PDBP

Program Description Based Programming
Scala eXchange

Luc Duponcheel

August 8, 2018

Intro

Intro

- this talk is about

Intro

- this talk is about
    - a `Dotty` *library* `PDBP`

Intro

- this talk is about
  - a `Dotty` *library* `PDBP`
  - inspired by the *function-level programming language* `FP`

John Backus

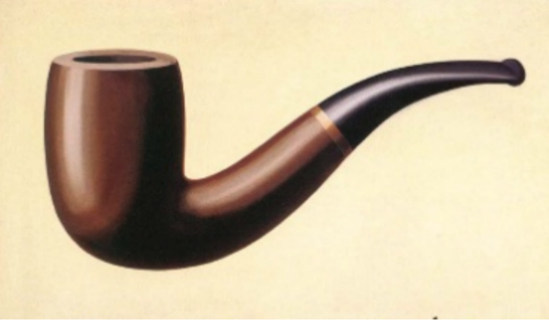
John Backus

- ACM Turing Award Winner 1977

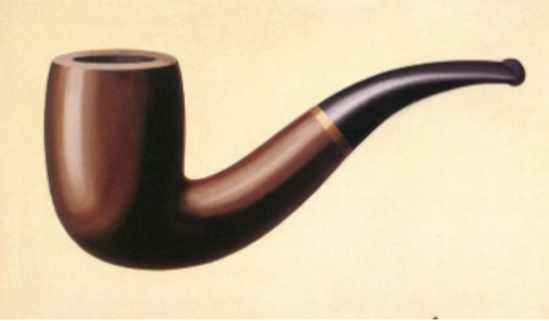
John Backus

- ACM Turing Award Winner 1977
- *Can programming be liberated from the Von Neumann style?*

What is this?

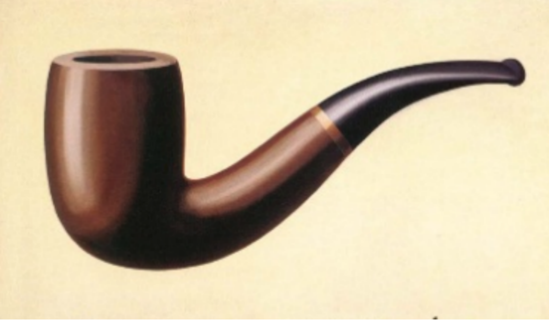


What is this?



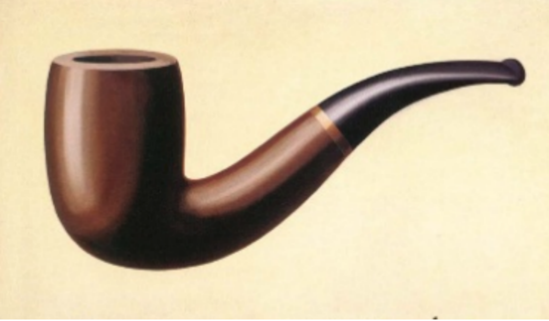
- A *pipe*?

What is this?



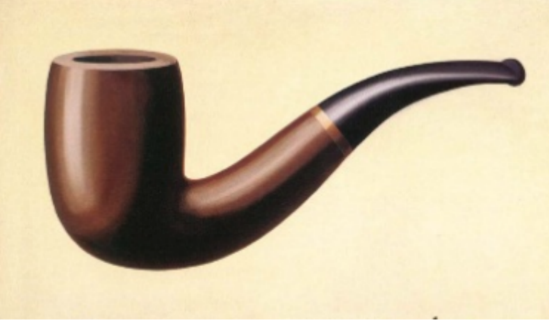
- A *pipe*?
- A painting *describing* a pipe?

What is this?



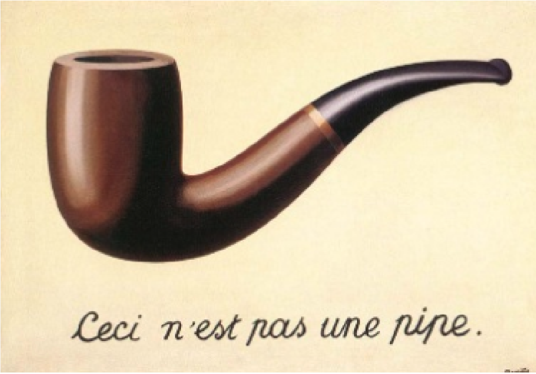
- A *pipe*?
- A painting *describing* a pipe?
- A slide *describing* a painting describing a pipe?

What is this?

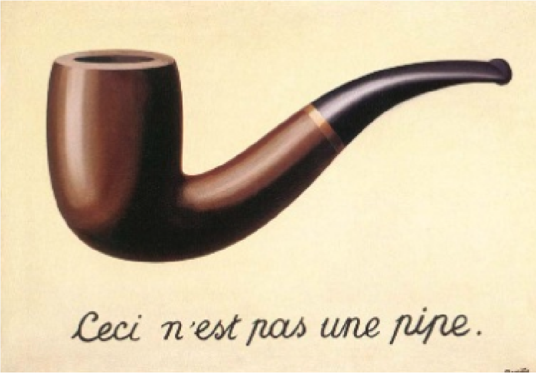


- A *pipe*?
- A painting *describing* a pipe?
- A slide *describing* a painting describing a pipe?
- ...

Ceci n’est pas une pipe



Ceci n'est pas une pipe



- It is a painting by René Magritte *describing* a pipe

This is not a program

```
val program: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        program
    } `in` {
      multiply
    }
  }
```

This is not a program

```
val program: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        program
    } `in` {
      multiply
    }
  }
```

- It is `Dotty` code *describing* a program

This is not a program

```
val program: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        program
    } `in` {
      multiply
    }
  }
```

- It is `Dotty` code *describing* a program
- It is a *program description*

This is not a program

```
val program: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        program
    } `in` {
      multiply
    }
  }
```

- It is Dotty code *describing* a program
- It is a *program description*
- Can you think of a more meaningful name?

