

Funkcionális minták

Knuth – kihívás I.

- ▶ szavak olvasása fájlból
- ▶ cél az n leggyakoribb szó, illetve azok gyakoriságának a kiírása

```
tr -cs A-Za-z '\n' | tr A-Z a-z | sort | uniq -c | sort -rn | sed ${1} q
```

Knuth – kihívás II.

```
(print_pairs ◦ sort_by_freq ◦ reverse_pairs ◦ count_occurences ◦ words)(text)
```

Eredmény kiírása.

Előfordulás szerinti
sorrendezés.

Szó - előfordulás párok
"megfordítása".

Előfordulások
számolása.

Szavak összegyűjtése.

Magasabb rendű függvények

- ▶ függvény, amelynek függvény a paramétere
 - ▷ `std::bind (+std::placeholders::_1, ...)`
 - ▷ függvény argumentumainak részleges lekötése
 - ▷ az argumentumok számát ismerni kell
 - ▷ lambda kifejezések
- ▶ körrizés
 - ▷ minden függvény egy paramétert vár
 - ▷ argumentumok számának ismerete nélkül is alkalmazható
- ▶ `std::invoke`
 - ▷ C++17, `<functional>`
 - ▷ feature test macro: `__cpp_lib_invoke`
 - ▷ `std::invoke("meghívható függvényszerűség", paraméterek...)`

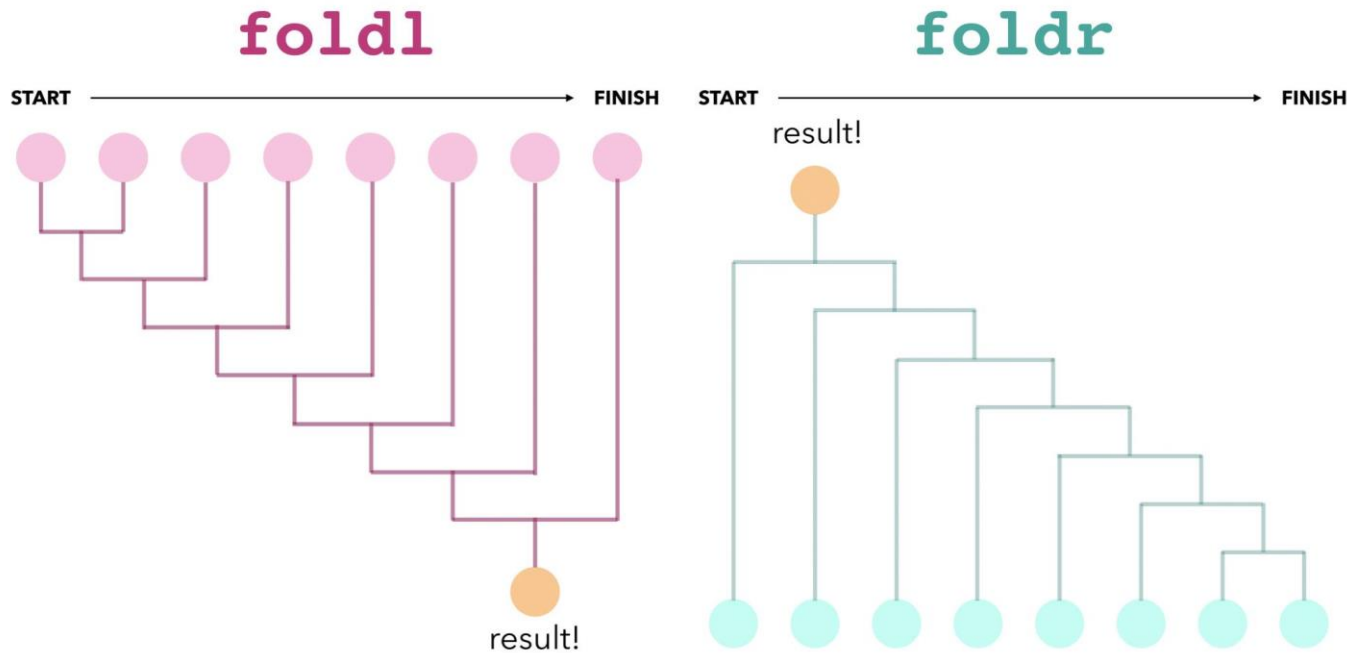
Funkcionális minták

Folding
Farokrekurzió
Funktorok
Applikatív funktor
Monádok

Folding I.

- ▶ fold
 - ▷ reduce, accumulate, aggregate, compress, inject
 - ▷ rekurzív adatstruktúrák bejárására és elemeinek kombinálása megadott művelet segítségével
- ▶ `std::accumulate`, `std::reduce`
- ▶ C++17 fold expression
 - ▷ $I \text{ op } \dots \text{ op } E = (((I \text{ op } E_1) \text{ op } E_2) \text{ op } \dots) \text{ op } E_N$
 - ▷ $E \text{ op } \dots \text{ op } I = (E_1 \text{ op } (\dots \text{ op } E_{N-1} \text{ op } (E_N \text{ op } I))))$
 - ▷ op
 - ▷ `+ - * / % ^ & | = < > << >> += -= *= /= %= ^= &= |= <<= >>= == != <= >= && || , . * -> *`


Folding II.



