Kwadrans o Monadach

Panicz Maciej Godek

godek.maciek@gmail.com

https://github.com/panicz/writings/tree/master/talks/jug/kwadrans

JUG Trójmiasto, 19.10.2023

Agenda

- koncepcja monady
- problemy z Haskellem
- piramida zagłady (z lukrem)
- utyskiwania

Agenda

- koncepcja monady
- problemy z Haskellem
- piramida zagłady (z lukrem)
- utyskiwania

Agenda

- koncepcja monady
- problemy z Haskellem
- piramida zagłady (z lukrem)
- utyskiwania

Agenda

- koncepcja monady
- problemy z Haskellem
- piramida zagłady (z lukrem)
- utyskiwania

Agenda

- koncepcja monady
- problemy z Haskellem
- piramida zagłady (z lukrem)
- utyskiwania

Programowanie bezpunktowe

Odwrotność pierwiastka kwadratowego

```
isqrt x = 1/(sqrt x)
```

bezpunktowo:

```
isqrt = (1/) . sqrt
```

gdzie `(f . g) x = f (g x)`

w JS:

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

Programowanie bezpunktowe

Odwrotność pierwiastka kwadratowego

```
isqrt x = 1/(sqrt x)
```

bezpunktowo:

```
isqrt = (1/) . sqrt
```

gdzie `(f . g) x = f (g x)`

w JS:

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

Programowanie bezpunktowe

Odwrotność pierwiastka kwadratowego

```
isqrt x = 1/(sqrt x)
```

bezpunktowo:

```
isqrt = (1/) . sqrt
```

gdzie `(f . g) x = f (g x)`

w JS:

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

Programowanie bezpunktowe

Odwrotność pierwiastka kwadratowego

```
isqrt x = 1/(sqrt x)
```

bezpunktowo:

```
isqrt = (1/) . sqrt
```

gdzie `(f . g) x = f (g x)`

w JS:

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

Programowanie bezpunktowe

Odwrotność pierwiastka kwadratowego

```
isqrt x = 1/(sqrt x)
```

bezpunktowo:

```
isqrt = (1/) . sqrt
```

gdzie `(f . g) x = f (g x)`

w JS:

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

Programowanie bezpunktowe

Odwrotność pierwiastka kwadratowego

```
isqrt x = 1/(sqrt x)
```

bezpunktowo:

```
isqrt = (1/) . sqrt
```

gdzie `(f . g) x = f (g x)`

w JS:

```
function compose(f, g) {
  return function(x) {
    return f(g(x));
  };
}
```

Operator kompozycji

Typ operatora kompozycji:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|) :: (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Typ operatora kompozycji:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|)  ::  (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Typ operatora kompozycji:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|)  ::  (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Typ operatora kompozycji:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|)  ::  (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Typ operatora kompozycji:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|)  ::  (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Typ operatora kompozycji:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|)  ::  (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Typ operatora kompozycji:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|)  ::  (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Typ operatora kompozycji:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|)  ::  (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Typ operatora kompozycji:

```
(.)  ::  (b -> c) -> (a -> b) -> (a -> c)
```

wygląda trochę dziwacznie, ale jeżeli zdefiniujemy

```
(g | f) x = f (g x)
```

typem “odwróconej kompozycji” jest

```
(|)  ::  (a -> b) -> (b -> c) -> (a -> c)
```

vide potoki w UNIXie

albo “wujek kolegi mojego brata” vs. “mojego brata kolegi wujek”

albo `f(g(x))` vs. `x->getG()->getF()`

czaicie rozumiecie

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
```

```
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
```

```
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
```

```
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
```

```
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
```

```
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
```

```
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
```

```
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
```

```
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
function id(x) { return x; }
```

Operator kompozycji

Własności operatora kompozycji:

- łączny: `f . (g . h) = (f . g) . h`
  tak jak: `x + (y + z) = (x + y) + z`
  albo o: `x * (y * z) = (x * y) * z`
- posiada element neutralny `id`:
  `f . id = id . f = f`
  tak jak: `x + 0 = 0 + x = x`
  albo: `x * 1 = 1 * x = x`

gdzie funkcja tożsamościowa jest zdeiniowana jako

