P{a}thodental Complexity

Alexander Ioffe
@deusaquilus

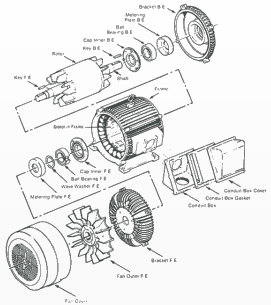
This Talk: https://github.com/deusaquilus/fs2023

Essential Complexity

[you-**can**-*n*ot m*a*k*e*-**it**-be-ter]

noun,boring

1. Programmatic complexity inherent to the business processes that it represents which cannot be simplified.

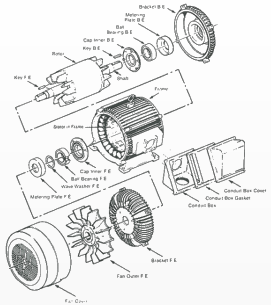


Essential Complexity

[you-**can**-*n*ot m*a*k*e*-**it**-be-ter]

noun,boring

1. Programmatic complexity inherent to the business processes that it represents which cannot be simplified.

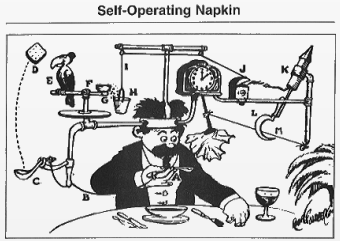


Accidental Complexity

[you-*did* som-**ting**-s*tu*-pid]

noun,vulgar

1. Programmatic complexity that occurs as a result of the way you chose to solve the problem be it an incorrect choice of approach, technology, etc… that could have been avoided.

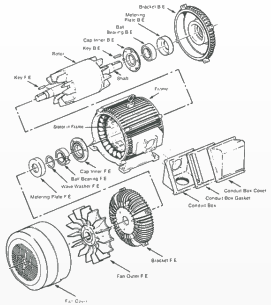


Essential Complexity

[you-**can**-*n*ot m*ake*-**it**-be-ter]

noun,boring

1. Programmatic complexity inherent to the business processes that it represents which cannot be simplified.

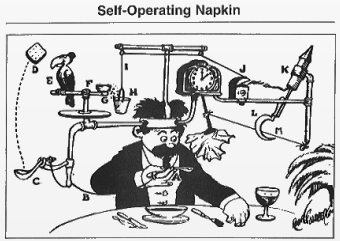


Accidental Complexity

[you-*did* som-**ting**-s*tu*-pid]

noun,vulgar

1. Programmatic complexity that occurs as a result of the way you chose to solve the problem be it an incorrect choice of approach, technology, etc… that could have been avoided.



Pathodental Complexity

[you-re-***bs***-ing your-*self*]

noun,uniquely-maladaptive

1. Complexity that is accidental but you have convinced yourself that it is essential or complexity that is essential but you have convinced yourself that it is accidental, by lying to yourself through your teeth.

The Great **Pathodental Lies** of FP

The Less Boilerplate the Better 01

Functional Programming is Simple 02


Story
Time

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

```
r = (x + y)/z
```

Basic math is representable in any language.

Extracting Business Logic **from `programming language`**

Saves **lots** of freaking money

01

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

```
r = (x + y)/z
```

Basic math is
representable
in any language.

```
def lcr(toCounterparty, date, reserveRequirement) =
  val hqla =
    hqlaAmount(
      lookupProduct(productAt(date)),
      reserveBalanceRequirement
    )
  val totalNetCashOutflow =
    totalNetCashOutflowAmount(toCounterparty, date)

  hqla / totalNetCashOutflow
```

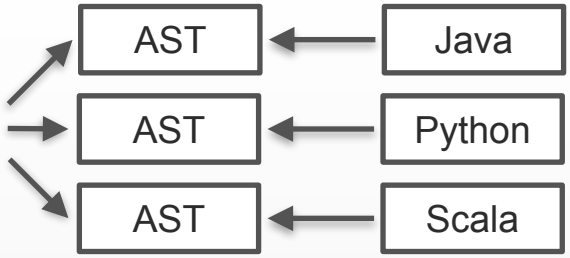
Why not
business logic?

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

`r = (x + y)/z`




```
def lcr(toCounterparty, date, reserveRequirement) =
  val hqla =
    hqlaAmount(
      lookupProduct(productAt(date)),
      reserveBalanceRequirement
    )
  val totalNetCashOutflow =
    totalNetCashOutflowAmount(toCounterparty, date)

  hqla / totalNetCashOutflow
```

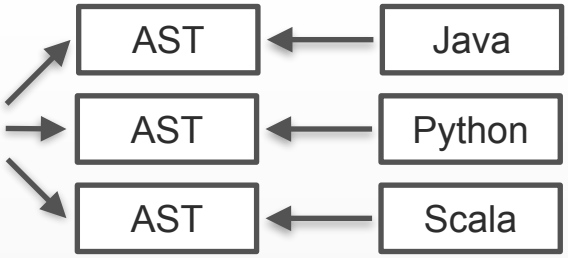
Why not
business logic?

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

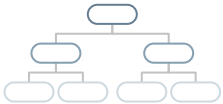
01

```
r = (x + y)/z
```




```
def lcr(toCounterparty, date, reserveRequirement) =
  val hqla =
    hqlaAmount(
      lookupProduct(productAt(date)),
      reserveBalanceRequirement
    )
  val totalNetCashOutflow =
    totalNetCashOutflowAmount(toCounterparty, date)

  hqla / totalNetCashOutflow
```

Language
Specific
Syntax Tree



Macros /
Compiler Magic

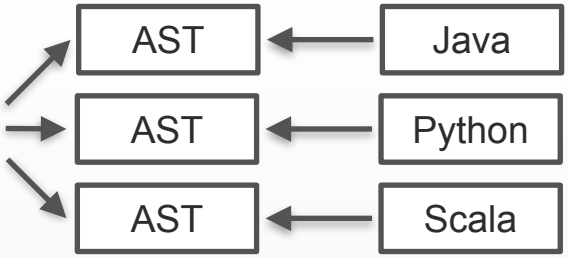
Business Logic
Syntax Tree
(Language Independent)

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

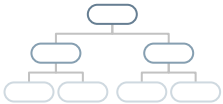
01

```
r = (x + y)/z
```




```
def lcr(toCounterparty, date, reserveRequirement) =
  val hqla =
    hqlaAmount(
      lookupProduct(productAt(date)),
      reserveBalanceRequirement
    )
  val totalNetCashOutflow =
    totalNetCashOutflowAmount(toCounterparty, date)

  hqla / totalNetCashOutflow
```

Language
Specific
Syntax Tree

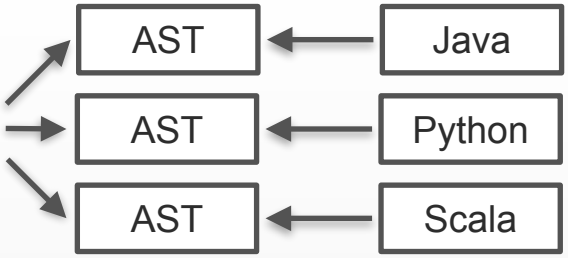


Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

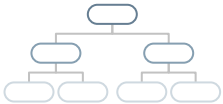
01

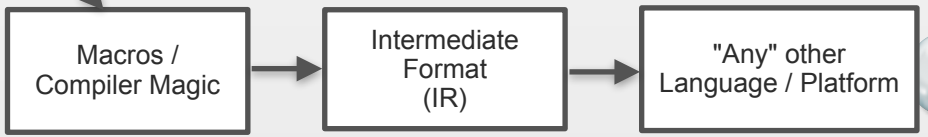
```
r = (x + y)/z
```




```
def lcr(toCounterparty, date, reserveRequirement) =
  val hqla =
    hqlaAmount(
      lookupProduct(productAt(date)),
      reserveBalanceRequirement
    )
  val totalNetCashOutflow =
    totalNetCashOutflowAmount(toCounterparty, date)

  hqla / totalNetCashOutflow
```