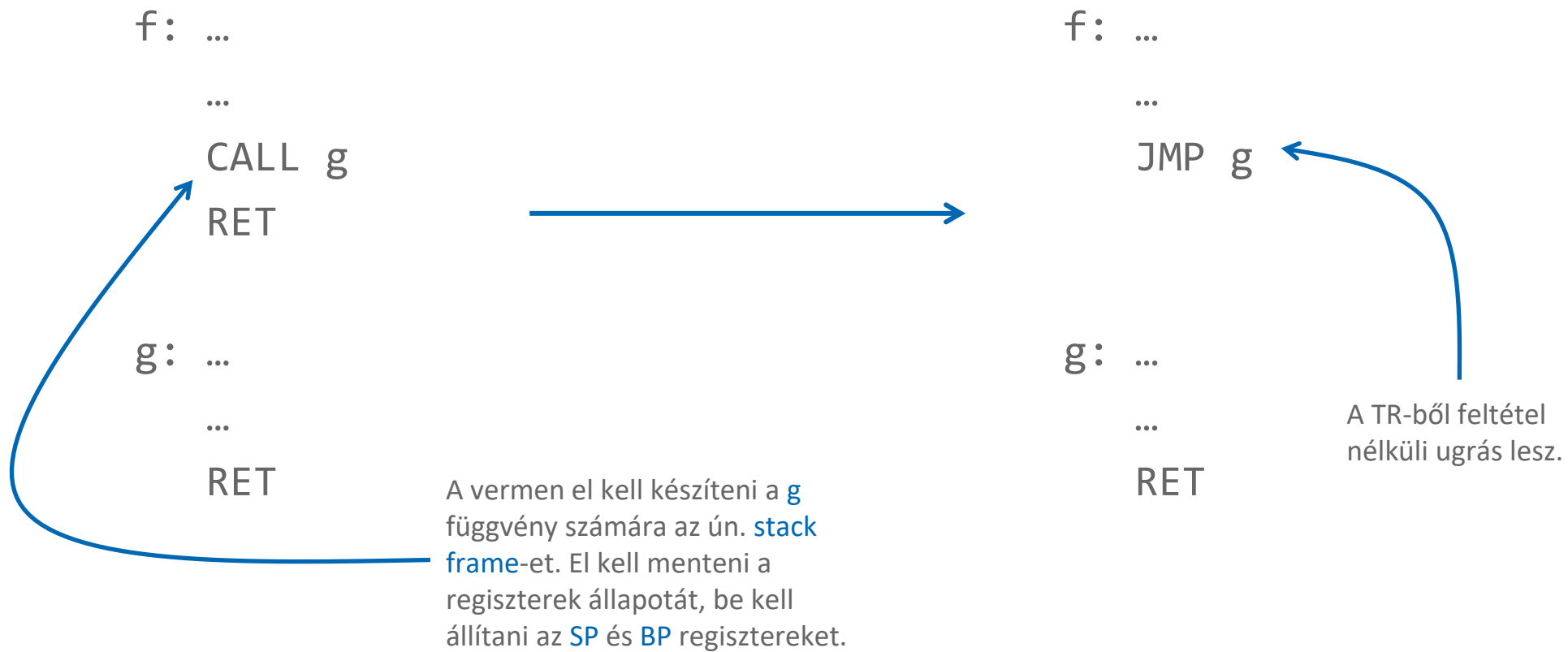
```
template<typename Cont, typename N, typename F>
auto foldl(const Cont &c, N n, F &&f) {
    return std::accumulate(std::cbegin(c), std::cend(c), std::move(n), std::forward<F>(f));
}

template<typename Cont, typename N, typename F>
auto foldr(const Cont &c, N n, F &&f) {
    return std::accumulate(std::crbegin(c), std::crend(c), std::move(n), std::forward<F>(f));
}
```

Tail Recursion / Tail Recursion Elimination

- ▶ függvényhívás
 - ▶ stack frame
 - ▶ idő- és tárigény
- ▶ TRE
 - ▶ optimalizáció
 - ▶ gyakorlatilag megegyezik egy ciklussal
 - ▶ speciális rekurzió
 - ▶ a függvény önmagát hívja meg, az eredményt nem dolgozza fel tovább
- ▶ rekurzió a funkcionális programok fontos része
 - ▶ TRE nélkül nem lenne hatékony


TRE – Assembly



ex_1: TR faktoriális

```
// not really tail recursive!  
int factorial(int n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * factorial(n-1);  
}
```

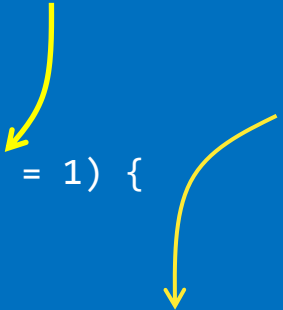
Módosítja a
rekurzív hívás
értékét...



```
// tail recursive version  
int factorial(int n, int acum = 1) {  
    if (n == 0)  
        return acum;  
    else  
        return factorial(n - 1, acum * n);  
}
```

Segédváltozó a rész-
eredmény tárolására.

Rekurzió eredményének
direkt visszaadása.



Felemelés / Lifting

- ▶ függvények transzformálása, hogy azok szélesebb körben használhatóak legyenek

