ ./hello_hs Вася
Привет от Haskell, Вася
```

Основные типы

Типом называется множество значений, разделяющих некоторые свойства. Примеры типов, постоянно встречающихся программисту: целые числа, символы, булевы значения.

Если выражение `e` имеет тип `T`, то мы пишем `e :: T`. Это называется **описанием типа** или **сигнатурой**. `::` читается «имеет тип».

Haskell – язык со **строгой типизацией**. В нем функции работают со значениями определенного типа, и попытка применить функцию к аргументу неправильного типа вызовет ошибку. Ошибки выявляются на стадии компиляции, поэтому типизация называется статической.

Строгая статическая типизация избавляет от существенной части ошибок в программах.

В Haskell не используется неявное приведение типов (например, целое число не будет по усмотрению компилятора трактоваться как число с плавающей точкой). Но, в отличие от других языков, в Haskell не требуется указывать тип каждого значения или выражения. Используется выведение типов.

В ghci тип значения можно узнать при помощи `:type`:

```
ghci> :type True
True :: Bool
ghci> :type 'a'
'a' :: Char
ghci> :type 23 < 529
23 < 529 :: Bool
```

Здесь мы рассмотрим основные типы и средства их комбинирования.

Единичный тип

Самый простой тип – единичный. Он записывается в виде `()` и содержит единственное значение `()`.

```
ghci> :type ()
() :: ()
```

Он похож на `void` в языках вроде C и Java.

Булевы значения

Тип `Bool` состоит из значений `False` и `True`. С ними можно манипулировать при помощи операторов `&&` и `||`, а также функции `not`.

```
ghci> True || False
True
ghci> True & False
False
ghci> not True
False
```

Символы

Тип `Char` соответствует множеству символов Unicode. Значения записываются в одинарных кавычках: `'a', 'b', 'c'`.

Как и в других языках, для ввода специальных символов можно использовать экранирование: `'\''` (кавычка), `'\n'` (перевод строки), `'\t'` (табуляция). Подробности см. в Haskell 98 Report.

Числа

Для чисел имеются следующие базовые типы:

- `Int` – целое число фиксированной точности (минимум 29 битов).
- `Integer` – целое число произвольной точности.
- `Float` – число с плавающей точкой одинарной точности.
- `Double` – число с плавающей точкой двойной точности.

`Integer` лучше `Int`, так как не вызывает опасности переполнения, но арифметика для чисел произвольной точности медленнее, чем арифметика фиксированной точности.

Операции с плавающей точкой оптимизированы для `Double`; использовать `Float` в целях экономии нет смысла.

Кортежи

Кортежем называют последовательность **компонентов** фиксированной длины.

Компоненты перечисляются через запятую в круглых скобках: `(1,'a'), (23,"bob",23)`, и т. д. Тип кортежа с компонентами типов T1, …, Tn записывается как (T1, …, Tn).

Разное количество компонентов и их типы дают разные, не совместимые типы кортежей.

Кортежи с двумя компонентами называются парами, с тремя – тройками, и т. д. Кортежей из одного компонента нет; пустой кортеж (), как мы уже знаем, имеет единичный тип.

Для пар в Prelude определены проекторы, возвращающие первый и второй компоненты:

```
fst (x,y) = x
snd (x,y) = y
```

Имейте в виду, что в Haskell запись вида `fst (x,y)` означает применение функции к одному аргументу-паре.

Функции

Функция типа `f :: X -> Y` сопоставляет элементам `e` типа `X` элементы `f e` типа `Y`.

Перед тем как определять функцию, желательно указать ее тип.

Тип может быть выведен автоматически, но он может оказаться более общим, чем предполагаемый, либо вообще не тем – это значит, что вы в чём-то ошиблись. Система сверяет ваши типы с выведенными.

Например, функция возведения в квадрат числа с плавающей точкой двойной точности определяется так:

```
1square :: Double -> Double
2square x = x*x
```

Тип аргумента должен совпадать с типом, указанным при объявлении функции (или выведенным). Это составляет основу выведения типов: если $f :: X \to Y$ и $e :: X$, то $f\ e :: Y$.

Каррирование

В математике формально не существует функций нескольких аргументов. Речь может идти об отображении пар, троек, и т. д. В Haskell подобное записывается в виде

```
1f' :: (X,Y) -> Z
```

Более распространенный в Haskell подход называется (в честь Хаскелла Карри) каррированием. Функцию n аргументов можно представить в виде функции, принимающей 1-й аргумент и возвращающей функцию,

которая принимает 2-й аргумент и т. д. Окончательно, n-я функция будет принимать n-й аргумент и вычислять результат.

Для двух аргументов тип записывается в виде

```
f :: X -> (Y -> Z)
```

Неформально, `f` принимает два аргумента типа `X` и `Y`. Применение `f` к значениям `a` и `b` происходит в два этапа:

- Если `a :: X`, то `f a :: Y -> Z`.
- Если `b :: Y`, то `(f a) b :: Z`.

Промежуточное выражение `f a` называется **частичным применением**.

- Применение функции левоассоциативно: запись `f a b` эквивалентна `(f a) b`.
- Оператор `->` правоассоциативен: тип `f :: X -> Y -> Z` интерпретируется как `f :: X -> (Y -> Z)`.

Например, функцию для вычисления гипотенузы в прямоугольном треугольнике можно сделать каррированной, а можно некаррированной:

```
hyp :: Double -> Double -> Double
hyp x y = sqrt (square x + square y)

hyp' :: (Double, Double) -> Double
hyp' (x,y) = sqrt (square x + square y)
```

В первом случае мы не выписываем в явном виде, что `hyp` зависит только от `x`, но возвращает другую функцию, зависящую от `y`. Функции каррируются по умолчанию.

Существенной разницы между `hyp' (x,y)` и `hyp x y` нет, но последний вариант выглядит проще и допускает частичное применение.

Если записать `hyp x`, то, как следует из определения каррирования, это будет функция типа `Double -> Double`.

Важной особенностью функциональных языков является работа с функциями как с объектами первого класса – с ними можно манипулировать на тех же условиях, что и с другими значениями.

Функция, принимающая функции в качестве аргументов и возвращающая в качестве результата, называется функцией высокого порядка.

Переход между функцией `(a,b) -> c` и `a -> b -> c`, где a, b, c – некоторые типы, можно выразить специальными функциями:

```
curry f x y     = f (x,y)
uncurry f (x,y) = f x y
```

`curry` и `uncurry` достаточно применить к одному аргументу, а именно к функции `(a,b) -> c` или `a -> b -> c`, и мы получим функцию, принимающую два аргумента через каррирование, либо в виде пары.

```
ghci> hyp 3 4
25.0
ghci> hyp' (3,4)
45.0
ghci> (uncurry hyp) (3,4)
65.0
ghci> (curry hyp') 3 4
85.0
```

Все эти соображения естественно переносятся на каррирование функций трех, четырех аргументов и т. д. (Но `curry` и `uncurry` определены в Prelude для функций двух аргументов.)

На первый взгляд, каррирование не приносит ничего нового при полном применении функции. Однако частичное применение позволяет фиксировать параметры более общих функций.

На наступній лекції ми продовжимо розгляд мови програмування Haskell.