Language
Specific
Syntax Tree



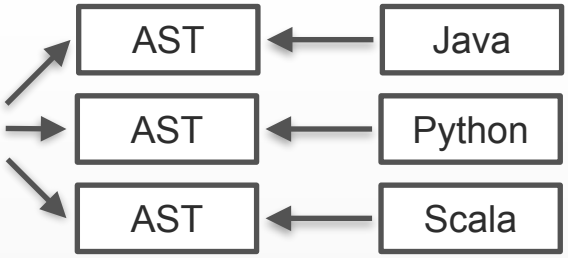


Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

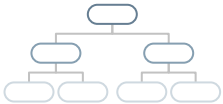
01

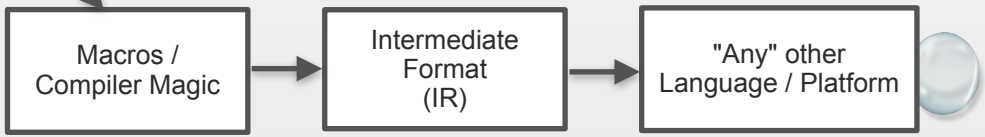
```
r = (x + y)/z
```



Language
Specific
Syntax Tree

What **features** should this
Business Logic / IR Format have???





IR

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

Use **OOP**?

02

IR

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

Use **OOP**?

02


```
public Map<String,Object> processSupplies(Supplier[] suppliers) {
  int total = 0;
  int max = -1;
  String maxSupplier = null;

  for(i = 0; i < supplier.length; i++) {
    Supplier supplier = suppliers[i];
    total += supplier.getQuantity();

    if(supplier.getQuantity() > max) {
      max = supplier.getQuantity();
      maxSupplier = supplier.supplierId;
    }
  }

  Map<String,Object> result = new HashMap<String,Object>();
  result.put("total", total);

  if(max > -1) {
    result.put("maxSupplier", maxSupplier);
  }

  return result;
}
```

Procedural styles
encode mechanism

https://morphir.finos.org/docs/functional-programming/

IR

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

Use **FP**!

02

ELM / F#

```
processSupplies suppliers =
  let
    total =
      suppliers
        |> List.map .quantity
        |> List.sum

    max =
      suppliers
        |> List.maximumBy .quantity
        |> List.map .supplierId

  in
    (total, max)
```

Procedural styles encode intent!

IR

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

Use **FP**!

02

Use Features
common in
`every FP language`

Procedural styles
encode intent!

IR

Extracting
Business Logic **from**
`programming language`
Saves **lots** of freaking money

01

Use Features
common in
`every FP language`
They're all actually pretty similar

02

morphir

The Fintech
Open Source
Foundation

IR

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

Use Features
common in
every FP language

They're all actually pretty similar

02

You will
need..

Enums

Some kind of
Coproduct!

```
type Customer
  = Person firstName:String lastName:String
  | Robot id:Int
  | Unknown
```

A.k.a.

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

Records

Some kind of
Product!

```
type alias Account =
  {
    accountNumber: AccountNumber,
    customerId: CustomerId,
    numOrders: Int
  }
```

A.k.a.

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

IR

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

Use Features
common in
`every FP language`

They're all actually pretty similar

02

You will
need..

Branching

Some kind of
branching!

```
if (a == b)
  x
else
  y
```

IS

```
Value.IfThenElse(
  Ref("==", a, b),
  x,
  y
)
```

Function Definitions

Some kind of
function def!

```
addOne: Int -> Int
addOne a = a + 1
```

IS

```
Value.Definition(
  "addOne",
  vars = Ref("a"),
  body = Ref("+", Ref("a"), Const(1))
)
```

IR

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

01

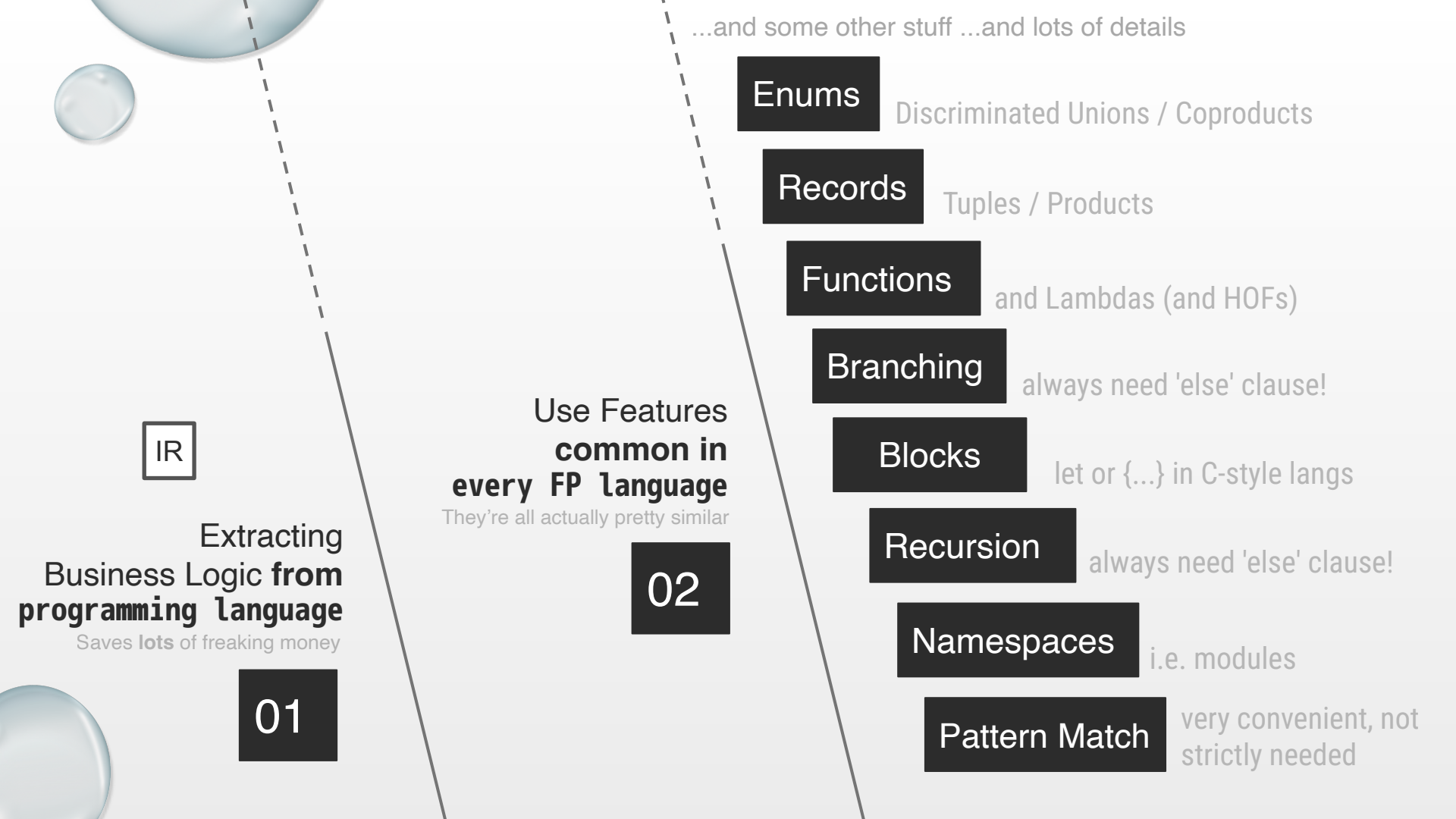
Use Features
common in
`every FP language`

They're all actually pretty similar

02

...and some other stuff

- Enums
- Records
- Functions
- Branching
- Blocks
- Recursion
- Namespaces
- Pattern Match


