

Функциональное программирование. Введение

Косарев Дмитрий

матмех

2 и 9 сентября 2024 г.

Дата сборки: 8 сентября 2024 г.



В этих слайдах

Галопом по Европам по основным особенностям *типизированного* функционального программирования

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

- Источник идей для других языков
Область научных исследований в Computer Science (одна из)
- Хранилище новых идей для программистов, которым не достаточно общепринятых подходов
- Практическое применение (непосредственно production)

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

Глобально функциональные языки делятся на две группы

- Наследники ML (статически типизированные)
- Наследники Scheme (динамически типизированные)

Определение 1

Определение (Functional language)

— language where functions are “first class values”

Определение (Функциональный язык (плохое определение))

— где в функции можно передавать как значения в другие функции

- 👎 Под плохое определение подходит даже Си
- 👎 Под него не подходит LISP

Определение 2

Определение (Функциональный язык (плохое определение))

— основанный на λ -исчислении

🗨 Не очень понятно что такое «основан на»?

🗨 «основан на» \equiv ? «можно реализовать»

- Если синтаксическая конструкция есть, но компилятор λ -исчисление не использует, то это «основан на»?
- Очень широкое определение, сюда подходит даже C++

```
// Y-combinator for the int type
```




```
boost::function<int(int)>  
    y(boost::function<int(boost::function<int(int)>,int)> f)  
{  
    return boost::bind(f, boost::bind(&y, f) ,_1);  
}
```

⁰<https://stackoverflow.com/a/154267/1065436> (проверено: 1 сентября 2024 г.)

Определение 3

Определение (Функциональный язык)

— язык, достаточно похожий (наследник) ML или Scheme

-  Полезное определение
-  Не очень формально
-  Может ограничить появление новых функциональных языков

Процедурная парадигма

- Программа записывается как набор процедур (функций), что вызывают друг друга
- Обычно различается три фазы: подготовка данных, анализ, выдача результата
- Обычно процедуры вызывают друг друга иерархично
- Обычно глобальное состояние расщепляется на части и передается как аргументы процедур (подвид императивного)

Определение (Функциональная парадигма)

Процедурная парадигма, где порицается изменяемое состояние

Определение (Чисто функциональная парадигма и язык)

В нём нет возможности сделать изменяемое состояние

Чистые (pure) функции

Определение

Чистая функция обладает следующими двумя свойствами

- Детерминированная: результат зависит только от аргументов
 - Нельзя использовать глобальные изменяемые данные
 - Нельзя запросить дату на компьютере и в зависимости от неё, что-то делать
- В процессе работы не совершающая «побочных эффектов»
 - Нельзя печатать, что-то на консоль
 - Вызывать внутри себя «нечистые» функции
 - На счет исключений бывают тонкости [\[1\]](#)

Замечания:

- Это свойство *функций*, а не языка программирования
- Их проще отлаживать, некоторые оптимизации (например, inlining) становятся чаще применимы
- Если до этого программировали только с изменяемым состоянием, то перестроиться может быть сложно

Домашнее задание подразумевает программирование на типизированном языке OCaml (наследник ML)

Присваивание *запрещено* из педагогических соображений

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

Другой синтаксис вызова функций: каррирование (currying)

Или шенфинкелизация



Хаскелл Карри (1900–1982)

В наследниках Си: $f(x, g2(y1, y2), z)$

Сигнатуры: $\text{float } f(\text{int } x, \text{int } y, \text{int } z)$

В OCaml (и некотором другом ФП):

$f \ x \ (g2 \ y1 \ y2) \ z$

Сигнатуры:

$f : \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{float}$

Такой вид синтаксиса называется
каррированием

Но можно и без кар-
рирования (как в Си):

$f(x, g2 \ y1 \ y2, z)$

...

$f: \text{int} * \text{int} * \text{int} \rightarrow \text{float}$

$g: \text{int} \rightarrow \text{int} \rightarrow \text{int}$

Каковы явные преимущества каррированных функций?

Можно не писать всюду скобки и запятые.

Например, можно описать API из функций `start`, `fin`, `push`, `add`, `mul` и писать код например так:

```
start push 1 push 2 add push 3 mul fin
```

И выражение будет иметь тип `int` и вычисляться в 9

Упражнение (★★★)

Опишите 5 функций для вычисления арифметического выражения (в польской нотации, стеком) так, чтобы их можно было вызывать как показано выше

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

Встроенные в синтаксис кортежи (n-ки, tuples)

```
let triple: (int, string, float) =  
(1, "two", 3.0)
```

Количество элементов в кортежах не ограничено.