```
id x = x
function id(x) { return x; }
```

Och, ta matematyka

Matematycy nazywają operator łączny z elementem neutralnym *monoidem* (albo *półgrupą z jedynką*).

A teraz wyobraźmy sobie takie uogólnienie operatora kompozycji:

```
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

Na przykład:

```
class WithLog<T> {
  public T value;
  public String log;
}
(f <|_WithLog g) a =
  WithLog b = g(a);
  WithLog c = f(b.value);
  return WithLog(value = c.value, log = b.log
+ c.log);
```

Och, ta matematyka

Matematycy nazywają operator łączny z elementem neutralnym *monoidem* (albo *półgrupą z jedynką*).
A teraz wyobraźmy sobie takie uogólnienie operatora kompozycji:

```
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

Na przykład:

```
class WithLog<T> {
  public T value;
  public String log;
}
(f <|_WithLog g) a =
  WithLog b = g(a);
  WithLog c = f(b.value);
  return WithLog(value = c.value, log = b.log
+ c.log);
```

Och, ta matematyka

Matematycy nazywają operator łączny z elementem neutralnym *monoidem* (albo *półgrupą z jedynką*).
A teraz wyobraźmy sobie takie uogólnienie operatora kompozycji:

```
<|m :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

Na przykład:

```
class WithLog<T> {
  public T value;
  public String log;
}
(f <|WithLog g) a =
  WithLog b = g(a);
  WithLog c = f(b.value);
  return WithLog(value = c.value, log = b.log
+ c.log);
```

Och, ta matematyka

Matematycy nazywają operator łączny z elementem neutralnym *monoidem* (albo *półgrupą z jedynką*).
A teraz wyobraźmy sobie takie uogólnienie operatora kompozycji:

```
<|m :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

Na przykład:

```
class WithLog<T> {
  public T value;
  public String log;
}
(f <|WithLog g) a =
  WithLog b = g(a);
  WithLog c = f(b.value);
  return WithLog(value = c.value, log = b.log
+ c.log);
```

Och, ta matematyka

Matematycy nazywają operator łączny z elementem neutralnym *monoidem* (albo *półgrupą z jedynką*).
A teraz wyobraźmy sobie takie uogólnienie operatora kompozycji:

```
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
```

Na przykład:

```
class WithLog<T> {
  public T value;
  public String log;
}
(f <|_WithLog g) a =
  WithLog b = g(a);
  WithLog c = f(b.value);
  return WithLog(value = c.value, log = b.log
+ c.log);
```

Uogólnienie operatora identyczności

$\texttt{id}_{\texttt{WithLog}}$ `x = WithLog(value = x, log = "")`

Trójkę $(m, \texttt{<|}_m, \texttt{id}_m)$ nazywamy *monadą*.

Na przykład $(\texttt{WithLog}, \texttt{<|}_{\texttt{WithLog}}, \texttt{id}_{\texttt{WithLog}})$ jest monadą.

Inne popularne przykłady: `Optional`, `List`.

Uogólnienie operatora identyczności

$\texttt{id}_{\texttt{WithLog}}$ `x = WithLog(value = x, log = "")`

Trójkę $(m, \texttt{<|}_m, \texttt{id}_m)$ nazywamy *monadą*.

Na przykład $(\texttt{WithLog}, \texttt{<|}_{\texttt{WithLog}}, \texttt{id}_{\texttt{WithLog}})$ jest monadą.

Inne popularne przykłady: `Optional`, `List`.

Uogólnienie operatora identyczności

$\texttt{id}_{\texttt{WithLog}}$ `x = WithLog(value = x, log = "")`

Trójkę $(m, \texttt{<|}_m, \texttt{id}_m)$ nazywamy *monadą*.

Na przykład $(\texttt{WithLog}, \texttt{<|}_{\texttt{WithLog}}, \texttt{id}_{\texttt{WithLog}})$ jest monadą.

Inne popularne przykłady: `Optional`, `List`.

Uogólnienie operatora identyczności

$\texttt{id}_{\texttt{WithLog}}$ `x = WithLog(value = x, log = "")`

Trójkę $(m, \texttt{<|}_m, \texttt{id}_m)$ nazywamy *monadą*.

Na przykład $(\texttt{WithLog}, \texttt{<|}_{\texttt{WithLog}}, \texttt{id}_{\texttt{WithLog}})$ jest monadą.

Inne popularne przykłady: `Optional`, `List`.

Ale dlaczego?

Problem Haskella: leniwa ewaluacja.
Rozwiązanie: “system wejścia-wyjścia oparty na monadach”
Ale co to znaczy?!

Ale dlaczego?

Problem Haskella: leniwa ewaluacja.

Rozwiązanie: “system wejścia-wyjścia oparty na monadach”

Ale co to znaczy?!

Ale dlaczego?

Problem Haskella: leniwa ewaluacja.
Rozwiązanie: “system wejścia-wyjścia oparty na monadach”
Ale co to znaczy?!

Ale dlaczego?

Problem Haskella: leniwa ewaluacja.
Rozwiązanie: “system wejścia-wyjścia oparty na monadach”
Ale co to znaczy?!

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = 36$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = 36$$

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = 36$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = 36$$

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