Program Description

Program Description

- by abuse of notation we are simply going to write *program* instead of *program description*

Program Description

- by abuse of notation we are simply going to write *program* instead of *program description*
- we hope that this will not lead to any confusion

Type class `Program`

```
trait Program[>-->[- _, + _]]
```

factorial program

```
val factorial: BigInt >--> BigInt =
  ‘if‘(isZero) {
    one
  } ‘else‘ {
    ‘let‘ {
      subtractOne >-->
        factorial
    } ‘in‘ {
      multiply
    }
  }
```


`factorial` program

```
val factorial: BigInt >--> BigInt =
  ‘if‘(isZero) {
    one
  } ‘else‘ {
    ‘let‘ {
      subtractOne >-->
        factorial
    } ‘in‘ {
      multiply
    }
  }
```

- `factorial` *definition* uses
  the capabilities *declared* in `Program`

`factorial` implementations

```
val factorial: BigInt >--> BigInt =
  ‘if‘(isZero) {
    one
  } ‘else‘ {
    ‘let‘ {
      subtractOne >-->
        factorial
    } ‘in‘ {
      multiply
    }
  }
```


`factorial` implementations

```
val factorial: BigInt >--> BigInt =
  ‘if‘(isZero) {
    one
  } ‘else‘ {
    ‘let‘ {
      subtractOne >-->
        factorial
    } ‘in‘ {
      multiply
    }
  }
```

- `factorial` *implementations* depend on `implicit object`'s *defining* the capabilities of `Program`

factorial meanings

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

`factorial` meanings

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- `factorial` *meanings* use
  natural transformations transforming implementations

program

```
val program: Z >--> Y
```

Composition

```
val `z>-->y`: Z >--> Y
val `y>-->x`: Y >--> X

val `z>-->x`: Z >--> X = `z>-->y >--> `y>-->x`
```

program

(design artifact)

```
 ---------------------------------
|  ----------        ----------   |
| | Z >--> Y | >--> | Y >--> X |  |
|  ----------        ----------   |
 ---------------------------------
                ||
                \/
           ----------
          | Z >--> X |
           ----------
```

mainProgram

```
val program: Unit >--> Unit
```

mainProgram

```
val producer: Unit >--> Z
val program: Z >--> Y
val consumer: Y >--> Unit

val mainProgram: Unit >--> Unit =
  producer >--> program >--> consumer
```


`mainProgram`

(architectural artifact)

```
 ------------------------------------------------
|  ----------------    ----    ----------------  |
| | Unit >--> Unit |-| ???? |-| Unit >--> Unit | |
|  ----------------    ----    ----------------  |
 ------------------------------------------------
                        ||
                        \/
                 ----------------
                | Unit >--> Unit |
                 ----------------
```

FP versus PDBP

FP versus PDBP

- FP and PDBP

FP versus PDBP

- FP and PDBP
    - promote *pointfree*, *composition* based, *functional programming*

FP versus PDBP

- FP and PDBP
  - promote *pointfree*, *composition* based, *functional programming*
- FP is a *language*
  PDBP is a *library*

`FP` versus `PDBP`

- `FP` and `PDBP`
  - promote *pointfree*, *composition* based, *functional programming*
- `FP` is a *language*
  `PDBP` is a *library*
  - `FP` is *heterogeneous*
    `PDBP` is *homogeneous*

`FP` versus `PDBP`

- `FP` and `PDBP`
  - promote *pointfree*, *composition* based, *functional programming*

- `FP` is a *language*
  `PDBP` is a *library*
  - `FP` is *heterogeneous*
    `PDBP` is *homogeneous*
  - `FP` *semantics* is *fixed*
    `PDBP` *semantics* is *not fixed*

FP versus PDBP

- FP and PDBP
  - promote *pointfree*, *composition* based, *functional programming*
- FP is a *language*
  PDBP is a *library*
  - FP is *heterogeneous*
    PDBP is *homogeneous*
  - FP *semantics* is *fixed*
    PDBP *semantics* is *not fixed*
  - FP *capabilities* are *fixed*
    PDBP *capabilities* are *not fixed*

FP versus PDBP

- FP and PDBP
  - promote *pointfree*, *composition* based, *functional programming*

- FP is a *language*
  PDBP is a *library*
  - FP is *heterogeneous*
    PDBP is *homogeneous*
  - FP *semantics* is *fixed*
    PDBP *semantics* is *not fixed*
  - FP *capabilities* are *fixed*
    PDBP *capabilities* are *not fixed*
  - FP *effects* are *impure*
    PDBP *effects* are *pure*

Homogeneous

Homogeneous

- programs are objects

Homogeneous

- programs are objects
  - programming using `PDBP` is all about
    passing around programming capabilities

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *production*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *production*
  - *recursion* using *stack*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *production*
  - *recursion* using *stack*
  - *recursion* using *heap*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *production*
  - *recursion* using *stack*
  - *recursion* using *heap*
- *test*

Semantics

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

- *production*
  - *recursion* using *stack*
  - *recursion* using *heap*
- *test*
  - ...

Capabilities

Capabilities

- *manipulating state*

Capabilities

- *manipulating state*
- *handling failure*

Capabilities

- *manipulating state*
- *handling failure*
- *handling latency*

Capabilities

- *manipulating state*
- *handling failure*
- *handling latency*
- *handling control*

Capabilities

- *manipulating state*
- *handling failure*
- *handling latency*
- *handling control*
- ...