Доступ к элементам кортежа

```
let (n, _, _) = triple in  
assert (n = 1)
```

А в общем виде с помощью сопоставления с образцом (*pattern matching*) (будет ниже)

Когда вызываем функции в стиле Си, мы передаем не n аргументов, а кортеж из n аргументов

```
f: int * int * int -> float  
...  
let result = f triple in  
...
```


1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

Область видимости

```
let c =  
  (* c is not bound *)  
  let a =  
    let rec fac n =  
      (* a, fac, n are bound *)  
      if n < 2 then 1  
      else n * fac (n-1)  
    in  
      fac 3  
  in  
    make a  
  (* a is not bound *)
```

```
val c = run() {  
  val a = 42  
  CreateCommand(a)  
} /* a is not bound */
```

Области видимости *не как в Си*

Что-то подобное есть в kotlin, но больше букв

Последнее выражение в блоке — результат

Структурное программирование

В OCaml нет return, break, continue, goto
Исключения есть

Упражнение (Какая реализация проверки високосного года лучше?)

```
/* C */
bool IsLeap(unsigned year) {
    if (year%400 == 0)
        return true;
    if (year%100 == 0)
        return false;
    return (year%4 == 0);
}
```

```
(* OCaml *)
let is_leap year =
    if year mod 400 = 0 then
        true
    else if year mod 100 = 0 then
        false
    else year mod 4 = 0
```

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

У процедурного программирования есть недостаток — аргументы надо передавать явно. Это можно решать по-разному

Определение (Объект)

Блок кода (читайте: метод); а также данные, которые живут дольше, чем блок, где они были объявлены

Замыкания

Задача

Дана функция `make` с типом `a -> b -> c`, и значение с типом `a`. Хочется, для фиксированного значения типа `a`, отдать наружу функцию `b -> c`, которая будет строить значения типа `c` из значений типа `b`.

Стандартный подход из мира
ООП:

```
// kotlin
```

```
fun make(a: Int, b: Float) : String = ...
```

```
class CreateCommand(val a: Int)
```

```
{
```

```
    fun execute(b: Float): String = make(a, b)
```

```
}
```

Замыкания

Задача

Дана функция `make` с типом $a \rightarrow b \rightarrow c$, и значение с типом a . Хочется, для фиксированного значения типа a , отдать наружу функцию $b \rightarrow c$, которая будет строить значения типа c из значений типа b .

```
let make : a -> b -> c = fun a b -> ...  
let make_of_b : b -> c =  
  let arg : a = ... in  
  make arg
```

Из функции `make_of_b` вернули частично примененную функцию, а аргумент хранится в *замыкании* (*closure*)

Замыкания

Задача

Дана функция `make` с типом $a \rightarrow b \rightarrow c$, и значение с типом a . Хочется, для фиксированного значения типа a , отдать наружу функцию $b \rightarrow c$, которая будет строить значения типа c из значений типа b .

```
let make : a -> b -> c = fun a b -> ...  
let make_of_a : a -> c =  
  let arg : b = ... in  
  fun a -> make a b
```

Из функций `make_of_a` вернули частично примененную функцию, а аргумент хранится в *замыкании* (*closure*)

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

Типы и статически типизированные языки

- Зачем нужны типы?
- Все ли значения имеют какой-то тип?
- Если компилятор говорит, что значение имеет такой-то тип, когда можно (или нельзя) ему доверять?

Типы и статически типизированные языки

- Если компилятор говорит, что значение имеет такой-то тип, когда можно (или нельзя) ему доверять?

```
// java
class Vehicle {}
class Car extends Vehicle {}
class Bus extends Vehicle {}

public static void main(String[] args) {
    Car[] c = { new Car() };
    Vehicle[] v = c;
    v[0] = new Bus(); // crashes with ArrayStoreException
}
```

<https://counterexamples.org/general-covariance.html>

(проверено: 1 сентября 2024 г.)

Типы и статически типизированные языки

- Если компилятор говорит, что значение имеет такой-то тип, когда можно (или нельзя) ему доверять?

```
// kotlin
class Foo {
    val nonNull: String
    fun crash() =
        nonNull.startsWith("foo") // crashes
    init {
        this.crash()
        nonNull = "Initialized"
    }
}
```

<https://counterexamples.org/under-construction.html>

(проверено: 1 сентября 2024 г.)