```
square (2*3) = square 6 =def 6 * 6 = 36
```

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

```
square (2*3) =def (2*3) * (2*3) = 6 * 6 = 36
```

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

```
square (2*3) = square 6 =def 6 * 6 = 36
```

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

```
square (2*3) =def (2*3) * (2*3) = 6 * 6 = 36
```

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = \texttt{36}$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = \texttt{36}$$

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = \texttt{36}$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = \texttt{36}$$

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = \texttt{36}$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = \texttt{36}$$

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = \texttt{36}$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = \texttt{36}$$

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = \texttt{36}$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = \texttt{36}$$

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = \texttt{36}$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = \texttt{36}$$

Strategie ewaluacji

```
square x = x * x
```

Kolejność “aplikatywna” (wyewaluuj argumenty przed ekspansją funkcji):

$$\texttt{square (2*3)} = \texttt{square 6} =_{def} \texttt{6 * 6} = \texttt{36}$$

Kolejność “normalna” (wyewaluuj argumenty tak późno, jak się da):

$$\texttt{square (2*3)} =_{def} \texttt{(2*3) * (2*3)} = \texttt{6 * 6} = \texttt{36}$$

Problem Haskella: leniwa ewaluacja

```
readNumber()*3 + 2*readNumber()
< 1
< 0
```

Problem Haskella: leniwa ewaluacja

```
readNumber()*3 + 2*readNumber()
< 1
< 0
```

Problem Haskella: leniwa ewaluacja

```
readNumber()*3 + 2*readNumber()
< 1
< 0
```

Pomysł na rozwiązanie:

```
let   a       = readNumber(   ) in
  let   b       = readNumber(   ) in
     a*2 + 3*b
```

gdzie

```
let name = value in expression
(λ name -> expression) value
```

Pomysł na rozwiązanie:

```
let   a       = readNumber(   ) in
  let   b       = readNumber(   ) in
      a*2 + 3*b
```

gdzie

```
let name = value in expression
(λ name -> expression) value
```

Pomysł na rozwiązanie:

```
let   a       = readNumber(   ) in
  let   b       = readNumber(   ) in
     a*2 + 3*b
```

gdzie

```
let name = value in expression
(λ name -> expression) value
```

Pomysł na rozwiązanie:

```
let   a       = readNumber(   ) in
  let   b       = readNumber(   ) in
     a*2 + 3*b
```

gdzie

```
let name = value in expression
(λ name -> expression) value
```

Działające rozwiązanie:

```
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
     a*2 + 3*b
```

Lepsze rozwiązanie:

```
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
    (a*2 + 3*b, w2)
```

Wyciągnięcie do funkcji