Effects

Effects

- *reading*

Effects

- *reading*
- *writing*

Foundations

Foundations

- `trait Program[>-->[- _, + _]]`
  corresponds to *arrows*

Foundations

- `trait Program[>-->[- _, + _]]`
  corresponds to *arrows*
- `trait Computation[C[+ _]]`
  corresponds to *monads*

Arrows versus monads

Arrows versus monads

- arrows generalize *functions*

Arrows versus monads

- arrows generalize *functions*
- *composition* based, *pointfree* functional programming

Arrows versus monads

- arrows generalize *functions*
- *composition* based, *pointfree* functional programming
- `val 'z=>x'   = 'z=>y' andThen 'y=>x'`

Arrows versus monads

- arrows generalize *functions*
- *composition* based, *pointfree* functional programming
- ```
      val `z=>x`   = `z=>y` andThen `y=>x`
  ```
- ```
      val `z>-->x` = `z>-->y` >-->  `y>-->z`
  ```

Arrows versus monads

Arrows versus monads

- monads generalize *expressions*

Arrows versus monads

- monads generalize *expressions*
- *binding* and *result* based, *pointful* functional programming

Arrows versus monads

- monads generalize *expressions*
- *binding* and *result* based, *pointful* functional programming
- `{ val z = ez ; { val y = ey ; ex(z,y) } }`

Arrows versus monads

- monads generalize *expressions*
- *binding* and *result* based, *pointful* functional programming
- `{ val z = ez ; { val y = ey ; ex(z,y) } }`
- `mz bind { z => my bind { y => mx(z,y) } }`

Arrows versus monads

- monads generalize *expressions*
- *binding* and *result* based, *pointful* functional programming
- `{ val z = ez ; { val y = ey ; ex(z,y) } }`
- `mz bind { z => my bind { y => mx(z,y) } }`
- `mz bind { z => my bind { y => result(ex(z,y)) } }`

Arrows versus monads (kleisli arrows)

Arrows versus monads (kleisli arrows)

- `val function:  Z => Y =  z => ey(x)`

Arrows versus monads (kleisli arrows)

- `val function:  Z => Y =  z => ey(x)`
    - expression is used to define function

Arrows versus monads (kleisli arrows)

- `val function:  Z => Y =  z => ey(x)`
  - expression is used to define function
- `val kleisliArrow:  Z => M[Y] =  z => my(x)`

Arrows versus monads (kleisli arrows)

- `val function: Z => Y = z => ey(x)`
  - expression is used to define function
- `val kleisliArrow: Z => M[Y] = z => my(x)`
  - monad is used to define kleisli arrow

Arrows versus monads

Arrows versus monads

- *arrows* can be programmed *pointful* (*arrow calculus*)

Arrows versus monads

- *arrows* can be programmed *pointful* (*arrow calculus*)
- *monads* can be programmed *pointfree* (*kleisli arrows*)

Power of expressions

Power of expressions

- *monads* are more *concrete* (less *abstract*) than *arrows*

Power of expressions

- *monads* are more *concrete* (less *abstract*) than *arrows*
    - *monads allow more description liberty*

Power of expressions

- *monads* are more *concrete* (less *abstract*) than *arrows*
    - *monads allow more description liberty*
    - *monads impose more implementation constraints*

Power of expressions

- *monads* are more *concrete* (less *abstract*) than *arrows*
  - *monads allow more description liberty*
  - *monads impose more implementation constraints*
- *Constraints Liberate, Liberties Constrain*

Elegance of use

Elegance of use

- *pointfree* programming is sometimes considered to be *abstruse*

Elegance of use

- *pointfree* programming is sometimes considered to be *abstruse*
- `Dotty` comes to the rescue

Elegance of use

- *pointfree* programming is sometimes considered to be *abstruse*
- `Dotty` comes to the rescue
  - `Dotty` is a `Sca`lable `la`nguage

Elegance of use

- *pointfree* programming is sometimes considered to be *abstruse*
- `Dotty` comes to the rescue
    - `Dotty` is a `Sca`lable `la`nguage
    - `Dotty` *library based* language extensions are *type safe*

Elegance of use

- *pointfree* programming is sometimes considered to be *abstruse*
- `Dotty` comes to the rescue
  - `Dotty` is a `Sca`lable `la`nguage
  - `Dotty` *library based* language extensions are *type safe*
- `PDBP` comes with a *program description DSL*

Program Description DSL

```
val factorial: BigInt >--> BigInt =
  `if`(isZero) {
    one
  } `else` {
    `let` {
      subtractOne >-->
        factorial
    } `in` {
      multiply
    }
  }
```

Computation Description DSL

```
val factorial: BigInt => C[BigInt] = { z =>
  isZero(z) bind { b =>
    if (b) {
      one(z)
    } else {
      subtractOne(z) bind { y =>
        factorial(y) bind { x =>
          multiply((y, x))
        }
      }
    }
  }
}
```


Uh! Oh!

```
val factorial: BigInt => C[BigInt] = { z =>
  isZero(z) bind { b =>
    if (b) {
      one(z)
    } else {
      subtractOne(z) bind { y =>
        factorial(y) bind { x =>
          multiply((z, x))
        }
      }
    }
  }
}
```


PDBP's choice

PDBP's choice

- the PDBP libary goes for

PDBP's choice

- the `PDBP` libary goes for
  - `private[pdbp]` *pointful monad API*
    provides *power of expression* for *library* developers

PDBP's choice

- the PDBP libary goes for
  - `private[pdbp]` *pointful monad API*
    provides *power of expression* for *library* developers
  - `public` *pointfree arrow API*
    provides *elegance of use* for *application* developers

PDBP's choice

- the PDBP libary goes for
  - `private[pdbp]` *pointful monad API*
    provides *power of expression* for *library* developers
  - `public` *pointfree arrow API*
    provides *elegance of use* for *application* developers
- the PDBP can live with

PDBP's choice

- the PDBP libary goes for
  - `private[pdbp]` *pointful monad API*
    provides *power of expression* for *library* developers
  - `public` *pointfree arrow API*
    provides *elegance of use* for *application* developers
- the PDBP can live with
  - corresponding implementation constraints

PDBP

```
//       _______         __    __        _______
//      / ___  /\       / /\  / /\      / ___  /\
//     / /__/ / / _____/ / / / /_/__   / /__/ / /
//    / _____/ / / ___  / / / ___  /\ /____  / /
//   / /\____\/ / /__/ / / / /__/ / / \___/ / /
//  /_/ /      /______/ / /______/ /     /_/ /
//  \_\/       \______\/  \______\/      \_\/
//                                           v1.0
//  Program Description Based Programming Library
//  author        Luc Duponcheel        2017-2018
```

PDBP library design decisions

PDBP library design decisions

- *syntax* separated from *semantics*

PDBP library design decisions

- *syntax* separated from *semantics*
- *syntax*

PDBP library design decisions

- *syntax* separated from *semantics*
- *syntax*
  - `trait`'s (*type classes*) *declare programming capabilities*