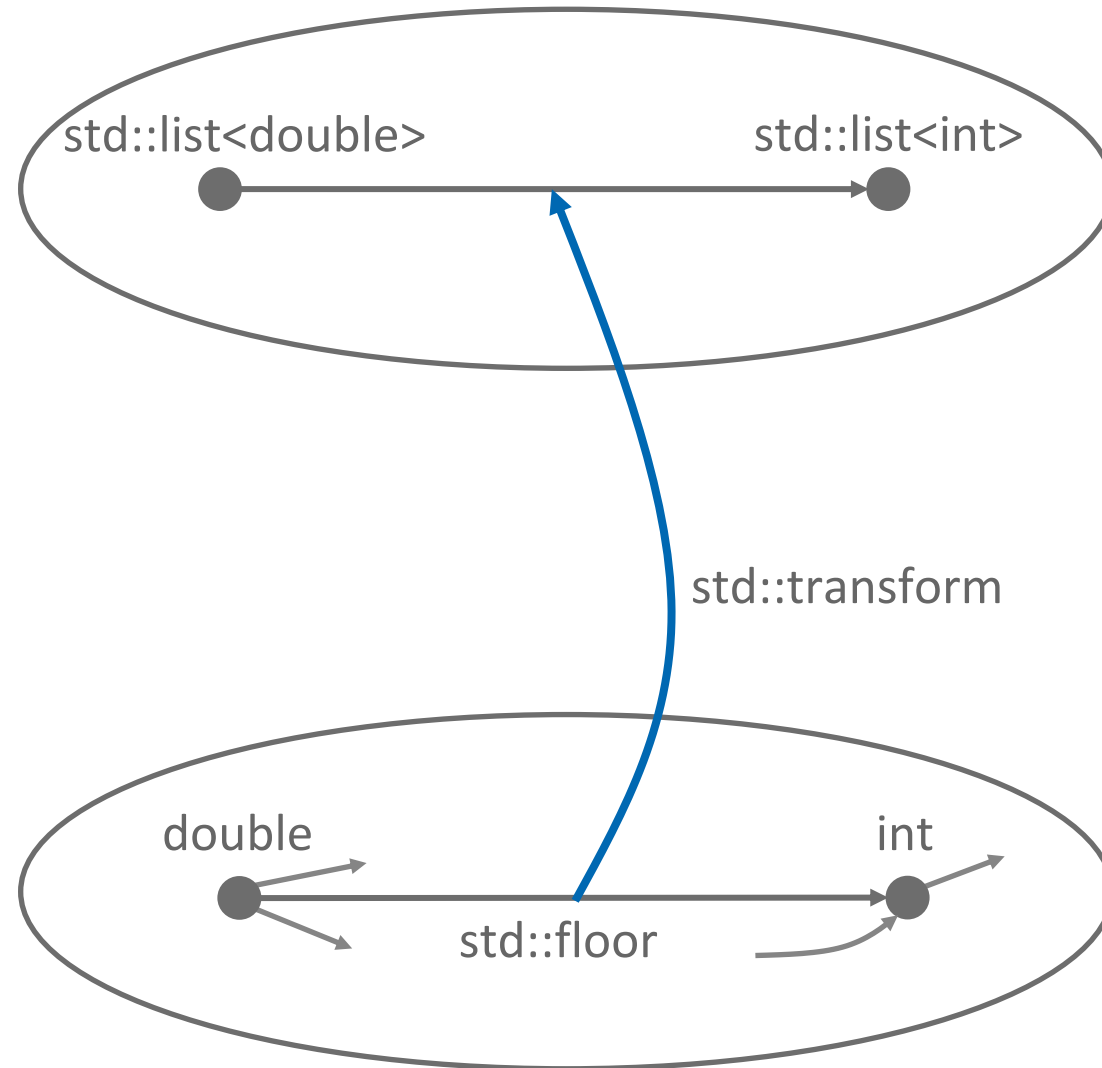
```
void to_upper(std::string&);
```



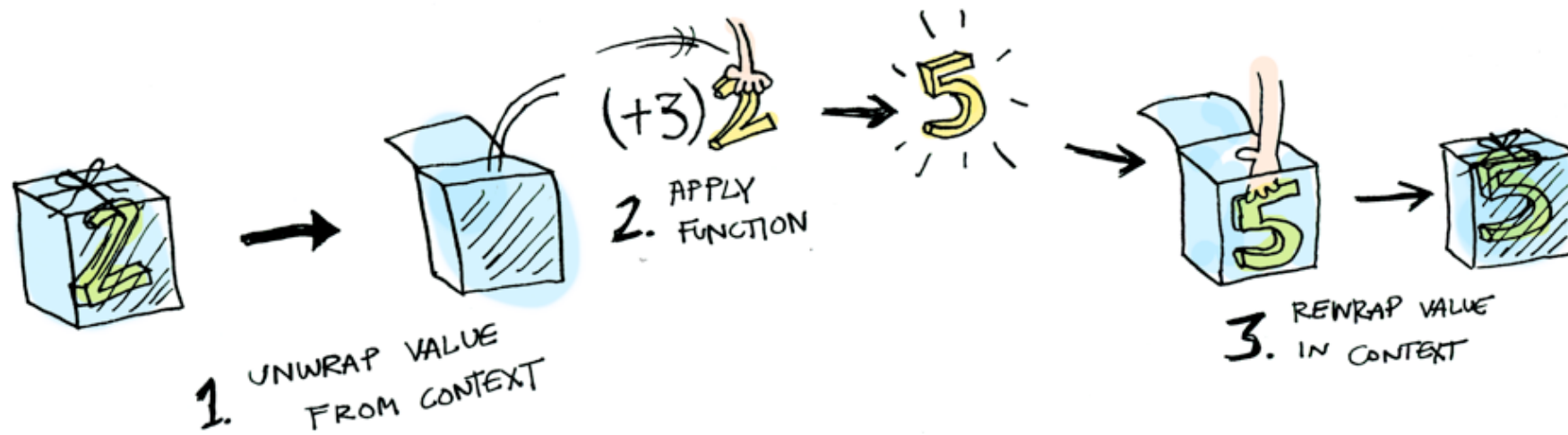
```
void map_to_upper(std::vector<std::string> &strs) {  
    for(auto &str :strs) {  
        to_upper(str);  
    }  
}
```