Типы и статически типизированные языки

- Если компилятор говорит, что значение имеет такой-то тип, когда можно (или нельзя) ему доверять?

“Well typed program can’t go wrong” [2]



Робин Милнер
(1934–2010)
Премия Тьюринга
1991

АВТОМАТИЧЕСКИЙ ВЫВОД ТИПОВ

Python

```
def sum(a, b):  
    return (a//b, a % b)  
  
a = 10  
b = 3  
print(f'Sum of {a} and {b} is {sum(a,b)}')
```

Язык со строгой статической типизацией синтаксически не особо длиннее

(* OCaml *)

```
let sum a b = (a/b, a mod b)  
let a = 10  
let b = 3  
let () = Printf.printf "Sum of %d and %d is %d" (sum a b)
```

Алгебраические типы данных

Алгебраические типы данных — это результат скрещивания enum'ов и структур из Си.

Пример: связный список значений типа 'a

Связный список — это структура данных, которая представляет собой или пустой список (nil), или элемент списка (cons), который состоит из элемента типа 'a, хранящегося в списке, и ещё одного списка (хвоста).

```
type 'a list =  
| Nil  
| Cons of 'a * 'a list
```

(Полиморфный) тип 'a list списков значений типа 'a

Nil и Cons – *конструкторы* типа 'a list

У конструктора Nil нет аргументов.

У конструктора Cons два аргумента с типами 'a и 'a list.

Пример: (односвязные) списки и синтаксический сахар для них

```
type 'a list =  
| Nil  
| Cons of 'a * 'a list
```

Вот так будете писать большинство своих типов данных

```
Nil : 'a list  
Cons (1, Nil) : int list  
  
Cons (2, Cons (1, Nil)) : int list  
  
N.B. Типы писать не нужно, они при-  
писаны в дидактических целях.
```

```
type 'a list =  
| []  
| (::) of 'a * 'a list
```

Здесь специально выбранный конструктор, чтобы он был инфиксным и право ассоциативным

```
[] : 'a list  
1::[] : int list  
[1] : int list  
1::2::[] : int list  
[1;2] : int list
```


Пример: тип option

```
type 'a option =  
  | None  
  | Some of 'a
```

Либо нет значения (**None**), либо какое-то есть (**Some**)

Аналог nullptr, только его нельзя случайно разименовывать^a

^aСм. Tony Hoare «Null References: The Billion Dollar Mistake»

```
let run_on_two x y f =  
  match x with  
  | None -> ()  
  | Some x ->  
    (match y with  
     | None -> ()  
     | Some y -> f x y)
```

```
(* somewhere in .mli file *)  
val run_on_two:  
  'a option ->  
  'b option ->  
  ('a -> 'b -> unit) ->  
  unit
```

Пример: тип bool и “Boolean blindness” [3]

```
(* filename.ml *)
(* from OCaml stdlib *)
true : bool

(* A custom one *)
type boolean = True | False

(* be used later *)
match ... with
| true  -> ...
| false -> ...

(* But if-then-else is
   better *)
if ... then ... else ...
```

```
(* filename.mli *)
val is_admin : user -> bool
val is_red   : node -> true
val list_filter: ('a -> bool) ->
                  'a list -> 'a list

(* Below is better *)
type role = Admin | User
val get_role : user -> role
type color = Red | Black
val get_color : node -> color

type filter = Keep | Remove
val list_filter: ('a -> filter) ->
                  'a list -> 'a list
```

Почему алгебраические типы данных (язык Норе, 1980)
называются «алгебраическими»?