...and some other stuff ...and lots of details
Enums
Discriminated Unions / Coproducts
Records
Tuples / Products
Functions
and Lambdas (and HOFs)
Branching
always need 'else' clause!
Blocks
let or {...} in C-style langs
Recursion
always need 'else' clause!
Namespaces
i.e. modules
Pattern Match
very convenient, not strictly needed
Use Features
common in
every FP language
They're all actually pretty similar
02
IR
Extracting
Business Logic from
programming language
Saves lots of freaking money
01

IR

Extracting
Business Logic **from**
`programming language`

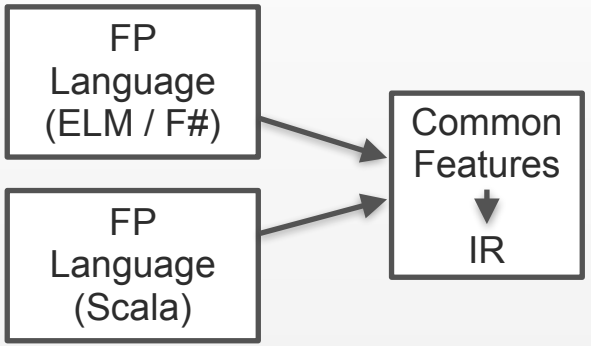
Saves **lots** of freaking money

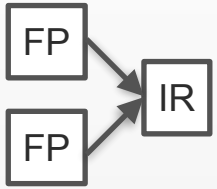
01

Use Features
common in
`every FP language`

They're all actually pretty similar

02





Use Features
common in
`every FP language`

They're all actually pretty similar

02

IR

Extracting
Business Logic **from**
`programming language`

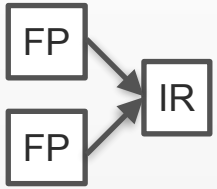
Saves **lots** of freaking money

01

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

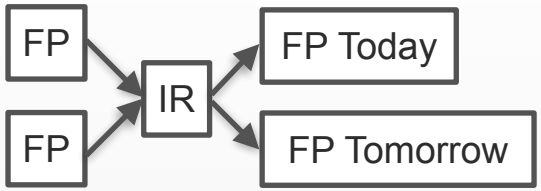
01



Use Features
common in
`every FP language`

They're all actually pretty similar

02



Write in one language
`transpile`
`to every other`

Hard but not impossible

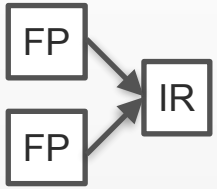
03

IR

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

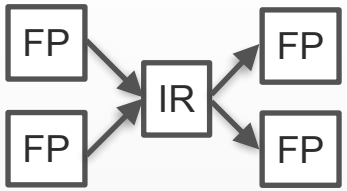
01



Use Features
common in
`every FP language`

They're all actually pretty similar

02



Write in one language
`transpile`
`to every other`

Hard but not impossible

03

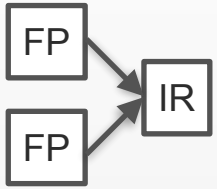
Model Data
to match
`intermediate`
`format`

04

Extracting
Business Logic **from**
`programming language`

Saves **lots** of freaking money

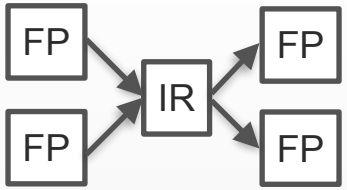
01



Use Features
common in
`every FP language`

They're all actually pretty similar

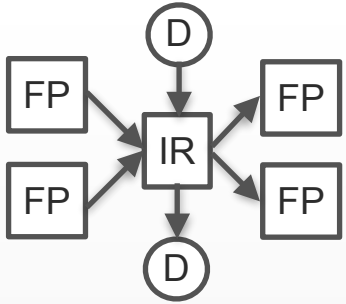
02



Write in one language
`transpile`
`to every other`

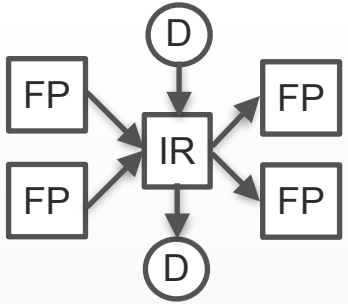
Hard but not impossible

03



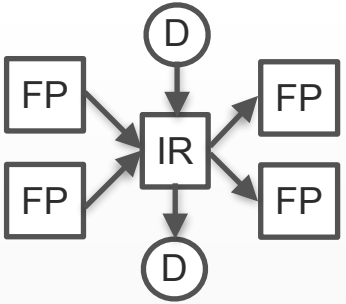
Model Data
to match
`intermediate`
`format`

04

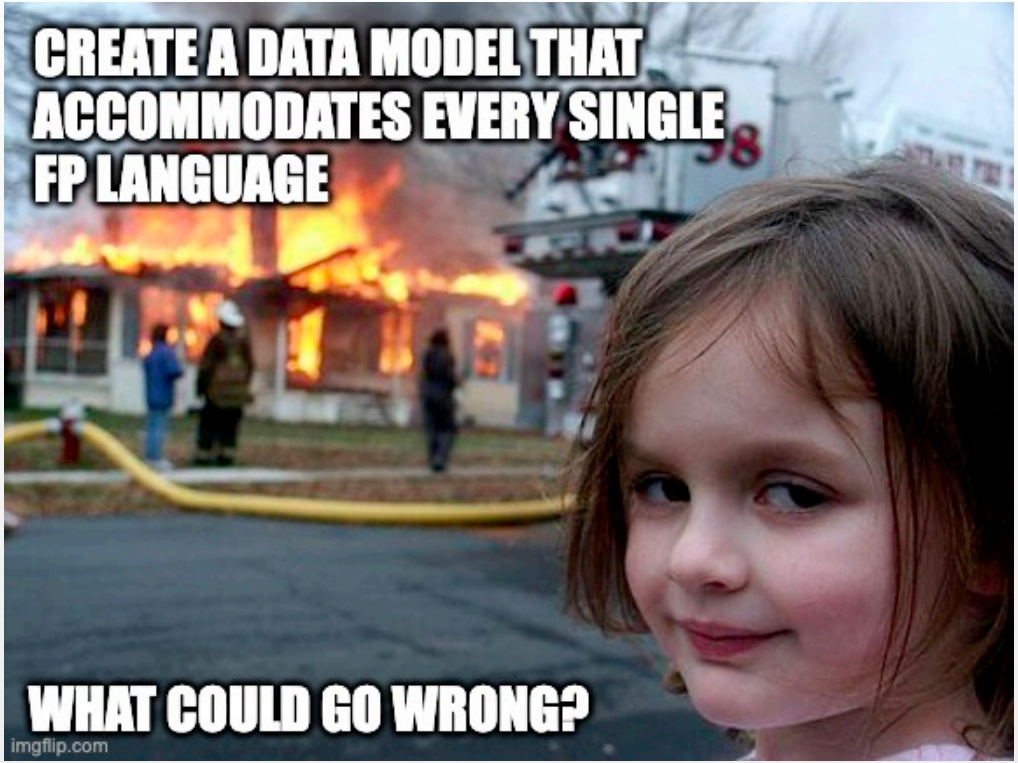


Model Data
to match
`intermediate`
`format`

04



Model Data
to match
`intermediate`
`format`



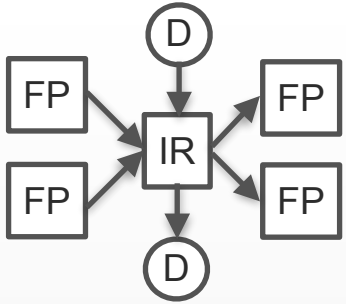
Basic Types

```
id : Int
id = 123
```

```
val x = 123
```

```
Value.Definition(
  "id",
  Const(Data.Int(123))
)
```

```
object Data {
  case class Int(value: scala.Int)
  case class Short(value: scala.Short)
  case class Long(value: scala.Long)
  case class Boolean(value: scala.Boolean)
  case class String(value: java.lang.String)
  case class Char(value: scala.Char)
  case class Byte(value: scala.Byte)
  case class Float(value: scala.Float)

  case class Decimal(value: scala.BigDecimal)
  case class Integer(value: scala.BigInt)

  case class LocalDate(value: java.time.LocalDate)
  case class LocalTime(value: java.time.LocalTime)
  case class Month(value: java.time.Month)
}
```