`PDBP` library design decisions

- *syntax* separated from *semantics*
- *syntax*
    - `trait`'s (*type classes*) *declare programming capabilities*
    - *program definitions* use them

`PDBP` library design decisions

- *syntax* separated from *semantics*
- *syntax*
  - `trait`'s (*type classes*) *declare programming capabilities*
  - *program definitions* use them
- *semantics*

`PDBP` library design decisions

- *syntax* separated from *semantics*
- *syntax*
  - `trait`'s (*type classes*) *declare programming capabilities*
  - *program definitions* use them
- *semantics*
  - `implicit object`'s *define programming capabilities*

`PDBP` library design decisions

- *syntax* separated from *semantics*
- *syntax*
  - `trait`'s (*type classes*) *declare programming capabilities*
  - *program definitions* use them
- *semantics*
  - `implicit object`'s *define programming capabilities*
  - *program implementations* depend on them

`PDBP` library design decisions

- *syntax* separated from *semantics*
- *syntax*
  - `trait`'s (*type classes*) *declare programming capabilities*
  - *program definitions* use them
- *semantics*
  - `implicit object`'s *define programming capabilities*
  - *program implementations* depend on them
    - *implicitly*

`PDBP` library design decisions

- *syntax* separated from *semantics*

- *syntax*
  - `trait`'s (*type classes*) *declare programming capabilities*
  - *program definitions* use them
- *semantics*
  - `implicit object`'s *define programming capabilities*
  - *program implementations* depend on them
    - *implicitly*

  - `[implicit] object`'s *define natural transformations*

PDBP library design decisions

- *syntax* separated from *semantics*
- *syntax*
    - `trait`'s (*type classes*) *declare programming capabilities*
    - *program definitions* use them
- *semantics*
    - `implicit object`'s *define programming capabilities*
    - *program implementations* depend on them
        - *implicitly*
    - `[implicit] object`'s *define natural transformations*
    - *program meanings* use them

`PDBP` library design decisions

- *syntax* separated from *semantics*

- *syntax*
    - `trait`'s (*type classes*) *declare programming capabilities*
    - *program definitions* use them
- *semantics*
    - `implicit object`'s *define programming capabilities*
    - *program implementations* depend on them
        - *implicitly*

    - `[implicit] object`'s *define natural transformations*
    - *program meanings* use them
        - *transforming program implementations*

PDBP library design decisions

`PDBP` library design decisions

- programs are *defined* in `class`'es
  declared to *implicitly* have programming capabilities

`PDBP` library design decisions

- programs are *defined* in `class`'es
  declared to *implicitly* have programming capabilities

- programs are *implemented* in `object`'s
  using *implicit dependency injection* by `import`

`Program` (cfr. *arrow*)

```
trait Program[>-->[- _, + _]]
```

Computation (cfr. *monad*)

```
trait Computation[C[+ _]]
```

Liskov Substitution Principle

Liskov Substitution Principle

- *impose less*

Liskov Substitution Principle

- *impose less*
- *provide more*

Internet Robustness Principle

Internet Robustness Principle

- *be liberal in what you receive*

Internet Robustness Principle

- *be liberal in what you receive*
- *be generous in what you send*

PDBP library details

Program

```
trait Program[>-->[- _, + _]]
    extends Function[>-->]
    with Composition[>-->]
    with Construction[>-->]
    with Condition[>-->]

    with Aggregation[>-->]
```


Program

```
trait Program[>-->[- _, + _]]
    extends Function[>-->]
    with Composition[>-->]
    with Construction[>-->]
    with Condition[>-->]

    with Aggregation[>-->]
```

- `trait`'s correspond to FP *forms*

`Function`

Function

- `` val `z>-->y` = function(`z=>y`) ``

Function

- `val ‘z>-->y‘ = function(‘z=>y‘)`
- *pure functions* are *atomic programs*

Function

- `` val `z>-->y` = function(`z=>y`) ``
- *pure functions* are *atomic programs*
    - up to you to define *granularity*

Composition

Composition

- `val ‘z>-->x‘ = ‘z>-->y‘ >--> ‘y>-->x‘`

Construction

Construction

- ```
  val `z>-->y&&x` = `z>-->y` & `z>-->x`
  ```


  ```

Construction

- val ‘z>-->y&&x‘ = ‘z>-->y‘ & ‘z>-->x‘
- val ‘z&&y>-->x&&w‘ = ‘z>-->x‘ && ‘y>-->w‘

Construction

- ```
  val ‘z>-->y&&x‘ = ‘z>-->y‘ & ‘z>-->x‘
  ```
- ```
  val ‘z&&y>-->x&&w‘ = ‘z>-->x‘ && ‘y>-->w‘
  ```
- ```
  val ‘z>-->x‘ =
    ‘let‘ ‘z>-->y‘
    ‘in‘ ‘z&&y>-->x‘
  ```


  ```

Condition

Condition

- ```
  val ‘y||x>-->z‘ = ‘y>-->z‘ | ‘x>-->z‘
  ```


  ```

Condition

- `val ‘y||x>-->z‘ = ‘y>-->z‘ | ‘x>-->z‘`
- `val ‘x||w>-->z||y‘ = ‘x>-->z‘ || ‘w>-->y‘`

Condition

- ```
  val `y||x>-->z` = `y>-->z` | `x>-->z`
  ```
- ```
  val `x||w>-->z||y` = `x>-->z` || `w>-->y`
  ```
- ```
  val `y>-->z` =
    `if`(`y>-->b`) `y>-t->z`
    `else` `y>-f->z`
  ```


  ```

Computation

```
private[pdbp] trait Computation[C[+ _]]
    extends Resulting[C]
    with Binding[C]
    with Program[[-Z, +Y] => Z => C[Y]]

    with Lifting[C]

    with Sequencing[C]
```


Resulting

Resulting

- `val cz = result(z)`

Binding

Binding

- val cy = cz bind { z => `z=>cy`(y) }

Binding

- `val cy = cz bind { z => `z=>cy`(y) }`
- `val cy = cz bind { z => result(`z=>y`(y)) }`