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection):
существуют такие s_i^k , что
 $s_i^k(C_k(x_{k_1}, \dots, x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness):
если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity):
если
 $C_{ij}(x_1, \dots, x_{n_{ij}}) = C_{ij}(y_1, \dots, y_{n_{ij}})$, то
 $x_k = y_k$
- Полнота (exhaustiveness):
если x алгебраического типа, то
 $\exists i, n: x = C_i(y_1, \dots, y_n)$

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection):
существуют такие s_i^k , что
 $s_i^k(C_k(x_{k_1}, \dots, x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness):
если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity):
если
 $C_{ij}(x_1, \dots, x_{n_{ij}}) = C_{ij}(y_1, \dots, y_{n_{ij}})$, то
 $x_k = y_k$
- Полнота (exhaustiveness):
если x алгебраического типа, то
 $\exists i, n: x = C_i(y_1, \dots, y_n)$

Доступ к аргументам можно осуществлять с помощью паттерн-мэтчинга

```
let rec somefunc xs =  
  match xs with  
  | [] ->  
    (* no argument *)  
    ...  
  | h :: tl ->  
    ... h ... tl ...  
    h ...
```

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection):
существуют такие s_i^k , что
 $s_i^k(C_k(x_{k_1}, \dots, x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness):
если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity):
если
 $C_{ij}(x_1, \dots, x_{n_{ij}}) = C_{ij}(y_1, \dots, y_{n_{ij}})$, то
 $x_k = y_k$
- Полнота (exhaustiveness):
если x алгебраического типа, то
 $\exists i, n: x = C_i(y_1, \dots, y_n)$

Дизъюнктивность: если значения начинаются с разных конструкторов, то они не равны

```
let rec neq_lists xs ys =  
  match (xs, ys) with  
  | [], _::_ -> true  
  | _::_, [] -> true  
  | x::tl1, y::tl2 ->  
    (x <> y) ||  
    neq_lists tl1 tl2  
  | [], [] -> false
```

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection):
существуют такие s_i^k , что
 $s_i^k(C_k(x_{k_1}, \dots, x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness):
если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity):
если
 $C_{ij}(x_1, \dots, x_{n_{ij}}) = C_{ij}(y_1, \dots, y_{n_{ij}})$, то
 $x_k = y_k$
- Полнота (exhaustiveness):
если x алгебраического типа, то
 $\exists i, n: x = C_i(y_1, \dots, y_n)$

Инъективность: равные значения начинаются с одного и того же конструктора, и аргументы равны попарно.

```
let rec eq_lists xs ys =  
  match xs, ys with  
  | [], [] -> true  
  | [], _ :: _  
  | _ :: _, [] -> false  
  | x::tl1, y :: tl2 ->  
    (x = y) &&  
    eq_lists tl1 tl2
```

Формальные аксиомы алгебраических типов

- Доступ к аргументам (selection):
существуют такие s_i^k , что
 $s_i^k(C_k(x_{k_1}, \dots, x_{k_n})) = x_{k_i}$
- Дизъюнктность (distinctness):
если $j \neq i$, то $C_j(x) \neq C_i(y)$
- Инъективность (injectivity):
если
 $C_{ij}(x_1, \dots, x_{n_{ij}}) = C_{ij}(y_1, \dots, y_{n_{ij}})$, то
 $x_k = y_k$
- Полнота (exhaustiveness):
если x алгебраического типа, то
 $\exists i, n: x = C_i(y_1, \dots, y_n)$

Полнота: значения алгебраического типа всегда начинаются только с тех конструкторов, которые перечислены в типе

```
let rec even_length xs =  
  match xs with  
  | [] -> true  
  | _ :: _ :: tl ->  
    even_length tl
```

(* Warning 8: this pattern-matching
is not exhaustive.

Here is an example of a case
that is not matched:

```
_ :: []  
*)
```


Использование вложенных функций

```
let sum_list: int list -> int =  
  let rec helper acc xs =  
    match xs with  
    | [] -> acc  
    | x::xs -> helper (acc+x) xs  
  in  
  helper 0
```

Здесь в теле функции `sum_list` объявлена функция `helper` с дополнительным параметром-аккумулятором.

Компилятор достаточно умный, чтобы не создавать новую функцию `helper` на каждый вызов `sum_list`.

Использование вложенных функций

```
let sum_list: int list -> int =  
  let rec helper acc xs =  
    match xs with  
    | [] -> acc  
    | x::xs -> helper (acc+x) xs  
  in  
  helper 0
```

Здесь в теле функции `sum_list` объявлена функция `helper` с дополнительным параметром-аккумулятором.

```
let sum_list (zs: int list) : int =  
  let rec helper acc xs =  
    match xs with  
    | [] -> acc  
    | x::xs -> helper (acc+x) xs  
  in  
  helper 0 zs
```

Компилятор достаточно умный, чтобы не создавать новую функцию `helper` на каждый вызов `sum_list`.

Определение (η-конверсия)

Правило преобразования λ-выражений и выражений в ФП.