```
myOperation :: RealWorld -> (Int, RealWorld)
myOperation w0 =
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
    (a*2 + 3*b, w2)
```

https://wiki.haskell.org/IO_inside

Problemy

- trzeba przekazywać dodatkowy parametr
- podatne na błędy (e.g. `w0` zamiast `w1`)
- rośnie nam poziom zagnieżdżeń

Problemy

- trzeba przekazywać dodatkowy parametr
- podatne na błędy (e.g. `w0` zamiast `w1`)
- rośnie nam poziom zagnieżdżeń

Problemy

- trzeba przekazywać dodatkowy parametr
- podatne na błędy (e.g. `w0` zamiast `w1`)
- rośnie nam poziom zagnieżdżeń

Problemy

- trzeba przekazywać dodatkowy parametr
- podatne na błędy (e.g. `w0` zamiast `w1`)
- rośnie nam poziom zagnieżdżeń

Zamiecenie w_n pod dywan

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Zamiecenie w_n pod dywan

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Zamiecenie w_n pod dywan

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
    continuation result w1
```

Zamiecenie w_n pod dywan

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Zamiecenie w_n pod dywan

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
pass readNumber
     (λ a -> pass readNumber
                  (λ b -> return a*2 + 3*b))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
pass readNumber
     (λ a -> pass readNumber
               (λ b -> λ w -> (a*2 + 3*b, w)))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
pass readNumber
     (λ a -> pass readNumber
               (λ b -> λ w -> (a*2 + 3*b, w)))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
        (λ b -> λ w -> (a*2 + 3*b, w)) y w2)

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
        (λ b -> λ w -> (a*2 + 3*b, w)) y w2)

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
        (λ w -> (a*2 + 3*y, w)) w2)

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
        (a*2 + 3*y, w2))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
pass readNumber (λ a -> λ w1 ->
  let (y, w2) = readNumber(w1) in
        (a*2 + 3*y, w2))

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
λ w0 -> let (x,w3) = readNumber(w0) in
   (λ a -> λ w1 -> let (y, w2) = readNumber(w1)
       in (a*2 + 3*y, w2)) x w3

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
   let (result, w1) = value w0 in
      continuation result w1
```

Ale czy to działa?

```
λ w0 -> let (x,w3) = readNumber(w0) in
   (λ a -> λ w1 -> let (y, w2) = readNumber(w1)
       in (a*2 + 3*y, w2)) x w3

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
   let (result, w1) = value w0 in
      continuation result w1
```

Ale czy to działa?

```
λ w0 -> let (x,w3) = readNumber(w0) in
   (λ w1 -> let (y, w2) = readNumber(w1) in
            (x*2 + 3*y, w2)) w3

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
   let (result, w1) = value w0 in
      continuation result w1
```

Ale czy to działa?

```
λ w0 -> let (x,w3) = readNumber(w0) in
  let (y, w2) = readNumber(w3) in
        (x*2 + 3*y, w2)

return value = λ world -> (value, world)

pass value continuation = λ w0 ->
  let (result, w1) = value w0 in
     continuation result w1
```

Ale czy to działa?

```
λ w0 -> let (x,w3) = readNumber(w0) in
  let (y, w2) = readNumber(w3) in
        (x*2 + 3*y, w2)
```

```
let (a,w1) = readNumber(w0) in
  let (b,w2) = readNumber(w1) in
    (a*2 + 3*b, w2)
```

To działa!

```
pass readNumber
      (λ a -> pass readNumber
                    (λ b -> return a*2 + 3*b))
```

Ale pisanie λ i rosnący poziom zagłębień są wkurzające!

```
pass readNumber (λ a
-> pass readNumber (λ b
-> return a*2 + 3*b))
```

To działa!

```
pass readNumber
      (λ a -> pass readNumber
                   (λ b -> return a*2 + 3*b))
```

Ale pisanie λ i rosnący poziom zagłębień są wkurzające!

```
pass readNumber (λ a
-> pass readNumber (λ b
-> return a*2 + 3*b))
```

To działa!

```
pass readNumber
      (λ a -> pass readNumber
                   (λ b -> return a*2 + 3*b))
```

Ale pisanie λ i rosnący poziom zagłębień są wkurzające!