Model Data
to match
intermediate
format

04

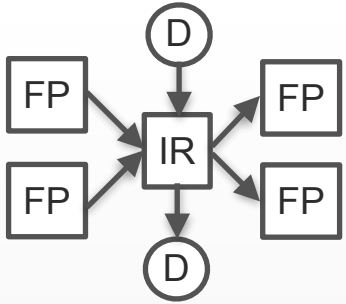
Basic Types

```
id : Int
id = 123
```

```
val x = 123
```

```
Value.Definition(
  "id",
  Const(Data.Int(123))
)
```

```
sealed trait Basic[+A]
object Data {
  case class Int(value: scala.Int) extends Basic[scala.Int]
  case class Short(value: scala.Short) extends Basic[scala.Short]
  case class Long(value: scala.Long) extends Basic[scala.Long]
  case class Boolean(value: scala.Boolean) extends Basic[scala.Boolean]
  case class String(value: java.lang.String) extends Basic[java.lang.String]
  case class Char(value: scala.Char) extends Basic[scala.Char]
  case class Byte(value: scala.Byte) extends Basic[scala.Byte]
  case class Float(value: scala.Float) extends Basic[scala.Float]
  ...
  case class Decimal(value: scala.BigDecimal) extends Basic[scala.BigDecimal]
  case class Integer(value: scala.BigInt) extends Basic[scala.BigInt]
  ...
  case class LocalDate(value: java.time.LocalDate) extends Basic[java.time.LocalDate]
  case class LocalTime(value: java.time.LocalTime) extends Basic[java.time.LocalTime]
  case class Month(value: java.time.Month) extends Basic[java.time.Month]
}
```



Model Data
to match
intermediate
format

04

Enums

```
type Customer
  = Person firstName:String lastName:String
  | Robot id:Int
  | Unknown
```

ELM / F#

Scala

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

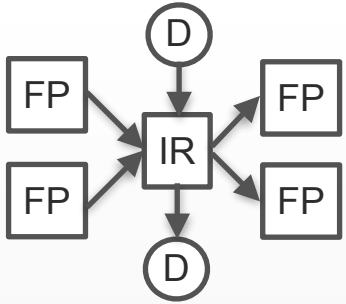
Records

```
type alias Account =
  {
    accountNumber: AccountNumber,
    customerId: CustomerId,
    numOrders: Int
  }
```

ELM / F#

Scala

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```



Model Data
to match
intermediate
format

04

Enums

```
type Customer
  = Person firstName:String lastName:String
  | Robot id:Int
  | Unknown
```

ELM / F#

Scala

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

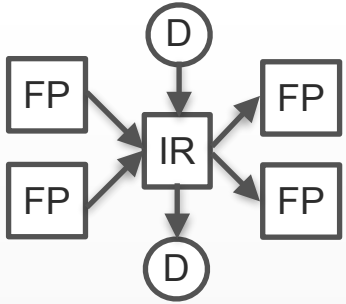
Records

```
type alias Account =
  {
    accountNumber: AccountNumber,
    customerId: CustomerId,
    numOrders: Int
  }
```

ELM / F#

Scala

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```



Model Data
to match
intermediate
format

04

Yes... it's actually portable!

Enums

```
type Customer
  = Person firstName:String lastName:String
  | Robot id:Int
  | Unknown
```

ELM / F#

Scala

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

```
person: Customer
person = Person "Joe" "Bloggs"
```

ELM / F#

Scala

```
val person =
  Customer.Person("Joe", "Bloggs")
```

Records

```
type alias Account =
  {
    accountNumber: AccountNumber,
    customerId: CustomerId,
    numOrders: Int
  }
```

ELM / F#

Scala

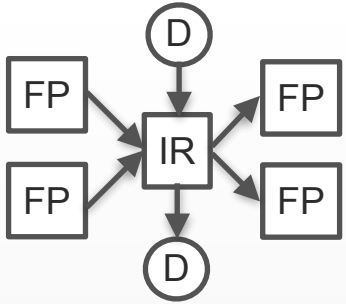
```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

```
acctNum = AccountNumber 123
custId = CustomerId 456
joeAccount = Account acctNum custId 550
```

ELM / F#

Scala

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)
```



Model Data
to match
intermediate
format

04

Yes... it's actually portable!

Enums

```
type Customer
  = Person firstName:String lastName:String
  | Robot id:Int
  | Unknown
```

ELM / F#

Scala

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

```
person: Customer
person = Person "Joe" "Bloggs"
```

ELM / F#

Scala

```
val person =
  Customer.Person("Joe", "Bloggs")
```

Records

```
type alias Account =
  {
    accountNumber: AccountNumber,
    customerId: CustomerId,
    numOrders: Int
  }
```

ELM / F#

Scala

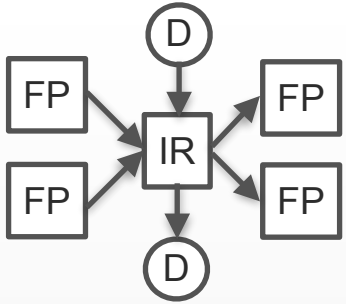
```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

```
acctNum = AccountNumber 123
custId = CustomerId 456
joeAccount = Account acctNum custId 550
```

ELM / F#

Scala

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)
```



Model Data
to match
intermediate
format

04

Yes... it's actually portable!

Enums

```
type Customer
  = Person firstName:String lastName:String
  | Robot id:Int
  | Unknown
```

ELM / F#

Scala

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

```
person: Customer
person = Person "Joe" "Bloggs"
```

ELM / F#

Scala

```
val person =
  Customer.Person("Joe", "Bloggs")
```

Records

```
type alias Account =
  {
    accountNumber: AccountNumber,
    customerId: CustomerId,
    numOrders: Int
  }
```

ELM / F#

Scala

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

```
acctNum = AccountNumber 123
custId = CustomerId 456
joeAccount = Account acctNum custId 550
```

ELM / F#

Scala

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)
```

Portable!... somewhat

Enums

```
Data.Enum(
  values = List(
   "firstName" -> Data.String("Joe"),
   "lastName"  -> Data.String("Bloggs")
  ),
  case = "Person"
)
```

```
person: Customer
person = Person "Joe" "Bloggs"
```

ELM / F#

Scala

```
val person =
  Customer.Person("Joe", "Bloggs")
```

Records

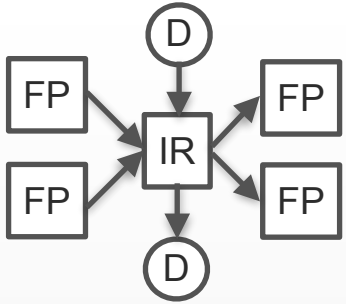
```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData,
    "customerId" -> custIdData,
    "numOrders" -> Data.Int32(550)
  )
)
```

```
acctNum = AccountNumber 123
custId = CustomerId 456
joeAccount = Account acctNum custId 550
```

ELM / F#

Scala

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)
```



Model Data **to match** intermediate format

04

Now how do we encode this data?