Утверждает, что $\forall f : (\lambda x. fx) \simeq f$

Когда переписывание идет справа налево, бывает говорят η-экспансия (англ. η-экспансия)

Для чистых функциональных языков переписывание корректно. Если есть присваивания — не всегда из-за соображений типизации.

```
sum_list xs = helper 0 xs
           ⇕
sum_list = helper 0
```

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

Рекурсия предпочтительна, циклов и присваиваний не пишут

```
(* make : 'a -> int -> 'a list *)  
let rec make x n =  
  if n < 1 then []  
  else x :: (make x (n-1))
```

Обычная рекурсивная функция: строим список от головы к хвосту

Рекурсия предпочтительна, циклов и присваиваний не пишут

```
(* make : 'a -> int -> 'a list *)  
let rec make x n =  
  if n < 1 then []  
  else x :: (make x (n-1))
```

```
(* make2 : 'a -> int -> 'a list *)  
let make2 x n =  
  let rec helper acc n =  
    if n < 1 then acc  
    else helper (x :: acc) (n-1)  
  in  
  helper [] n
```

Обычная рекурсивная функция: строим список от головы к хвосту

Хвостовая рекурсия получается, если результат функции — это

- либо значение без рекурсивного вызова
- либо вызов функции (возможно той же самой) с какими-то аргументами

Хвостовые вызовы использует константное количество стека, и поэтому примерно эквивалентны циклу

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. Вывод типов

Функция List.map

```
(* OCaml *)  
let rec map f ys =  
  match ys with  
  | [] -> []  
  | x::xs -> f x :: (map f xs)
```

```
let rez = map func xs
```

```
(* In REPL (exec 'ocaml') *)  
# map ((+)1) [1; 2; 3; 4];;  
- : int list = [2; 3; 4; 5]
```

```
// kotlin  
val ys = listOf(...);  
ys.map( ::func )
```


Функция List.fold_left

```
(* OCaml *)  
let rec fold_left f acc ys =  
  match ys with  
  | [] -> acc  
  | x::xs -> fold_left f (f acc x) xs
```

```
# fold_left (+) 0 [1;2;3;4];;  
- : int = 10  
# fold_left (^) "0" ["1";"2";"3"];;  
- : string = "0123"  
  
# fold_left (^) "0" ["1";"2";"3"];;  
- : string = "0123"
```

```
// kotlin  
var numbers = listOf(1,2,3,4);  
numbers.fold(0) { a, b -> a+b }
```

1. Определения
2. Синтаксис вызова функций
3. Кортежи
4. Области видимости
5. Замыкания
6. Алгебраические типы данных
7. Хвостовые вызовы функций (хвостовая рекурсия)
8. Стандартные функции для списков
9. **Вывод типов**

Автоматический вывод типов. Пример 1

```
let s f g x = f x (g x)
```

Вывод:

- $f : 'a \rightarrow 'b \rightarrow 'c$
- $g : 'd \rightarrow 'e$
- $s : ('a \rightarrow 'b \rightarrow 'c) \rightarrow ('d \rightarrow 'e) \rightarrow 'x \rightarrow 'r$
- $'c \sim 'r$ т.к. результат f — это результат s
- $'x \sim 'a$ т.к. 1й аргумент f — это 3й аргумент s
- $'x \sim 'd$ аналогично для g
- $'e \sim 'b$ т.к. результат g — это 2й аргумент s
- Итого:
 $s : ('a \rightarrow 'b \rightarrow 'c) \rightarrow ('a \rightarrow 'b) \rightarrow 'a \rightarrow 'c$

Автоматический вывод типов. Пример 2

Трассировка вывода типа:

- $\text{map} : 'f \rightarrow 'ys \rightarrow 'res$
- $ys : 'y \text{ list}$ и $'y \text{ list} \simeq 'ys$
- $rez : 'r \text{ list}$
- $'f \simeq ('a \rightarrow 'b)$ т.к. f применяется к какому-то аргументу как функция
- $x : 'y$ и $xs : 'y \text{ list}$
- $'y \simeq 'a$ т.к. f применяется к x
- $'r \simeq 'b$ т.к. результат $f \ x$ складывается в список, который хранит значения типа $'r$
- Итого:
 $\text{map} : ('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list}$

```
(* OCaml *)  
let rec map f ys =  
  match ys with  
  | [] -> []  
  | x::xs -> f x :: (map f xs)
```

