The Ultimatest Monad Tutorial

Panicz Maciej Godek

godek.maciek@gmail.com

15.12.2023

Agenda

- the concept of monads
- problems with Haskell
- pyramid of doom (with sugar coating)
- rants

Agenda

- the concept of monads
- problems with Haskell
- pyramid of doom (with sugar coating)
- rants

Agenda

- the concept of monads
- problems with Haskell
- pyramid of doom (with sugar coating)
- rants

Agenda

- the concept of monads
- problems with Haskell
- pyramid of doom (with sugar coating)
- rants

Agenda

- the concept of monads
- problems with Haskell
- pyramid of doom (with sugar coating)
- rants

Point-free programming

Inverse of a square root

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

point-free style:

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

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

in JS:

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

Point-free programming

Inverse of a square root

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

point-free style:

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

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

in JS:

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

Point-free programming

Inverse of a square root

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

point-free style:

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

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

in JS:

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

Point-free programming

Inverse of a square root

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

point-free style:

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

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

in JS:

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

Point-free programming

Inverse of a square root

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

point-free style:

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

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

in JS:

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

Point-free programming

Inverse of a square root

`isqrt x = 1/(sqrt x)`

point-free style:

`isqrt = (1/) . sqrt`

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

in JS:

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

Function composition operator

Type of the function composition operator:

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

looks a bit awkward, but if we define

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

the type of the “swapped composition” is

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

vide UNIX pipes

or “the uncle of the friend of my brother” vs. “my brother's friend's uncle”

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

Function composition operator

Type of the function composition operator:

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

looks a bit awkward, but if we define

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

the type of the "swapped composition" is

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

vide UNIX pipes

or "the uncle of the friend of my brother" vs. "my brother's friend's uncle"

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

Function composition operator

Type of the function composition operator:

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

looks a bit awkward, but if we define

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

the type of the “swapped composition” is

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

vide UNIX pipes

or “the uncle of the friend of my brother” vs. “my brother's friend's uncle”

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

Function composition operator

Type of the function composition operator:

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

looks a bit awkward, but if we define

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

the type of the "swapped composition" is

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

vide UNIX pipes

or "the uncle of the friend of my brother" vs. "my brother's friend's uncle"

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

Function composition operator

Type of the function composition operator:

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

looks a bit awkward, but if we define

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

the type of the “swapped composition” is

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

vide UNIX pipes

or “the uncle of the friend of my brother” vs. “my brother's friend's uncle”

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

Function composition operator

Type of the function composition operator:

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

looks a bit awkward, but if we define

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

the type of the "swapped composition" is

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

vide UNIX pipes

or "the uncle of the friend of my brother" vs. "my brother's friend's uncle"

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

Function composition operator

Type of the function composition operator:

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

looks a bit awkward, but if we define

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

the type of the "swapped composition" is

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

vide UNIX pipes

or "the uncle of the friend of my brother" vs. "my brother's friend's uncle"

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

Function composition operator

Type of the function composition operator:

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

looks a bit awkward, but if we define

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

the type of the “swapped composition” is

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

vide UNIX pipes

or “the uncle of the friend of my brother” vs. “my brother's friend's uncle”

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

Function composition operator

Properties of function composition:

- associative: `f . (g . h) = (f . g) . h`
  like: `x + (y + z) = (x + y) + z`
  or: `x * (y * z) = (x * y) * z`
- has a neutral element `id`:
  `f . id = id . f = f`
  like: `x + 0 = 0 + x = x`
  or: `x * 1 = 1 * x = x`

where the `id` function is defined as

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

All that math

In mathematics, an associative operator with neutral element is called *a monoid* (or *semigroup with identity*).

Now imagine the following *generalization* of the composition operator:

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

For example:

```
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);
```

All that math

In mathematics, an associative operator with neutral element is called *a monoid* (or *semigroup with identity*).
Now imagine the following *generalization* of the composition operator:

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

For example:

```
class WithLog<T> {
  public T value;
  public String log;
}
```

$(f <|_{\texttt{WithLog}} g)$ a =

```
  WithLog b = g(a);
  WithLog c = f(b.value);
  return WithLog(value = c.value, log = b.log
+ c.log);
```

All that math

In mathematics, an associative operator with neutral element is called *a monoid* (or *semigroup with identity*).
Now imagine the following *generalization* of the composition operator:

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

For example:

```
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);
```

All that math

In mathematics, an associative operator with neutral element is called *a monoid* (or *semigroup with identity*).
Now imagine the following *generalization* of the composition operator:

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

For example:

```
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);
```

All that math

In mathematics, an associative operator with neutral element is called *a monoid* (or *semigroup with identity*).

Now imagine the following *generalization* of the composition operator:

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

For example:

```
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);
```

Generalization of idenity

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

The triple $(m, <|_m, \texttt{id}_m)$ is called *a monad*.

For example, $(\texttt{WithLog}, <|_{\texttt{WithLog}}, \texttt{id}_{\texttt{WithLog}})$ is a monad.

Other popular examples: `Optional`, `List`.

Generalization of idenity

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

The triple $(m,\ \mathtt{<|}_m,\ \mathtt{id}_m)$ is called *a monad*.

For example, $(\mathtt{WithLog},\ \mathtt{<|}_{\mathtt{WithLog}},\ \mathtt{id}_{\mathtt{WithLog}})$ is a monad.

Other popular examples: `Optional`, `List`.