Enums

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

```
val person =
  Customer.Person("Joe", "Bloggs")
```

```
Data.Enum(
  values = List(
   "firstName" -> Data.String("Joe"),
   "lastName"  -> Data.String("Bloggs")
  ),
  case = "Person"
)
```

```
case class Enum(
    values: scala.List[(java.lang.String, Data)],
    enumLabel: java.lang.String,
    shape: Schema.Enum
) extends Data
```

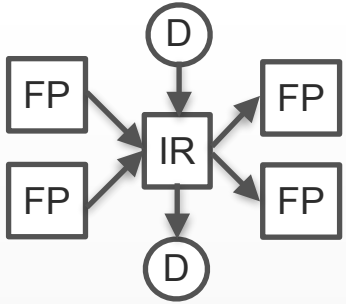
Records

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)
```

```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData,
    "customerId" -> custIdData,
    "numOrders" -> Data.Int32(550)
  )
)
```

```
case class Record(
  values: scala.List[(Label, Data)],
  shape: Schema.Record
) extends Data
```



Model Data
to match
intermediate
format

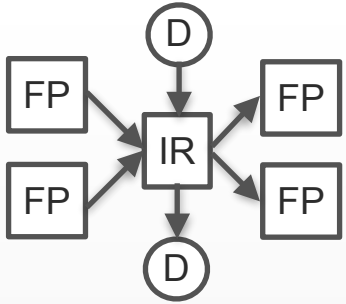
04

Enums

```
val person =
  Customer.Person("Joe", "Bloggs")
```

Records

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)
```



Model Data
to match
intermediate
format

04

Useful to track the schema!

```
Data.Enum(
  values = List(
   "firstName" -> Data.String("Joe"),
   "lastName"  -> Data.String("Bloggs")
  ),
  case = "Person",
  schema = Schema.Enum(
    "Customer",
    List(
     Enum("Person", List("firstName" -> String, "lastName" -> String)),
     Enum("Robot", List("id" -> Int32)),
     Enum("Unknown", List())
    )
  )
)
```

```
case class Enum(
    values: scala.List[(java.lang.String, Data)],
    enumLabel: java.lang.String,
    shape: Schema.Enum
) extends Data
```

```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData,
    "customerId" -> custIdData,
    "numOrders" -> Data.Int32(550)
  ),
  schema = Schema.Record(
    "Account",
    List(
      "accountNumber" -> acctNumData.schema,
      "customerId" -> custIdData.schema,
      "numOrders" -> Int32
    )
  )
)
```

```
case class Record(
  values: scala.List[(Label, Data)],
  shape: Schema.Record
) extends Data
```

Enums

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

```
val person =
  Customer.Person("Joe", "Bloggs")

val data: Data.Enum =
  DataBuilder.gen[Customer].build(person)
```

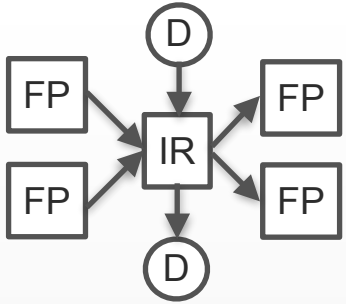
```
Data.Enum(
  values = List(
   "firstName" -> Data.String("Joe"),
   "lastName"  -> Data.String("Bloggs")
  ),
  case = "Person",
  schema = ...
)
```

Records

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)

val data: Data.Record =
  DataBuilder.gen[Account].build(joeAccount)
```

```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData,
    "customerId" -> custIdData,
    "numOrders" -> Data.Int32(550)
  ),
  schema = ...
)
```



Model Data
to match
intermediate
format

04

DataBuilder figures out Schema from type level!

Enums

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

```
val person =
  Customer.Person("Joe", "Bloggs")

val data: Data.Enum =
  DataBuilder.gen[Customer].build(person)
```

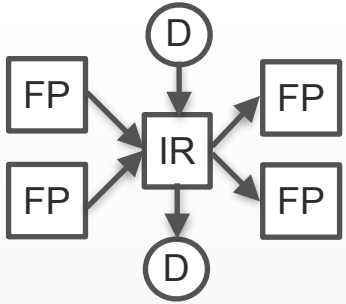
```
Data.Enum(
  values = List(
   "firstName" -> Data.String("Joe"),
   "lastName"  -> Data.String("Bloggs")
  ),
  case = "Person",
  schema = ...
)
```

Records

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

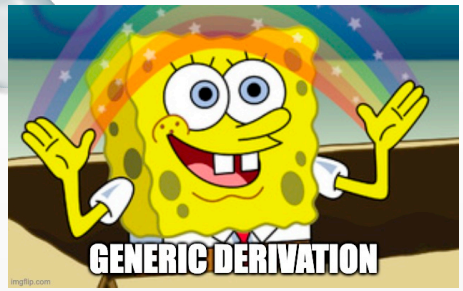
```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)

val data: Data.Record =
  DataBuilder.gen[Account].build(joeAccount)
```

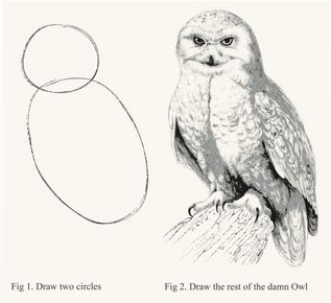
```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData,
    "customerId" -> custIdData,
    "numOrders" -> Data.Int32(550)
  ),
  schema = ...
)
```

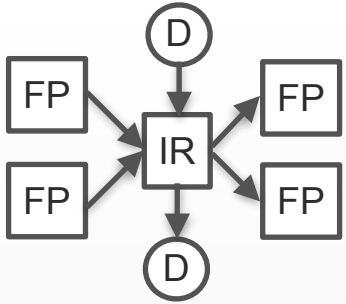


Model Data
to match
intermediate
format

04







Model Data
to match
intermediate
format

04

```
val person =
  Customer.Person("Joe", "Bloggs")

val data: Data.Enum =
  DataBuilder.gen[Customer].build(person)
```

```
Data.Enum(
  values = List(
   "firstName" -> Data.String("Joe"),
   "lastName"  -> Data.String("Bloggs")
  ),
  case = "Person",
  schema = ...
)
```

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)
val data: Data.Record =
  DataBuilder.gen[Account].build(joeAccount)
```

```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData,
    "customerId" -> custIdData,
    "numOrders" -> Data.Int32(550)
  ),
  schema = ...
)
```

Enums

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```

```
val person =
  Customer.Person("Joe", "Bloggs")
```

```
val data: Data.Enum =
  DataBuilder.gen[Customer].build(person)
```

```
Data.Enum(
  values = List(
   "firstName" -> Data.String("Joe"),
   "lastName"  -> Data.String("Bloggs")
  ),
  case = "Person",
  schema = ...
)
```

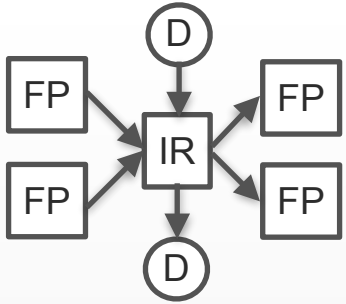
Records

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)
```

```
val data: Data.Record =
  DataBuilder.gen[Account].build(joeAccount)
```

```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData:Data.What???,
    "customerId" -> custIdData:Data.What???,
    "numOrders" -> Data.Int32(550)
  ),
  schema = ...
)
```



Model Data
to match
intermediate
format

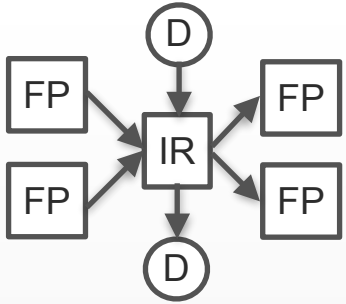
04

Records

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)

val data: Data.Record =
  DataBuilder.gen[Account].build(joeAccount)
```

```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData:Data.What???,
    "customerId" -> custIdData:Data.What???,
    "numOrders" -> Data.Int32(550)
  ),
  schema = ...
)
```