Автоматический вывод типов. Пример 3. (1/2)

Что может пойти не так?

```
let rec map: ('a -> 'b) -> 'a list -> 'b list = fun f -> function
  | [] -> []
  | x::xs -> x :: f x :: map f xs (* Bug *)
```

Автоматический вывод типов. Пример 3. (1/2)

Что может пойти не так?

```
let rec map: ('a -> 'b) -> 'a list -> 'b list = fun f -> function
  | [] -> []
  | x::xs -> x :: f x :: map f xs (* Bug *)
```

Несмотря на то, что приписали тип, вывелось не то, что написали:

```
let rec map (* : ('a -> 'a) -> 'a list -> 'a list *)
= fun f -> function
  | [] -> []
  | x::xs -> x :: f x :: map f xs
```

Автоматический вывод типов. Пример 3. (1/2)

Что может пойти не так?

```
let rec map: ('a -> 'b) -> 'a list -> 'b list = fun f -> function
  | [] -> []
  | x::xs -> x :: f x :: map f xs (* Bug *)
```

Несмотря на то, что приписали тип, вывелось не то, что написали:

```
let rec map (* : ('a -> 'a) -> 'a list -> 'a list *)
  = fun f -> function
    | [] -> []
    | x::xs -> x :: f x :: map f xs
```

Лайфхак 1:

```
let rec map: 'a 'b . ('a -> 'b) -> 'a list -> 'b list = fun f -> function
  | [] -> []
  | x::xs -> x :: f x :: map f xs
(* Error: This definition has type 'c. ('c -> 'c) -> 'c list -> 'c list
   which is less general than 'a 'b. ('a -> 'b) -> 'a list -> 'b list
```

Автоматический вывод типов. Пример 3. (2/2)

Лайфхак 2:

```
(* filename.ml *)  
let rec map = fun f ys ->  
  match ys with  
  | [] -> []  
  | x::xs -> x :: (f x) :: (map f xs)  
  
(* filename.mli *)  
val map: ('a -> 'b) -> 'a list -> 'b list
```

Файлы интерфейса (signature)

- Объявления значений с типами
- Объявления модулей и типов модулей
- Документация

Файлы реализации (structure)

- Реализация значений
- Реализация модулей
- Сигнатуры типов модулей
- и т.д.

Каким должен быть вывод типов?

- Он должен завершаться (что на самом деле не так просто [4])
- Должен работать быстро
 - От компилятора в целом это тоже требуется

Каким должен быть вывод типов?

- Он должен завершаться (что на самом деле не так просто [4])
- Должен работать быстро
 - От компилятора в целом это тоже требуется
- У выражения не должно быть можно вывести два типа непохожих друг на друга
 - Другими словами: если можно выражению приписать два типа t_1 и t_2 , то должен существовать третий тип t , такой что первые два являются его специализацией

$(\text{int} \rightarrow \text{string}) \rightarrow \text{int list} \rightarrow \text{string list}$	$(* t_1 *)$
$(\text{bool} \rightarrow \text{int}) \rightarrow \text{bool list} \rightarrow \text{int list}$	$(* t_2 *)$
$(\text{'a'} \rightarrow \text{'b'}) \rightarrow \text{'a list} \rightarrow \text{'b list}$	$(* t *)$

- И это требование **сложно** исполнить!

Новые понятия:

- 1 каррированные функции
- 2 вложенные функции
- 3 сопоставление с образцом и wildcard
- 4 хвостовая рекурсия
- 5 автоматический вывод типов
- 6 разделение на файлы интерфейса и реализации

- [1] *Why is catching an exception non-pure, but throwing an exception is pure?* url: <https://stackoverflow.com/a/12345665/106543>.
- [2] Robin Milner. «A theory of type polymorphism in programming». B: (1978). doi: [https://doi.org/10.1016/0022-0000\(78\)90014-4](https://doi.org/10.1016/0022-0000(78)90014-4).
- [3] Einar Landre. *Prefer Domain-Specific Types to Primitive Types*. 2020. url: https://github.com/97-things/97-things-every-programmer-should-know/blob/master/en/thing_65/README.md.
- [4] Radu Grigore. *Java Generics are Turing Complete*. 2016. arXiv: 1605.05274 [cs.PL]. url: <https://arxiv.org/abs/1605.05274>.