Felemelés / Funktor

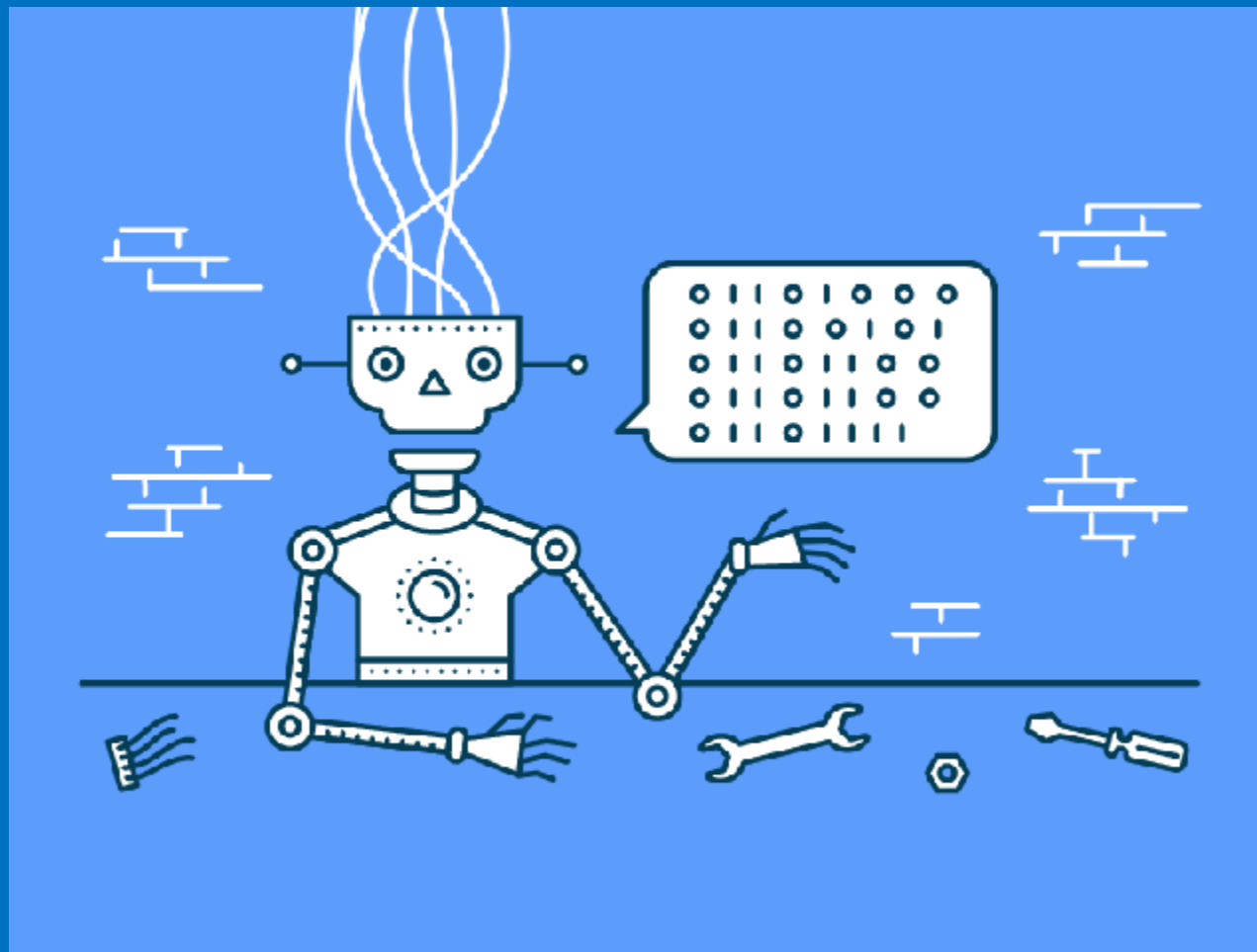
- `std::transform`



Funkcionális minták – Funktor



ex_2: fmap



ex_2: Funktorok, fmap

```
template<typename F>
auto fmap(F f) {
    return [f = std::move(f)]<template<typename...> class V, typename ...T>(const V<T...> &x) {
        using Ti = typename std::decay_t<decltype(x)>::value_type;
        using Oi = decltype(f(std::declval<Ti>()));
        V<Oi> y;
        std::transform(std::cbegin(x), std::cend(x), std::back_inserter(y), f);
        return y;
    };
}
```

C++20. Írjuk át függvény-objektummá..!

V-nek vannak sablonparaméterei.

És ha más allokátort szeretnék?

f-et alkalmazva x elemeire mit kapunk?

Miket is tárolunk x-ben?

```
std::list<double> l{3.14, 3.5, 4.5};
auto floor = [](double d){ return std::floor(d); };
```