Kleisli

Kleisli

- `type Kleisli[C[+ _]] = [-Z, + Y] => Z => C[Y]`

Kleisli

- `type Kleisli[C[+ _]] = [-Z, + Y] => Z => C[Y]`

- ```
  private[pdbp] trait Computation[C[+ _]]
      extends Resulting[C]
      with Binding[C]
      with Program[Kleisli[C]]

      // ...
  ```



  ```

factorial Helper Functions

```
val isZeroFunction: BigInt => Boolean = { i =>
  i == 0
}

def oneFunction[Z]: Z => BigInt = { z =>
  1
}

val subtractOneFunction: BigInt => BigInt = { i =>
  i - 1
}

val multiplyFunction: (BigInt && BigInt) => BigInt = { (i, j) =>
  i * j
}
```


factorial Helper Programs

```
val isZeroHelper: BigInt >--> Boolean =
  function(isZeroFunction)

val subtractOneHelper: BigInt >--> BigInt =
  function(subtractOneFunction)

val multiplyHelper: (BigInt && BigInt) >--> BigInt =
  function(multiplyFunction)

def oneHelper[Z]: Z >--> BigInt =
  function(oneFunction)
```

factorial Atomic Programs

```
val isZero: BigInt >--> Boolean =
  isZeroHelper

val subtractOne: BigInt >--> BigInt =
  subtractOneHelper

val multiply: (BigInt && BigInt) >--> BigInt =
  multiplyHelper

def one[Z]: Z >--> BigInt =
  oneHelper
```

factorial program

```
val factorial: BigInt >--> BigInt =
  ‘if‘(isZero) {
    one
  } ‘else‘ {
    ‘let‘ {
      subtractOne >-->
        factorial
    } ‘in‘ {
      multiply
    }
  }
```

mainFactorial

```
val producer: Unit >--> BigInt

val consumer: BigInt >--> Unit

lazy val mainFactorial: Unit >--> Unit =
  producer >-->
    factorial >-->
    consumer
```

activeTypes

```
object activeTypes {
  type Active[+Z] = Z
  type `=>A` = Kleisli[Active]
}
```


activeProgram

```
implicit object activeProgram
    extends Computation[Active]
    with Program[`=>A`] {

  override private[pdbp] def result[Z]: Z => Active[Z] =
    `z=>az`

  override private[pdbp] def bind[Z, Y](
      az: Active[Z],
      `z=>ay`: => (Z => Active[Y])): Active[Y] =
    `z=>ay`(az)

}
```


activeMeaningOfActive

```
implicit object activeMeaningOfActive
    extends MeaningOfActive[Active]()
    with ComputationMeaning[Active, Active]()
    with ProgramMeaning[`=>A`, `=>A`]()
```

FactorialMain

```
import ... activeTypes._

import ... implicits.activeProgram

import ... MainFactorial
```

Effectful producer and consumer

```
private def effectfulReadIntFromConsoleWithMessage(
    message: String): Unit >--> BigInt =
  function(effectfulReadIntFromConsoleFunction(message))

private def effectfulWriteLineToConsoleWithMessage[Y](
    message: String): Y >--> Unit =
  function[Y, Unit](effectfulWriteLineToConsoleFunction(message))

val effectfulReadIntFromConsole: Unit >--> BigInt =
  effectfulReadIntFromConsoleWithMessage("please type an integer")

val effectfulWriteFactorialOfIntToConsole: BigInt >--> Unit =
  effectfulWriteLineToConsoleWithMessage(
    "the factorial value of the integer is")
```


FactorialMain

```
object FactorialMain extends MainFactorial[`=>A`]() {

  // ... effectful utilities

  override val producer =
    effectfulReadIntFromConsole

  override val consumer =
    effectfulWriteFactorialOfIntToConsole

}
```


FactorialMain

```
object FactorialMain extends MainFactorial[`=>A`]() {
  def main(args: Array[String]): Unit = {
    import ... activeMeaningOfActive.meaning
    meaning(mainFactorial)(())
  }
}
```


Problems and Solutions

Problems and Solutions

- Problem:
  the `factorial` semantics above *is not stack safe*

Problems and Solutions

- Problem:
  the `factorial` semantics above *is not stack safe*
- Solution:
  `FreeTransformation` and `FreeTransformedMeaning`

Problems and Solutions

- Problem:
  the `factorial` semantics above *is not stack safe*
- Solution:
  `FreeTransformation` and `FreeTransformedMeaning`

- Problem:
  `producer` and `consumer` above *execute effects*

Problems and Solutions

- Problem:
  the `factorial` semantics above *is not stack safe*
- Solution:
  `FreeTransformation` and `FreeTransformedMeaning`

- Problem:
  `producer` and `consumer` above *execute effects*
- Solution:
  `read` and `write` *describe effects*

ComputationTransformation

```
private[pdbp] trait ComputationTransformation[
    FC[+ _]: Computation,
    T[+ _]]
    extends Computation[T]
    with Program[Kleisli[T]] {

  private[pdbp] val transform: FC `-U->` T

}
```


ComputationTransformation

```
private[pdbp] trait ComputationTransformation[
    FC[+ _]: Computation,
    T[+ _]]
    extends Computation[T]
    with Program[Kleisli[T]] {

  private[pdbp] val transform: FC `-U->` T

}
```

- *computation transformations* transform
  computations FC (and corresponding *kleisli programs*) to
  computations T (and corresponding *kleisli programs*)

ProgramMeaning

```
trait ProgramMeaning[
  `>-FP->`[- _, + _]: Program,
  `>-T->`[- _, + _]] {

  private[pdbp] lazy val binaryTransformation:
      `>-FP->` `-B->` `>-T->`

  lazy val meaning: `>-FP->` `-B->` `>-T->` =
    binaryTransformation

}
```


ComputationMeaning

```
private[pdbp] trait ComputationMeaning[
  FC[+ _]: Computation, T[+ _]]
    extends ProgramMeaning[Kleisli[FC], Kleisli[T]] {

  private[pdbp] val unaryTransformation: FC `-U->` T

  // ...

}
```


ComputationMeaning

```
private[pdbp] trait ComputationMeaning[
  FC[+ _]: Computation, T[+ _]]
    extends ProgramMeaning[Kleisli[FC], Kleisli[T]] {

  private[pdbp] val unaryTransformation: FC `-U->` T

  // ...