Model Data
to match
intermediate
format

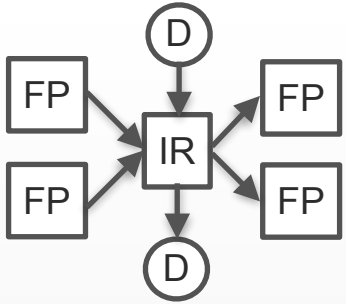
04

Records

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

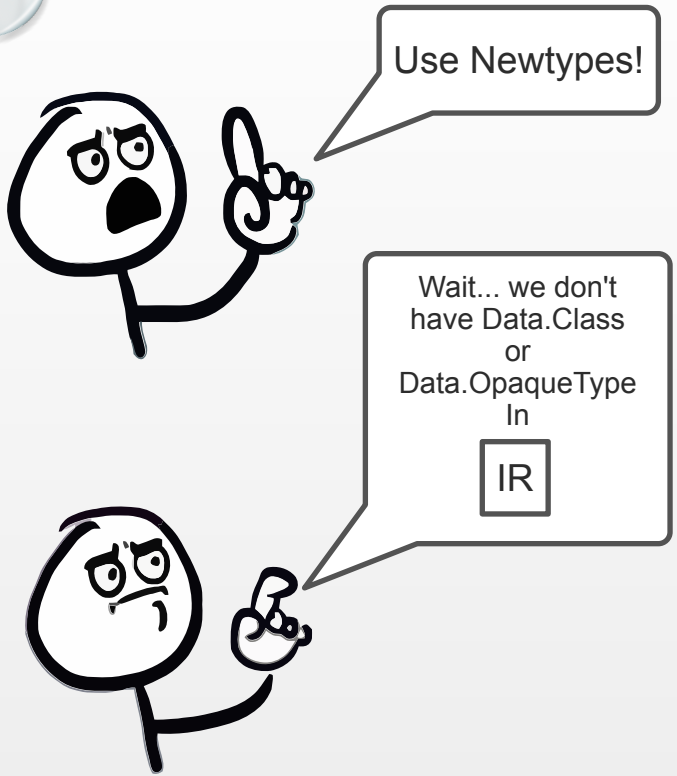
```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)

val data: Data.Record =
  DataBuilder.gen[Account].build(joeAccount)
```

```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData:Data.Int(123),
    "customerId" -> custIdData:Data.Int(456),
    "numOrders" -> Data.Int32(550)
  ),
  schema = ...
)
```



Model Data
to match
intermediate
format

04

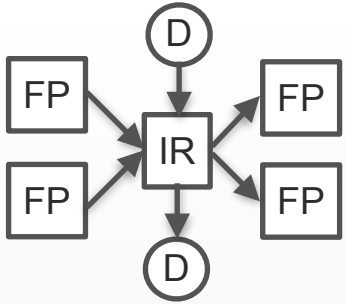


Records

```
case class Account(
  accountNumber: AccountNumber,
  customerId: CustomerId,
  numOrders: Int
)
```

```
val acctNum = AccountNumber(123)
val custId = CustomerId(456)
val joeAccount = Account(acctNum, custId, 550)

val data: Data.Record =
  DataBuilder.gen[Account].build(joeAccount)
```

```
Data.Record(
  values = List(
    "accountNumber" -> acctNumData:Data.What???,
    "customerId" -> custIdData:Data.What???,
    "numOrders" -> Data.Int32(550)
  ),
  schema = ...
)
```



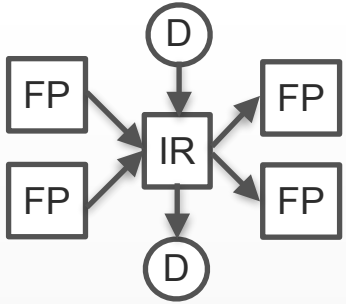
Model Data **to match** intermediate format

04

```
type Customer
  = Person firstName:String lastName:String
  | Robot id:Int
  | Unknown
```

IS

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```



Model Data
to match
intermediate
format

04

If types are new enums...

```
type AccountNumber
  = AccountNumber Int
```

IS

```
enum AccountNumber {
  case AccountNumber(value: Int)
}
```

```
type CustomerId
  = CustomerId Int
```

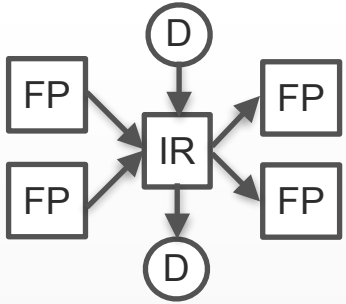
IS

```
enum CustomerId {
  case CustomerId(value: Int)
}
```

```
type Customer
  = Person firstName:String lastName:String
  | Robot id:Int
  | Unknown
```

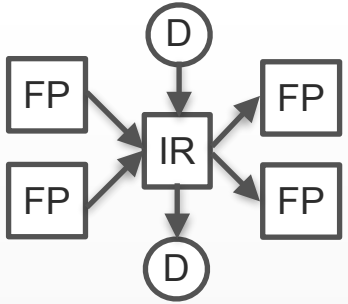
IS

```
enum Customer {
  case Person(firstName: String, lastName: String)
  case Robot(id: Int)
  case Unknown
}
```



Model Data
to match
intermediate
format

04



Model Data
to match
intermediate
format

04

```
type AccountNumber
  = AccountNumber Int
```

IS

```
enum AccountNumber {
  case AccountNumber(value: Int)
}
```

```
type CustomerId
  = CustomerId Int
```

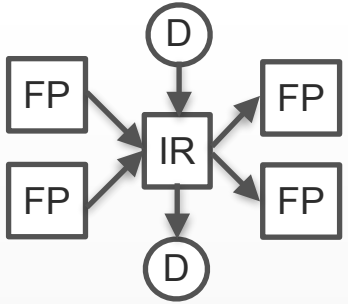
IS

```
enum CustomerId {
  case CustomerId(value: Int)
}
```

```
acctNum = AccountNumber 123
custId = CustomerId 456
```

IS

```
val acctNum = AccountNumber.AccountNumber(123)
val custId = CustomerId.CustomerId(456)
```



Model Data
to match
intermediate
format

04

```
type AccountNumber
  = AccountNumber Int
```

IS

```
enum AccountNumber {
  case AccountNumber(value: Int)
}
```

```
Data.Enum(
  values = List("value" -> Data.Int32(123)),
  case = "AccountNumber",
  schema = Schema.Enum("AccountNumber",
    List(
      Enum("AccountNumber", List("value" -> Int32))
    )
  )
)
```

```
type CustomerId
  = CustomerId Int
```

IS

```
enum CustomerId {
  case CustomerId(value: Int)
}
```

```
Data.Enum(
  values = List("value" -> Data.Int32(123)),
  case = "CustomerId",
  schema = Schema.Enum("CustomerId",
    List(
      Enum("CustomerId", List("value" -> Int32))
    )
  )
)
```

```
acctNum = AccountNumber 123
custId = CustomerId 456
```

IS

```
val acctNum = AccountNumber.AccountNumber(123)
val custId = CustomerId.CustomerId(456)
```

```
type AccountNumber
  = AccountNumber Int
```

IS

```
enum AccountNumber {
  case AccountNumber(value: Int)
}
```

```
type CustomerId
  = CustomerId Int
```

IS

```
enum CustomerId {
  case CustomerId(value: Int)
}
```

```
acctNum = AccountNumber 123
custId = CustomerId 456
```