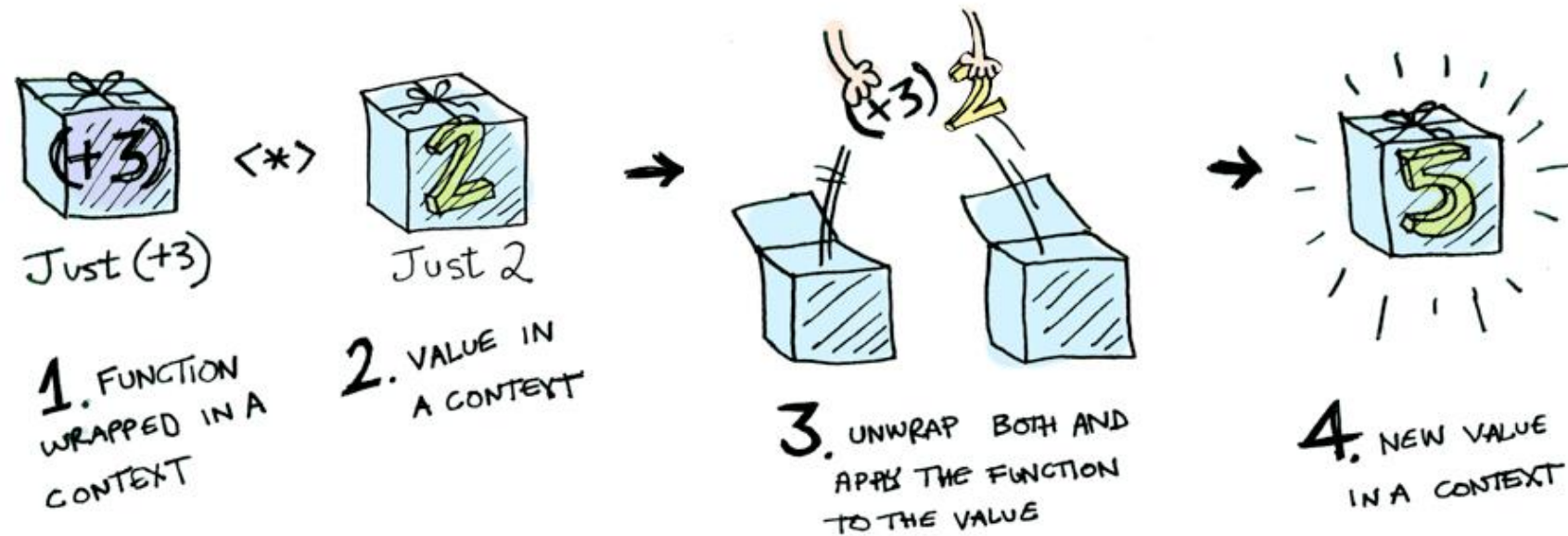
```
for (const auto &e : fmap(floor)(l)) {
    std::cout << e << " ";
}
```

std::floor nem csak double-val hívható meg...

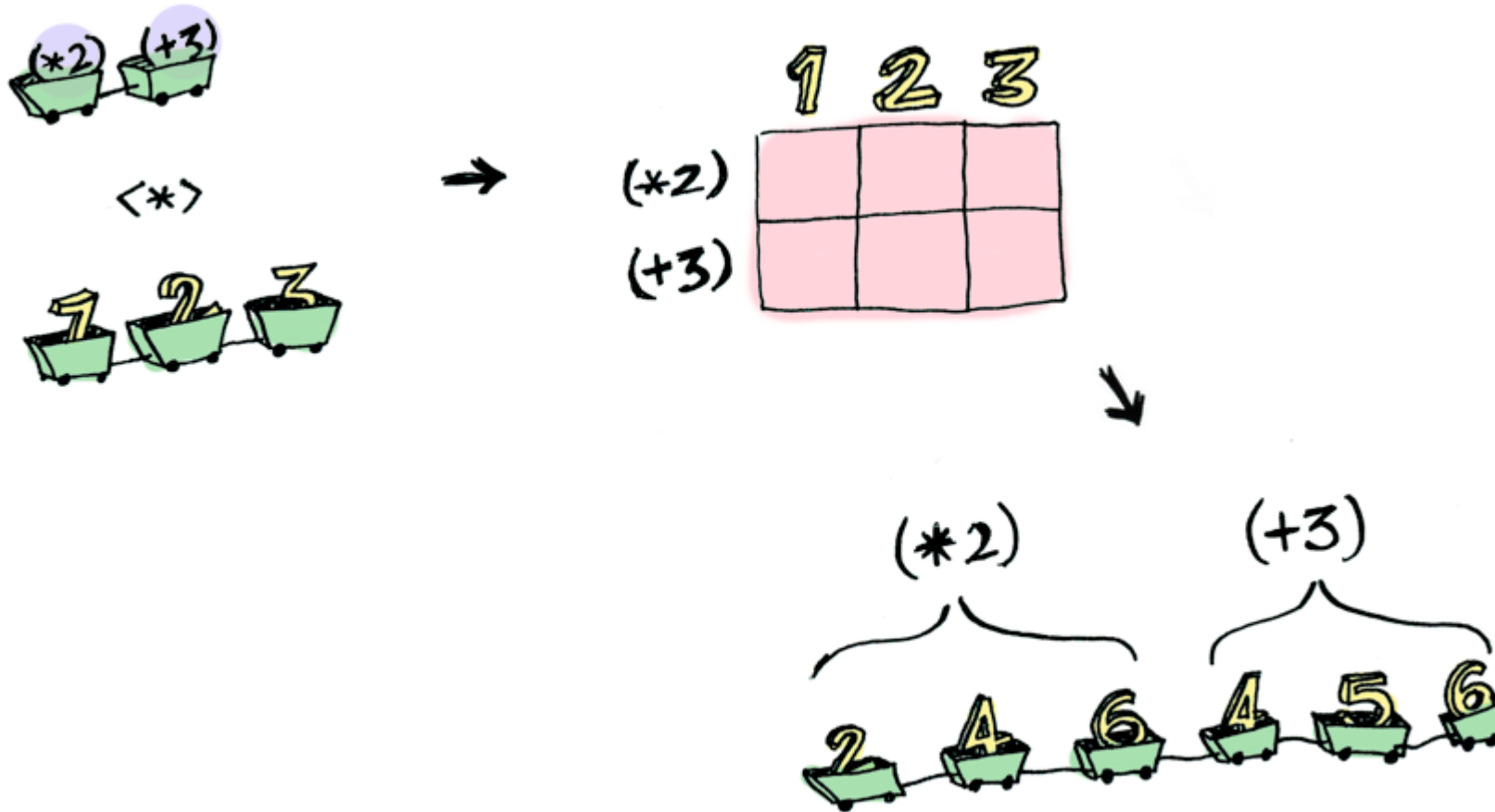


```
#include <utility>
#include <algorithm>
#include <type_traits>
```