}
```

- *computation meanings* are
  program meanings for corresponding kleisli programs

FreeTransformed

```
sealed trait Free[C[+ _], +Z]

final case class Transform[C[+ _], +Z]
  (cz: C[Z]) extends Free[C, Z]
final case class Result[C[+ _], +Z]
  (z: Z) extends Free[C, Z]
final case class Bind[C[+ _], -Z, ZZ <: Z, +Y]
  (fczz: Free[C, ZZ], ‘z=>fcy‘: Z => Free[C, Y])
    extends Free[C, Y]

type FreeTransformed[C[+ _]] = [+Z] => Free[C, Z]
```

FreeTransformation

```
private[pdbp]
  trait FreeTransformation[FC[+ _]: Computation]
      extends ComputationTransformation[
        FC,
        FreeTransformed[FC]] {

  // unfold


  // transform => Transform

  // result => Result
  // bind => Bind

}
```


activeFreeTypes

```
object activeFreeTypes {

  type ActiveFree = FreeTransformed[Active]

  type ‘=>AF‘ = Kleisli[ActiveFree]

}
```

activeFreeProgram

```
import ... implicits.activeProgram

implicit object activeFreeProgram
    extends Computation[ActiveFree]
    with Program[`=>AF`]

    with FreeTransformation[Active]()
    with ComputationTransformation[Active, ActiveFree]()
```

FreeTransformedMeaning

```
private[pdbp] trait FreeTransformedMeaning[
  FC[+ _]: Computation, T[+ _]](
  implicit toBeTransformedMeaning:
    ComputationMeaning[FC, T])
    extends ComputationMeaning[FreeTransformed[FC], T] {

  // fold (tail recursive)


  // Transform(fcz) => fcz

  // Result(z) => result(z)
  // Bind(Result(y), y2ftfcz) => fold(y2ftfcz(y))
  // Bind(Bind(fcx, x2ftfcy), y2ftfcz) =>
  //   fold(Bind(fcx, Bind(x2ftfcy, y2ftfcz)))

}
```


activeMeaningOfActiveFree

```
import ... activeMeaningOfActive

implicit object activeMeaningOfActiveFree
    extends FreeTransformedMeaning[Active, Active]()
    with ComputationMeaning[ActiveFree, Active]()

    with ProgramMeaning[`=>AF`, `=>A`]()
```

FactorialMain

```
import ... activeFreeTypes._

import ... implicits.activeFreeProgram

import ... MainFactorial
```


FactorialMain

```
object FactorialMain extends MainFactorial[`=>AF`]() {

  // ... effectful utilities

  override val producer = effectfulReadIntFromConsole

  override val consumer = effectfulWriteFactorialOfIntToConsole

}
```

FactorialMain

```
object FactorialMain extends MainFactorial[`=>AF`]() {

  def main(args: Array[String]): Unit = {

    import ... activeMeaningOfActiveFree.meaning

    meaning(mainFactorial)(())

  }

}
```


Reading

```
trait Reading[R, >-->[- _, + _]] {

  private[pdbp] def `u>-->r`: Unit >--> R




}
```

Reading

```
trait Reading[R, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] def ‘u>-->r‘: Unit >--> R =
    read[Unit]




  def read[Z]: Z >--> R

}
```


Reading

```
trait Reading[R, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] def `u>-->r`: Unit >--> R =
    read[Unit]

  private[pdbp] def `z>-->r`[Z]: Z >--> R =
    compose(`z>-->u`, `u>-->r`)

  def read[Z]: Z >--> R = `z>-->r`

}
```


implicit function type

```
type `I=>`[-X, +Y] = implicit X => Y
```

implicit function type

```
type `I=>`[-X, +Y] = implicit X => Y
```

- *reduces boilerplate* of *reading* code

implicit function type

```
type `I=>`[-X, +Y] = implicit X => Y
```

- *reduces boilerplate* of *reading* code
- *optimizes performance* of *reading* code

ReadingTransformed

```
type ReadingTransformed[R, C[+ _]] =
  [+Z] => R `I=>` C[Z]
```


ReadingTransformation

```
private[pdbp] trait ReadingTransformation[
  R, FC[+ _]: Computation]
    extends ComputationTransformation[
      FC,
      ReadingTransformed[R, FC]]

    with Reading[
      R,
      Kleisli[ReadingTransformed[R, FC]]] {

  // ...

}
```


activeReadingTypes

```
object activeReadingTypes {

  type ActiveReading[R] = ReadingTransformed[R, Active]

  type `=>AR`[R] = Kleisli[ActiveReading[R]]

}
```


ActiveReadingProgram

```
private[pdbp] trait ActiveReadingProgram[R]
    extends Computation[ActiveReading[R]]
    with Program[`=>AR`[R]]
    with Reading[R, `=>AR`[R]]

    with ComputationTransformation[Active, ActiveReading[R]]
    with ReadingTransformation[R, Active]
```


activeIntReadingProgram

```
import  ... implicits.activeProgram

implicit object activeIntReadingProgram
    extends ActiveReadingProgram[BigInt]()
    with Computation[ActiveReading[BigInt]]()
    with Program[`=>AR`[BigInt]]()
    with Reading[BigInt, `=>AR`[BigInt]]()

    with ReadingTransformation[BigInt, Active]()
    with ComputationTransformation[Active, ActiveReading[BigInt]]()
```

ReadingTransformedMeaning

```
private[pdbp] trait ReadingTransformedMeaning[
      R,
      FC[+ _]: Computation,
      T[+ _]](
    implicit toBeTransformedMeaning:
      ComputationMeaning[FC, T])
    extends ComputationMeaning[
        ReadingTransformed[R, FC],
        ReadingTransformed[R, T]]
```

activeIntReadingMeaningOfActiveIntReading

```
import ... activeIntReadingProgram

implicit object activeIntReadingMeaningOfActiveIntReading
    extends ReadingTransformedMeaning[
      BigInt, Active, Active]()
    with ComputationMeaning[
      ActiveReading[BigInt],
      ActiveReading[BigInt]]()

    with ProgramMeaning[
      `=>AR`[BigInt],
      `=>AR`[BigInt]]()
```

FactorialOfIntReadMain