IS

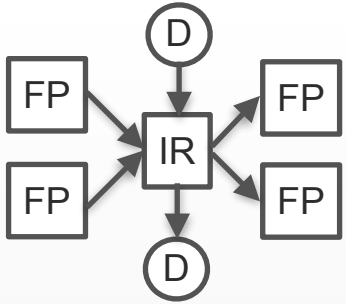
```
val acctNum = AccountNumber.AccountNumber(123)
val custId = CustomerId.CustomerId(456)
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber.AccountNumber](acctNum)
```

```
Data.Enum(
  values = List("value" -> Data.Int32(123)),
  case = "AccountNumber",
  schema = Schema.Enum("AccountNumber",
    List(
      Enum("AccountNumber", List("value" -> Int32))
    )
  )
)
```

```
val data: Data.Enum =
  DataBuilder.build[CustomerId.CustomerId](acctNum)
```

```
Data.Enum(
  values = List("value" -> Data.Int32(123)),
  case = "CustomerId",
  schema = Schema.Enum("CustomerId",
    List(
      Enum("CustomerId", List("value" -> Int32))
    )
  )
)
```



Model Data
to match
intermediate
format

04

```
type AccountNumber
  = AccountNumber Int
```

IS

```
enum AccountNumber {
  case AccountNumber(value: Int)
}
```

FANTASY

```
type CustomerId
  = CustomerId Int
```

IS

```
enum CustomerId {
  case CustomerId(value: Int)
}
```

FANTASY

```
acctNum = AccountNumber 123
custId = CustomerId 456
```

IS

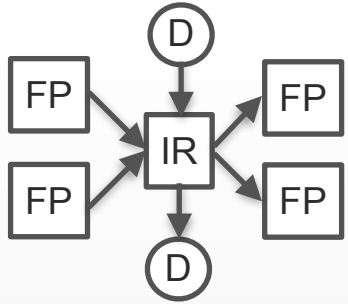
```
val acctNum = AccountNumber.AccountNumber(123)
val custId = CustomerId.CustomerId(456)
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber.AccountNumber](acctNum)
```

```
Data.Enum(
  values = List("value" -> Data.Int32(123)),
  case = "AccountNumber",
  schema = Schema.Enum("AccountNumber",
    List(
      Enum("AccountNumber", List("value" -> Int32))
    )
  )
)
```

```
val data: Data.Enum =
  DataBuilder.build[CustomerId.CustomerId](acctNum)
```

```
Data.Enum(
  values = List("value" -> Data.Int32(123)),
  case = "CustomerId",
  schema = Schema.Enum("CustomerId",
    List(
      Enum("CustomerId", List("value" -> Int32))
    )
  )
)
```



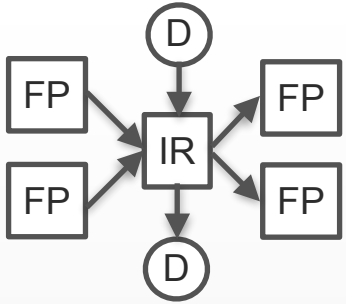
Model Data
to match
intermediate
format

04

```
type AccountNumber
  = AccountNumber Int
```

Has to Be

```
// Scala 2
class AccountNumber(val value: Int) extends AnyVal

// Scala 3
opaque type AccountNumber = Int

// zio-prelude
object AccountNumber extends Newtype[Int]
type AccountNumber = AccountNumber.type

// estatico
@newtype case class AccountNumber(value: Int)

// refined
type AccountNumber = Int Refined (conditions)

// etc, etc...
```



Model Data
to match
intermediate
format

04

```
type AccountNumber
  = AccountNumber Int
```

Has to Be

```
// Scala 2
class AccountNumber(val value: Int) extends AnyVal

// Scala 3
opaque type AccountNumber = Int

// zio-prelude
object AccountNumber extends Newtype[Int]
type AccountNumber = AccountNumber.type

// estatico
@newtype case class AccountNumber(value: Int)

// refined
type AccountNumber = Int Refined (conditions)

// etc, etc...
```

Just implement a custom data-builder!

```
type AccountNumber
  = AccountNumber Int
```

Has to Be

```
// Scala 2
class AccountNumber(val value: Int) extends AnyVal

// Scala 3
opaque type AccountNumber = Int

// zio-prelude
object AccountNumber extends Newtype[Int]
type AccountNumber = AccountNumber.type

// estatico
@newtype case class AccountNumber(value: Int)

// refined
type AccountNumber = Int Refined (conditions)

// etc, etc...
```

Just implement a custom data-builder!

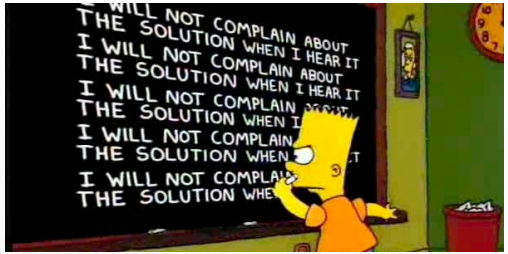
```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  new DataBuilder[AccountNumber] {
    val label = "AccountNumber"
    override def build(value: AccountNumber): Data.Enum =
      Data.Enum("value" -> Data.Int(value.value))(label, schema)
    override def schema: Schema.Enum =
      Schema.Enum(
        label,
        Schema.Enum.Enum(label, "value" -> Schema.Int)
      )
  }
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

```scala
type AccountNumber
  = AccountNumber Int
```

Has to Be

```scala
// Scala 2
class AccountNumber(val value: Int) extends AnyVal

// Scala 3
opaque type AccountNumber = Int

// zio-prelude
object AccountNumber extends Newtype[Int]
type AccountNumber = AccountNumber.type

// estatico
@newtype case class AccountNumber(value: Int)

// refined
type AccountNumber = Int Refined (conditions)

// etc, etc...
```




```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  new DataBuilder[AccountNumber] {
    val label = "AccountNumber"
    override def build(value: AccountNumber): Data.Enum =
      Data.Enum("value" -> Data.Int(value.value))(label, schema)
    override def schema: Schema.Enum =
      Schema.Enum(
        label,
        Schema.Enum.Enum(label, "value" -> Schema.Int)
      )
  }
```

```scala
implicit val customerIdDeriver: DataBuilder[CustomerId] =
  new DataBuilder[CustomerId] {
```

```scala
implicit val productIdDeriver: DataBuilder[ProductId] =
  new DataBuilder[ProductId] {
```

```scala
implicit val orderIdDeriver: DataBuilder[ProductId] =
  new DataBuilder[OrderId] {
```

```scala
implicit val supplierIdDeriver: DataBuilder[SupplierId] =
  new DataBuilder[SupplierId] {
```

```scala
implicit val firmIdDeriver: DataBuilder[FirmId] =
  new DataBuilder[FirmId] {
```

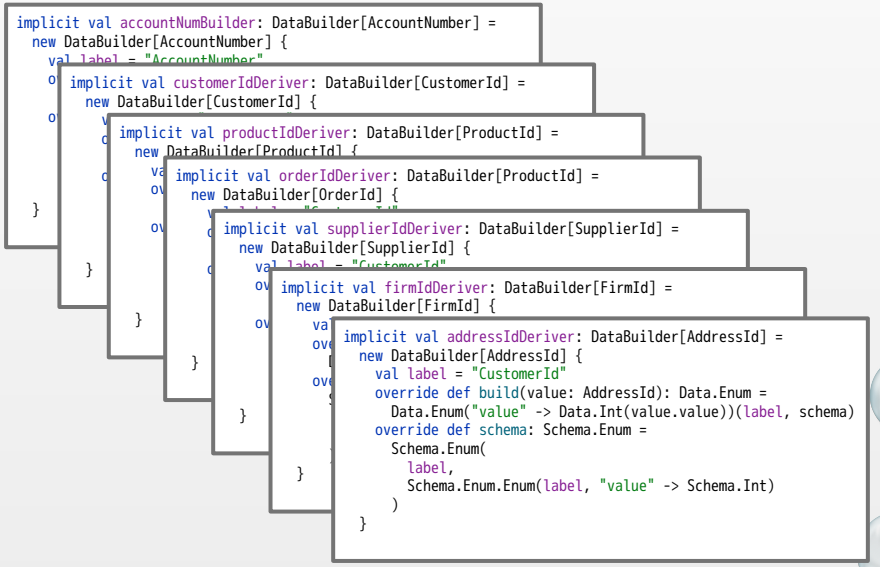
```scala
implicit val addressIdDeriver: DataBuilder[AddressId] =
  new DataBuilder[AddressId] {
    val label = "CustomerId"
    override def build(value: AddressId): Data.Enum =
      Data.Enum("value" -> Data.Int(value.value))(label, schema)
    override def schema: Schema.Enum =
      Schema.Enum(
        label,
        Schema.Enum.Enum(label, "value" -> Schema.Int)
      )
  }
```