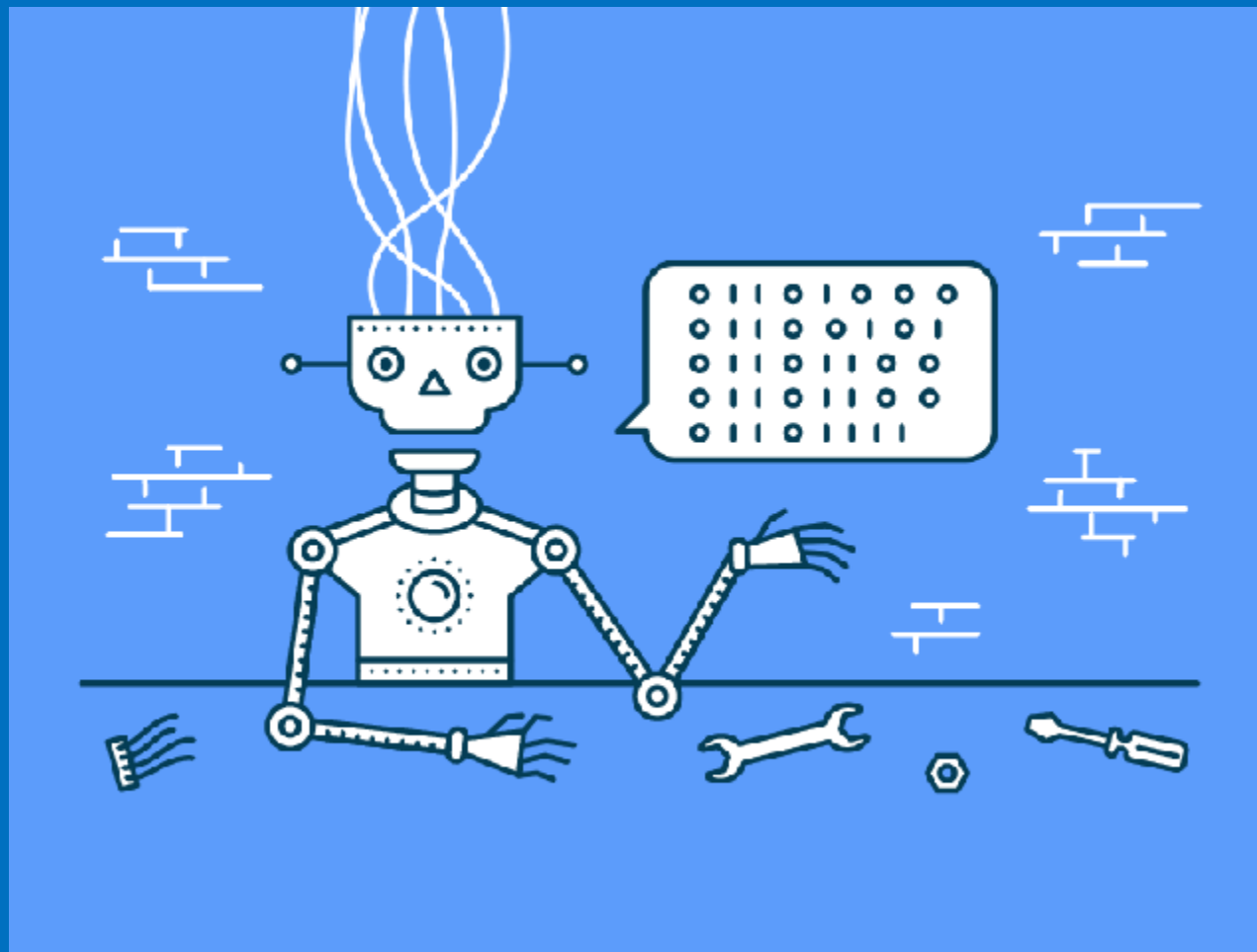
Funkcionális minták – Applikatív funktor I.