```
pass readNumber (λ a
-> pass readNumber (λ b
-> return a*2 + 3*b))
```

Piramida zagłady

```
if ($_POST['user_name']) {
    if ($_POST['user_password_new']) {
        if ($_POST['user_password_new'] === $_POST['user_password_repeat']) {
            if (strlen($_POST['user_password_new']) > 5) {
                if (strlen($_POST['user_name']) < 65 && strlen($_POST['user_name']) > 1) {
                    if (preg_match('/^[a-z\d]{2,64}$/i', $_POST['user_name'])) {
                        $user = read_user($_POST['user_name']);
                        if (!isset($user['user_name'])) {
                            if ($_POST['user_email']) {
                                if (strlen($_POST['user_email']) < 65) {
                                    if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                        create_user();
                                        $_SESSION['msg'] = 'You are now registered so please login';
                                        header('Location: ' . $_SERVER['PHP_SELF']);
                                        exit();
                                    } else $msg = 'You must provide a valid email address';
                                } else $msg = 'Email must be less than 64 characters';
                            } else $msg = 'Email cannot be empty';
                        } else $msg = 'Username already exists';
                    } else $msg = 'Username must be only a-z, A-Z, 0-9';
                } else $msg = 'Username must be between 2 and 64 characters';
            } else $msg = 'Password must be at least 6 characters';
        } else $msg = 'Passwords do not match';
    } else $msg = 'Empty Password';
} else $msg = 'Empty Username';
$_SESSION['msg'] = $msg;
```

Lukier składniowy (`do`-notation):

```
do result <- action
   actions ...
```

przekształcamy do:

```
pass action (λ result -> do actions ...)
```

Uwaga: W Haskellu, `pass` zapisujemy jako »= i wymawiamy “bind”.

Lukier składniowy (`do`-notation):

```
do result <- action
   actions ...
```

przekształcamy do:

```
pass action (λ result -> do actions ...)
```

Uwaga: W Haskellu, `pass` zapisujemy jako »= i wymawiamy “bind”.

Lukier składniowy (`do`-notation):

```
do result <- action
   actions ...
```

przekształcamy do:

```
pass action (λ result -> do actions ...)
```

Uwaga: W Haskellu, `pass` zapisujemy jako »= i wymawiamy “bind”.

Lukier składniowy (`do`-notation):

```
do result <- action
   actions ...
```

przekształcamy do:

```
pass action (λ result -> do actions ...)
```

Uwaga: W Haskellu, `pass` zapisujemy jako »= i wymawiamy “bind”.

Relacja pomiędzy `pass` a `<|`

```
pass_m :: m a -> (a -> m b)
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
pass value function = (function <| id) value
(f <| g) x = pass (g x) f
return_m :: (a -> m a)
return_m = id_m
```

Relacja pomiędzy `pass` a `<|`

```
pass_m ::  m a -> (a -> m b)
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
pass value function = (function <| id) value
(f <| g) x = pass (g x) f
return_m ::  (a -> m a)
return_m = id_m
```

Relacja pomiędzy `pass` a `<|`

```
pass_m ::  m a -> (a -> m b)
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
pass value function = (function <| id) value
(f <| g) x = pass (g x) f
return_m ::  (a -> m a)
return_m = id_m
```

Relacja pomiędzy `pass` a `<|`