```
import ... activeReadingTypes._

import ... implicits.activeIntReadingProgram

import ... MainFactorial
```

readIntFromConsoleEffect

```
private def readIntFromConsoleEffectWithMessage(
  message: String): BigInt =
  effectfulReadIntFromConsoleFunction(message)(())

implicit val readIntFromConsoleEffect: BigInt =
  readIntFromConsoleEffectWithMessage(
    "please type an integer to read")
```

FactorialOfIntReadMain

```
object FactorialOfIntReadMain
    extends MainFactorial[`=>AR`[BigInt]]() {

  // ... effectful utilities

  import ... readIntFromConsoleEffect

  override val producer =
    activeIntReadingProgram.read

  override val consumer =
    effectfulWriteFactorialOfIntReadToConsole

}
```


FactorialOfIntReadMain

```
object FactorialOfIntReadMain
    extends MainFactorial[`=>AR`[BigInt]]() {

  def main(args: Array[String]): Unit = {

    ... activeIntReadingMeaningOfActiveIntReading.meaning

    meaning(mainFactorial)(())

  }

}
```


factorialMultipliedByIntRead

```
val factorialMultipliedByIntRead: BigInt >--> BigInt =
    (factorial & read) >--> multiply
```

MainFactorialMultipliedByIntRead

```
trait MainFactorialMultipliedByIntRead[
    >-->[- _, + _]: Program
                  : [>-->[- _, + _]] => Reading[BigInt, >-->]] {

  // ...

  lazy val mainFactorialMultipliedByIntRead: Unit >--> Unit =
    producer >-->
      factorialMultipliedByIntRead >-->
      consumer

}
```


FactorialMultipliedByIntReadMain

```
object FactorialMultipliedByIntReadMain
    extends MainFactorialMultipliedByIntRead[`=>AR`[BigInt]]() {

  // effectful utilities

  override val producer =
    effectfulReadIntFromConsole

  override val consumer =
    effectfulWriteFactorialOfIntMultipliedByIntReadToConsole

}
```


FactorialMultipliedByIntReadMain

```
object FactorialMultipliedByIntReadMain
    extends MainFactorialMultipliedByIntRead[‘=>AR‘[BigInt]]() {

  def main(args: Array[String]): Unit = {

    import ... readIntFromConsoleEffect

    import ... activeIntReadingMeaningOfActiveIntReading.meaning

    meaning(mainFactorialMultipliedByIntRead)(())

  }

}
```

Writing

```
trait Writing[W: Writable, >-->[- _, + _]] {

  private[pdbp] val ‘w>-->u‘: W >--> Unit

}
```


Writing

```
trait Writing[W: Writable, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] val `w>-->u`: W >--> Unit =
    write(identity)




  def write[Z]:                   Z >--> Unit


}
```

Writing

```
trait Writing[W: Writable, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] val ‘w>-->u‘: W >--> Unit =
    write(identity)




  def write[Z]: (Z => W) ‘I=>‘ Z >--> Unit


}
```

Writing

```
trait Writing[W: Writable, >-->[- _, + _]] {
  this: Function[>-->] & Composition[>-->] =>

  private[pdbp] val ‘w>-->u‘: W >--> Unit =
    write(identity)

  private[pdbp] def ‘z>-w->u‘[Z]:
      (Z => W) ‘I=>‘ Z >--> Unit =
    compose(function(implicitly), ‘w>-->u‘)

  def write[Z]: (Z => W) ‘I=>‘ Z >--> Unit =
    ‘z>-w->u‘

}
```


Writing

```
def writingUsing[Z, Y,  ](
                              ):
    (Z >--> Y) => (                Z >--> Y)
```

Writing

```
def writingUsing[Z, Y, X](
    `(z&&y)=>x`: ((Z && Y) => X)):
    (Z >--> Y) => (               Z >--> Y)
```

Writing

```
def writingUsing[Z, Y, X](
    `(z&&y)=>x`: ((Z && Y) => X)):
    (Z >--> Y) => ((X => W) `I=>` Z >--> Y)
```

Writing

```
def writingUsing[Z, Y, X](
    `(z&&y)=>x`: ((Z && Y) => X)):
    (Z >--> Y) => ((X => W) `I=>` Z >--> Y) = { `z>-->y` =>
    val `(z&&y)>-->x` = function(`(z&&y)=>x`)
    val `z>-->(x&&y)` =
      `let` {
        `z>-->y`
      } `in` {
        `let` {
          `(z&&y)>-->x`
        } `in` {
          `(z&&y&&x)>-->(x&&y)`
        }
      }
    compose(compose(`z>-->(x&&y)`, left(write),
      `(u&&y)>-->y`)
}
```

Writable (cfr. *monoid*)

```
private[pdbp] trait Writable[W]
    extends Startable[W]
    with Appendable[W]
```

Startable

```
private[pdbp] trait Startable[W] {

  private[pdbp] val start: W

}
```

Appendable

```
private[pdbp] trait Appendable[W] {

  private[pdbp] val append: W && W => W

}
```

WritingTransformed

```
type WritingTransformed[W, FC[+ _]] =
  [+Z] => FC[W && Z]
```

WritingTransformation

```
private[pdbp] trait WritingTransformation[
  W: Writable, FC[+ _]: Computation]
    extends ComputationTransformation[
      FC, WritingTransformed[W, FC]]

    with Writing[
      W,
      Kleisli[WritingTransformed[W, FC]]] {

  // ...