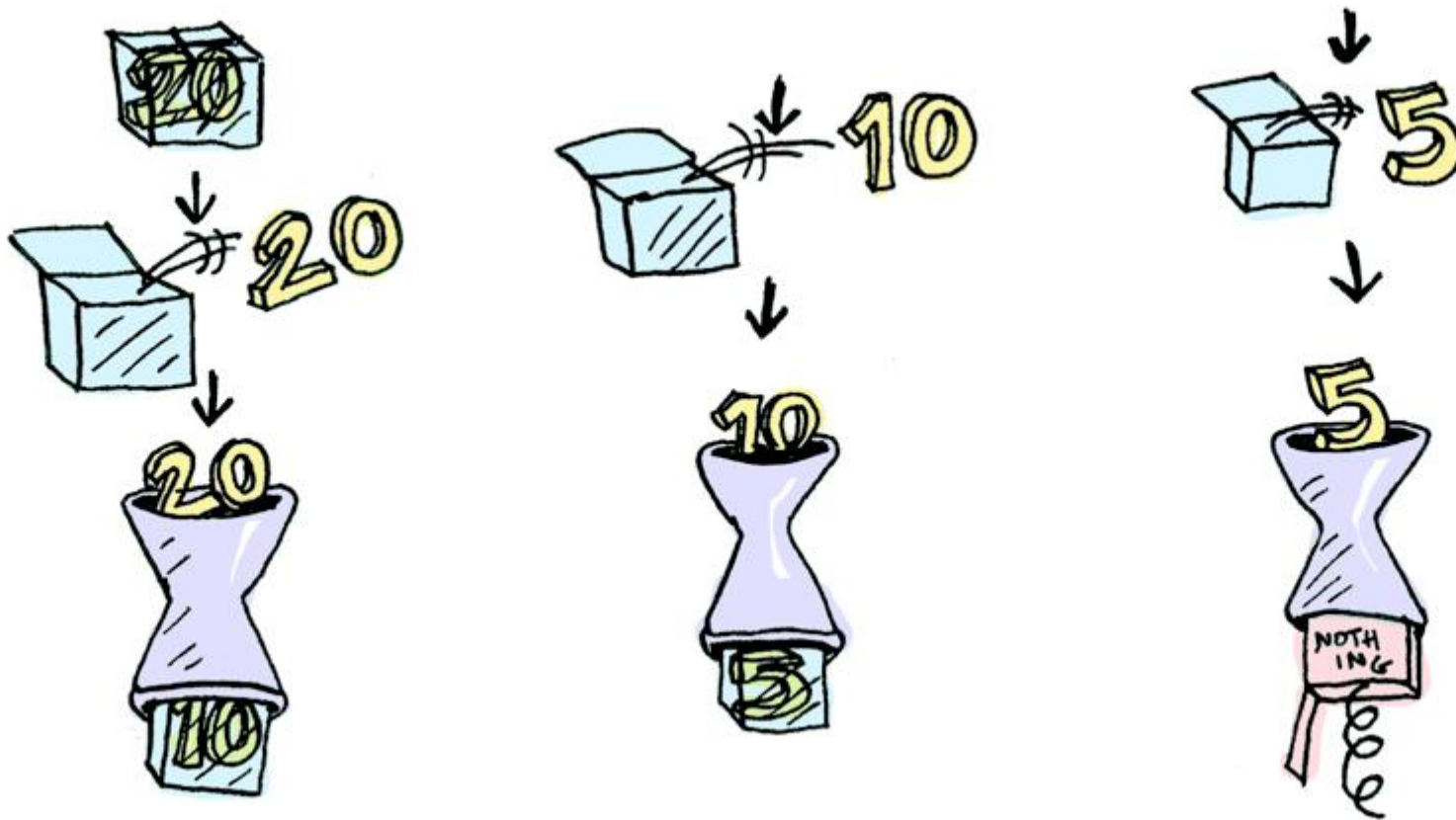
Funkcionális minták – Applikatív funktor II.



ex_3: zapper



Funkcionális minták – Monád



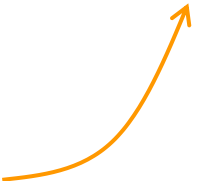
Kleisli – kategória I.

```
std::string log;
```

```
bool negate(bool x) {  
    log += "Negate";  
    return !x;  
}
```

```
std::pair<bool, std::string>  
negate(bool x, const std::string log) {  
    return std::make_pair(!x, log + "Negate");  
}
```

De a negate függvénynek miért kell tudnia, hogy miképp kell a naplózási adatokat összegyűjteni?



Kleisli – kategória II.

```
std::pair<bool, std::string>  
negate(bool x) {  
    return std::make_pair(!x, "Negate");  
}
```

Komponálás...



```
template<typename A, typename B, typename C>  
std::function<std::pair<C, std::string>(A)>  
compose(std::function<std::pair<B, std::string>(A)> f, std::function<std::pair<C, std::string>(B)> g){  
    return [f, g](A x) {  
        const auto p1 = f(x);  
        const auto p2 = g(p1.first);  
        return std::make_pair(p2, p1.second + p2.second);  
    };  
}
```

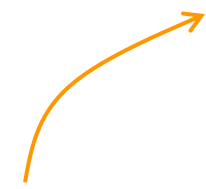
Kategória?



Kleisli – kategória III.

id:


```
template<typename A>
std::pair<A, std::string> id(A x) {
    return std::make_pair(x, std::string{});
}
```

An orange arrow points from the word 'Kleisli-kompozíció:' to the 'id' function definition.

Kategória.

Kleisli-kompozíció:

```
template<typename F, typename G> auto compose(F f, G g) {
    return [f, g](A x) {
        const auto p1 = f(x);
        const auto p2 = g(p1.first);
        return std::make_pair(p2, p1.second + p2.second);
    };
}
```

An orange arrow points from the word 'Kleisli-kompozíció:' to the 'compose' function definition.

Monoid.

An orange arrow points from the word 'Monoid.' to the 'std::string{}' in the 'id' function definition.An orange arrow points from the word 'Monoid.' to the '+' operator in the 'compose' function definition.

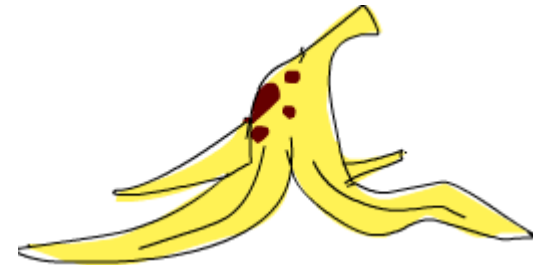
Kleisli – kategória / Monád

- ▶ két művelet: kötés (bind), és a return
 - ▶ a return vesz egy egyszerű típusértéket és monadikus értéket hoz létre
 - ▶ a kötés vesz egy monadikus értéket és egy függvényt
 - ▶ kivonja a beburkolt értéket, és átadja a függvénynek
 - ▶ a függvény által létrehozott értéket monadikussá teszi
- ▶ M monád, ha
 - ▶ $\text{fmap} :: (a \rightarrow b) \rightarrow (M\ a \rightarrow M\ b)$
 - ▶ $\text{join} :: M\ (M\ a) \rightarrow M\ a$
 - ▶ $\text{return} :: a \rightarrow M\ a$
- ▶ fmap
 - ▶ kap egy függvényt, és visszaad egy függvényt, ami ugyanazt csinálja, mint a paraméter, csak monadikus értékekkel
- ▶ join
 - ▶ két réteg monadikus információt egyrétegűvé lapít