```
pass_m ::  m a -> (a -> m b)
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
pass value function = (function <| id) value
(f <| g) x = pass (g x) f
return_m ::  (a -> m a)
return_m = id_m
```

Relacja pomiędzy `pass` a `<|`

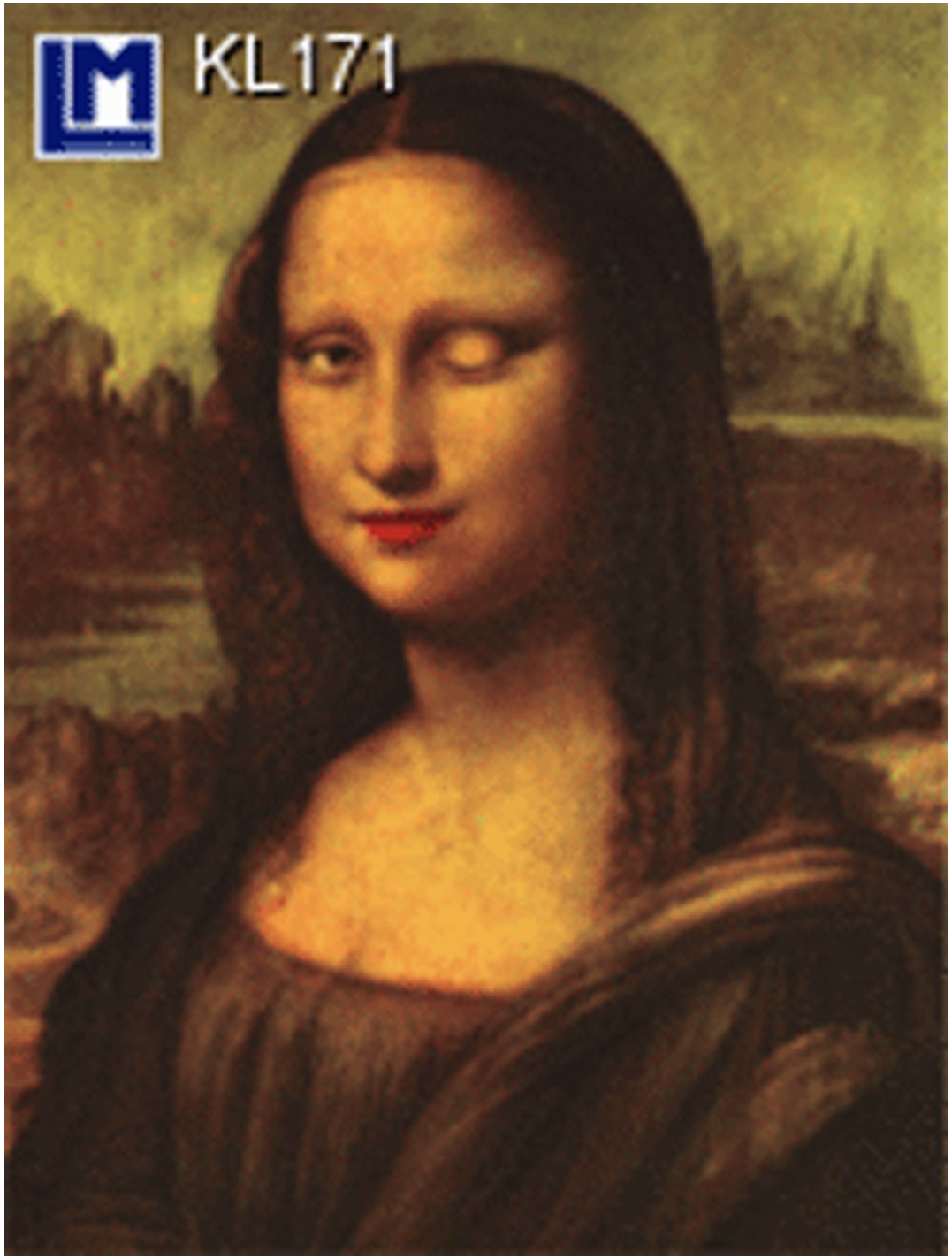
```
pass_m ::  m a -> (a -> m b)
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
pass value function = (function <| id) value
(f <| g) x = pass (g x) f
return_m ::  (a -> m a)
return_m = id_m
```

Relacja pomiędzy `pass` a `<|`

```
pass_m ::  m a -> (a -> m b)
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
pass value function = (function <| id) value
(f <| g) x = pass (g x) f
return_m ::  (a -> m a)
return_m = id_m
```

Relacja pomiędzy `pass` a `<|`

```
pass_m ::  m a -> (a -> m b)
<|_m :: (b -> m c) -> (a -> m b) -> (a -> m c)
pass value function = (function <| id) value
(f <| g) x = pass (g x) f
return_m ::  (a -> m a)
return_m = id_m
```


KL171