}
```

ToConsole

```
case class ToConsole(effect: Effect)

type Effect = Unit => Unit
```

toConsoleWritable

```
implicit object toConsoleWritable extends Writable[ToConsole] {

  override private[pdbp] val start: ToConsole =
    ToConsole { _ =>
      ()
    }

  override private[pdbp] val append:
      ToConsole && ToConsole => ToConsole = {
    (tc1, tc2) =>
      ToConsole { _ =>
        { tc1.effect(()); tc2.effect(()) }
      }
  }

}
```


WritingToConsoleTransformedMeaning

```
private[pdbp] trait WritingToConsoleTransformedMeaning[
    FC[+ _]: Computation,
    T[+ _]]
    (implicit toBeTransformedMeaning:
      ComputationMeaning[FC, T])
    extends ComputationMeaning[
      WritingTransformed[ToConsole, FC],
        T] {

  // executes console effect

}
```


infoUtils

```
def infoFunction[Z, Y](string: String): Z && Y => String = {
  case (z, y) =>
    s"INFO -- $currentCalendarInMilliseconds -- $string($z) => $y"
}

def info[
    W: Writable, Z, Y,
    >-->[- _, + _]: [>-->[- _, + _]] => Writing[W, >-->]]
    (string: String):
    (Z >--> Y) => ((String => W) `I=>` Z >--> Y) = {
  val implicitWriting = implicitly[Writing[W, >-->]]
  implicitWriting.writingUsing(infoFunction(string))
}
```

WritingAtomicPrograms

```
val isZero: (String => W) `I=>` BigInt >--> Boolean =
  info("isZero") { isZeroHelper }

val subtractOne: (String => W) `I=>` BigInt >--> BigInt =
  info("subtractOne") { subtractOneHelper }

val multiply: (String => W) `I=>` (BigInt && BigInt) >--> BigInt =
  info("multiply") { multiplyHelper }

def one[Z]: (String => W) `I=>` Z >--> BigInt =
  info("one") { oneHelper }
```

WritingFactorial

```
val factorial: (String => W) `I=>` BigInt >--> BigInt =
  info("factorial") {
    `if`(isZero) {
      one
    } `else` {
      `let` {
        subtractOne >-->
          factorial
      } `in` {
        multiply
      }
    }
  }
```


MainWritingFactorial

```
trait MainWritingFactorial[
    W: Writable,
    >-->[- _, + _]: Program
                  : [>-->[- _, + _]] => Writing[W, >-->]] {

  // ...

  lazy val mainWritingFactorial:
      (String => W) `I=>` Unit >--> Unit = {
    producer >-->
      writingFactorial >-->
      consumer
  }

}
```


activeWritingToConsoleProgram

```
import ... toConsoleWritable
import ... activeProgram

implicit object activeWritingToConsoleProgram
    extends ActiveWritingProgram[ToConsole]()
    with Computation[ActiveWriting[ToConsole]]()
    with Program[`=>AW`[ToConsole]]()
    with Writing[ToConsole, `=>AW`[ToConsole]]()

    with ComputationTransformation[
      Active, ActiveWriting[ToConsole]]()
    with WritingTransformation[ToConsole, Active]()
```


activeIntReadingWithWritingToConsoleProgram

```
import ... toConsoleWritable
import ... activeWritingToConsoleProgram

implicit object activeIntReadingWithWritingToConsoleProgram
    extends ActiveReadingWithWritingProgram[BigInt, ToConsole]()
    with Computation[ActiveReadingWithWriting[BigInt, ToConsole]]()
    with Program[‘=>ARW‘[BigInt, ToConsole]]()
    with Reading[BigInt, ‘=>ARW‘[BigInt, ToConsole]]()
    with Writing[ToConsole, ‘=>ARW‘[BigInt, ToConsole]]()

    // ...
```

activeIntReadingMeaning OfActiveIntReadingWithWritingToConsole

```
import ... activeMeaningOfActiveWritingToConsole

implicit object
  activeIntReadingMeaningOfActiveIntReadingWithWritingToConsole
    extends ReadingTransformedMeaning[
      BigInt, ActiveWriting[ToConsole], Active]()
    with ComputationMeaning[
      ActiveReadingWithWriting[BigInt, ToConsole],
      ActiveReading[BigInt]]()
    with ProgramMeaning[
      `=>ARW`[BigInt, ToConsole],
      `=>AR`[BigInt]]()
```


FactorialOfIntRead WritingToConsoleWrittenToConsoleMain

```
import ... activeReadingWithWritingTypes._

import ... implicits.toConsoleWritable

import ... implicits.activeIntReadingWithWritingToConsoleProgram

import ... MainWritingFactorial
```


FactorialOfIntRead WritingToConsoleWrittenToConsoleMain

```
object FactorialOfIntReadWritingToConsoleWrittenToConsoleMain
    extends MainWritingFactorial[
      ToConsole,
      `=>ARW`[BigInt, ToConsole]]() {

  import ... readIntFromConsoleEffect
  import ... writeFactorialOfIntToConsoleEffect
  import ... writeToConsoleEffect

  override val producer =
    activeIntReadingWithWritingToConsoleProgram.read

  override val consumer =
    activeIntReadingWithWritingToConsoleProgram.write

}
```

FactorialOfIntRead WritingToConsoleWrittenToConsoleMain

```
object FactorialOfIntReadWritingToConsoleWrittenToConsoleMain
    extends MainWritingFactorial[ToConsole, `=>ARW`[BigInt, ToConsole]]() {

  def main(args: Array[String]): Unit = {

    import ... implicits.
      activeIntReadingMeaningOfActiveIntReadingWithWritingToConsole.meaning

    meaning(mainWritingFactorial)(())

  }

}
```


running

```
please type an integer to read
2
INFO -- 2018-08-01 18:00:42.639 -- isZero(2) => false
INFO -- 2018-08-01 18:00:42.645 -- subtractOne(2) => 1
INFO -- 2018-08-01 18:00:42.646 -- isZero(1) => false
INFO -- 2018-08-01 18:00:42.647 -- subtractOne(1) => 0
INFO -- 2018-08-01 18:00:42.647 -- isZero(0) => true
INFO -- 2018-08-01 18:00:42.648 -- one(0) => 1
INFO -- 2018-08-01 18:00:42.648 -- factorial(0) => 1
INFO -- 2018-08-01 18:00:42.649 -- multiply((1,1)) => 1
INFO -- 2018-08-01 18:00:42.649 -- factorial(1) => 1
INFO -- 2018-08-01 18:00:42.650 -- multiply((2,1)) => 2
INFO -- 2018-08-01 18:00:42.650 -- factorial(2) => 2
the factorial value of the integer read is
2
```