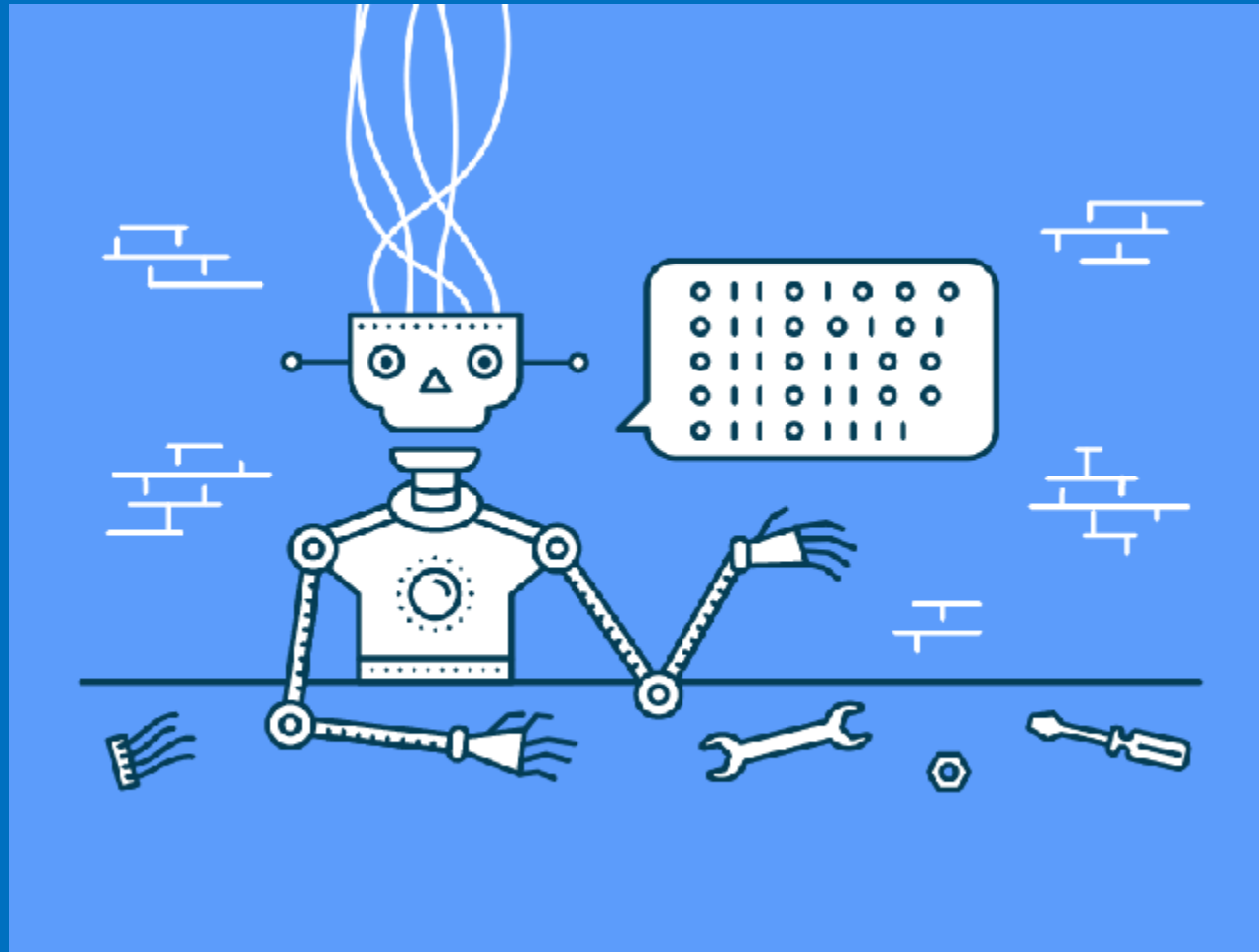
Pierre, a kötéláncos



- ▶ Pierre szeret magasan a házak között egy kötélen egyensúlyozni. Egyetlen problémája a galambok, amit rászálnak a egyensúlyozáshoz használt rúdra.
- ▶ Ha az egyik oldalon hárommal több galamb van, akkor Pierre egyensúlyát veszti.



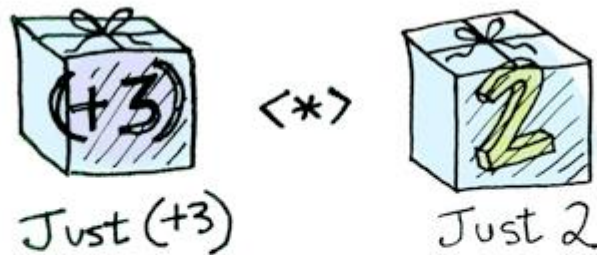
ex_4: Pierre, a kötéláncos



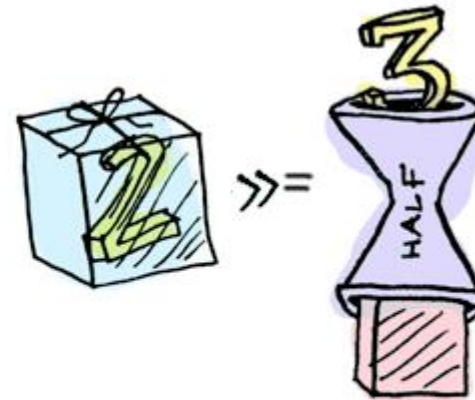
Funkcionális minták – összehasonlítás



Functor



Applicative



Monad

Köszönöm a figyelmet!

Folytatjuk...