Generalization of idenity

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

The triple $(m,\ \mathtt{<|}_m,\ \mathtt{id}_m)$ is called *a monad*.

For example, $(\mathtt{WithLog},\ \mathtt{<|}_{\mathtt{WithLog}},\ \mathtt{id}_{\mathtt{WithLog}})$ is a monad.

Other popular examples: `Optional`, `List`.

Generalization of idenity

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

The triple $(m,\ \texttt{<|}_m,\ \texttt{id}_m)$ is called *a monad*.

For example, $(\texttt{WithLog},\ \texttt{<|}_{\texttt{WithLog}},\ \texttt{id}_{\texttt{WithLog}})$ is a monad.

Other popular examples: `Optional`, `List`.

But why?

Problem with Haskella: lazy evaluation.
Solution: “input/output system based on monads”
But what does it mean?

But why?

Problem with Haskella: lazy evaluation.

Solution: “input/output system based on monads”

But what does it mean?

But why?

Problem with Haskella: lazy evaluation.
Solution: “input/output system based on monads”
But what does it mean?

But why?

Problem with Haskella: lazy evaluation.
Solution: “input/output system based on monads”
But what does it mean?

Evaluation strategies

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

The "applicative" order (evaluate arguments before expanding function):

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

The "normal" order (evaluate arguments as late as possible):

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

Evaluation strategies

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

The “applicative” order (evaluate arguments before expanding function):

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

The “normal” order (evaluate arguments as late as possible):

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

Evaluation strategies

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

The “applicative” order (evaluate arguments before expanding function):

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

The “normal” order (evaluate arguments as late as possible):

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

Evaluation strategies

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

The "applicative" order (evaluate arguments before expanding function):

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

The "normal" order (evaluate arguments as late as possible):

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

Evaluation strategies

`square x = x * x`

The “applicative” order (evaluate arguments before expanding function):

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

The “normal” order (evaluate arguments as late as possible):

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

Evaluation strategies

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

The “applicative” order (evaluate arguments before expanding function):

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

The “normal” order (evaluate arguments as late as possible):

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

Evaluation strategies

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

The “applicative” order (evaluate arguments before expanding function):

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

The “normal” order (evaluate arguments as late as possible):

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

Evaluation strategies

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

The “applicative” order (evaluate arguments before expanding function):

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

The “normal” order (evaluate arguments as late as possible):

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

Evaluation strategies

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

The “applicative” order (evaluate arguments before expanding function):

square (2⋆3) = square 6 $=_{def}$ 6 ⋆ 6 = 36

The “normal” order (evaluate arguments as late as possible):

square (2⋆3) $=_{def}$ (2⋆3) ⋆ (2⋆3) = 6 ⋆ 6 = 36

Evaluation strategies

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

The "applicative" order (evaluate arguments before expanding function):

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

The "normal" order (evaluate arguments as late as possible):

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

Evaluation strategies

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

The “applicative” order (evaluate arguments before expanding function):

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

The “normal” order (evaluate arguments as late as possible):

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

The problem with Haskell: lazy evaluation

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

The problem with Haskell: lazy evaluation

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

The problem with Haskell: lazy evaluation

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

Idea for a solution

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

gdzie

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

Idea for a solution

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

gdzie

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

Idea for a solution

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

gdzie

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

Idea for a solution

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

gdzie

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

A working solution

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

A better solution

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

Extracting a function

```
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

Problems

- we need to pass additional argument
- error-prone (e.g. `w0` zamiast `w1`)
- indentation level increases

Problems

- we need to pass additional argument
- error-prone (e.g. `w0` zamiast `w1`)
- indentation level increases

Problems

- we need to pass additional argument
- error-prone (e.g. `w0` zamiast `w1`)
- indentation level increases

Problems

- we need to pass additional argument
- error-prone (e.g. `w0` zamiast `w1`)
- indentation level increases

Sweeping w_n under the rug

```
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
```

Sweeping w_n under the rug

```
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
```

Sweeping $\mathtt{w}_n$ under the rug

```
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
```

Sweeping w_n under the rug

```
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
```

Sweeping w_n under the rug

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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
```

But does it work?

```
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)
```

It works!

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

But typing λ and the increasing indentation level are annoying!

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

It works!

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

But typing λ and the increasing indentation level are annoying!

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

It works!

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

But typing λ and the increasing indentation level are annoying!

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

“The pyramid of doom”

```
$msg = '';
if ($_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;
```

Syntactic sugar (do-notation):

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

is transformed to:

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

Note: In Haskell, `pass` is written down as »= and pronounced "bind".

Syntactic sugar (`do`-notation):

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

is transformed to:

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

Note: In Haskell, `pass` is written down as `>>=` and pronounced "bind".

Syntactic sugar (`do`-notation):

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

is transformed to:

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

Note: In Haskell, `pass` is written down as »= and pronounced "bind".

Syntactic sugar (`do`-notation):

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

is transformed to:

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

Note: In Haskell, `pass` is written down as »= and pronounced "bind".

Relation between `pass` and `<|`

```
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
```

Relation between `pass` and `<|`

```
passm :: 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
returnm :: (a -> m a)
returnm = idm
```

Relation between `pass` and `<|`

```
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
```

Relation between `pass` and `<|`

```
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
```

Relation between `pass` and `<|`

```
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
```

Relation between `pass` and `<|`

```
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
```

Relation between `pass` and `<|`

```
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
```