```scala
type AccountNumber
  = AccountNumber Int
```

Has to Be

```scala
// Scala 2
class AccountNumber(val value: Int) extends AnyVal

// Scala 3
opaque type AccountNumber = Int

// zio-prelude
object AccountNumber extends Newtype[Int]
type AccountNumber = AccountNumber.type

// estatico
@newtype case class AccountNumber(value: Int)

// refined
type AccountNumber = Int Refined (conditions)

// etc, etc...
```

```scala
implicit val customerIdBuilder: CustomDataBuilder[CustomerId] =
  new CustomDataBuilder[CustomerId] {
    val wrapper = MakeNewtypeEnum("CustomerId", Schema.Int)
    override def build(value: CustomerId): Data.Enum =
      wrapper.construct(Data.Int(value.value))
    override def schema: Schema.Enum =
      wrapper.schema
  }
```



```
type AccountNumber
  = AccountNumber Int
```

Has to Be

```
// Scala 2
class AccountNumber(val value: Int) extends AnyVal

// Scala 3
opaque type AccountNumber = Int

// zio-prelude
object AccountNumber extends Newtype[Int]
type AccountNumber = AccountNumber.type

// estatico
@newtype case class AccountNumber(value: Int)

// refined
type AccountNumber = Int Refined (conditions)

// etc, etc...
```

Add **helper functions** for user constructed type-classes?

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  new DataBuilder[AccountNumber] {
    val label = "AccountNumber"
    override def build(value: AccountNumber): Data.Enum =
      Data.Enum("value" -> Data.Int(value.value))(label, schema)
    override def schema: Schema.Enum =
      Schema.Enum(
        label,
        Schema.Enum.Enum(label, "value" -> Schema.Int)
      )
  }
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

```
type AccountNumber
  = AccountNumber Int
```

Has to Be

```
// Scala 2
class AccountNumber(val value: Int) extends AnyVal

// Scala 3
opaque type AccountNumber = Int

// zio-prelude
object AccountNumber extends Newtype[Int]
type AccountNumber = AccountNumber.type

// estatico
@newtype case class AccountNumber(value: Int)

// refined
type AccountNumber = Int Refined (conditions)

// etc, etc...
```

Probably want to do better **than this**...

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  new DataBuilder[AccountNumber] {
    val wrapper = MakeNewtypeEnum("AccountNumber", Schema.Int)
    override def build(value: AccountNumber): Data.Enum =
      wrapper.construct(Data.Int(value.value))
    override def schema: Schema.Enum =
      wrapper.schema
  }
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

```
type AccountNumber
  = AccountNumber Int
```

Has to Be

```
// Scala 2
class AccountNumber(val value: Int) extends AnyVal

// Scala 3
opaque type AccountNumber = Int

// zio-prelude
object AccountNumber extends Newtype[Int]
type AccountNumber = AccountNumber.type

// estatico
@newtype case class AccountNumber(value: Int)

// refined
type AccountNumber = Int Refined (conditions)

// etc, etc...
```

Make it a **one liner**!

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber]((an: AccountNumber) => an.value :Int)
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

Just implement a custom data-builder!

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber](_.value :Int)

implicit val customerIdDeriver: DataBuilder[CustomerId] =
  NewtypeEnum.build[CustomerId](_.value:Int)

implicit val productIdDeriver: DataBuilder[ProductId] =
  NewtypeEnum.build[ProductId](_.value:String)

implicit val productTypeDeriver: DataBuilder[ProductType] =
  NewtypeEnum.build[ProductType](_.value:Char)

etc...

val data: Data.Enum =
 DataBuilder.build[AccountNumber](acctNum)
```

Overload Them!

```
object NewtypeEnum {
  @targetName("deriveInt")
  inline def build[NewType](extractValue: NewType => Int) =
    new DataBuilder[NewType] {
      val wrapper = MakeNewtypeEnum(typeName[NewType], Schema.Int)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(Data.Int(extractValue(value)))
      override def schema: Schema                    =
        wrapper.schema
    }

  @targetName("deriveString")
  inline def build[NewType](extractValue: NewType => String) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], Schema.String)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(Data.String(extractValue(value)))
      override def schema: Schema =
        wrapper.schema
    }

  rinse... repeat!
}
```

Just implement a custom data-builder!

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber](_.value :Int)

implicit val customerIdDeriver: DataBuilder[CustomerId] =
  NewtypeEnum.build[CustomerId](_.value:Int)

implicit val productIdDeriver: DataBuilder[ProductId] =
  NewtypeEnum.build[ProductId](_.value:String)

implicit val productTypeDeriver: DataBuilder[ProductType] =
  NewtypeEnum.build[ProductType](_.value:Char)

etc...
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

Overload Them!

```
object NewtypeEnum {
  inline def build[NewType](extractValue: NewType => Int) =
  inline def build[NewType](extractValue: NewType => String) =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => Boolean) =
  inline def build[NewType](extractValue: NewType => Long) =
  inline def build[NewType](extractValue: NewType => Short) =
  inline def build[NewType](extractValue: NewType => Double) =
  inline def build[NewType](extractValue: NewType => Float) =
  inline def build[NewType](extractValue: NewType => Byte) =
}
```

Just implement a custom data-builder!

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber](_.value :Int)

implicit val customerIdDeriver: DataBuilder[CustomerId] =
  NewtypeEnum.build[CustomerId](_.value:Int)

implicit val productIdDeriver: DataBuilder[ProductId] =
  NewtypeEnum.build[ProductId](_.value:String)

implicit val productTypeDeriver: DataBuilder[ProductType] =
  NewtypeEnum.build[ProductType](_.value:Char)
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

Overload Them!

```
object NewtypeEnum {
  inline def build[NewType](extractValue: NewType => Int) =
  inline def build[NewType](extractValue: NewType => String) =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => Boolean) =
  inline def build[NewType](extractValue: NewType => Long) =
  inline def build[NewType](extractValue: NewType => Short) =
  inline def build[NewType](extractValue: NewType => Double) =
  inline def build[NewType](extractValue: NewType => Float) =
  inline def build[NewType](extractValue: NewType => Byte) =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => LocalDate) =
  inline def build[NewType](extractValue: NewType => LocalTime) =
  inline def build[NewType](extractValue: NewType => LocalDateTime) =
  inline def build[NewType](extractValue: NewType => Month) =
  inline def build[NewType](extractValue: NewType => DayOfMonth) =
}
```

Just implement a custom data-builder!

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber](_.value :Int)

implicit val customerIdDeriver: DataBuilder[CustomerId] =
  NewtypeEnum.build[CustomerId](_.value:Int)

implicit val productIdDeriver: DataBuilder[ProductId] =
  NewtypeEnum.build[ProductId](_.value:String)

implicit val productTypeDeriver: DataBuilder[ProductType] =
  NewtypeEnum.build[ProductType](_.value:Char)
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

Overload Them!

```
object NewtypeEnum {
  inline def build[NewType](extractValue: NewType => Int) =
  inline def build[NewType](extractValue: NewType => String) =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => Boolean) =
  inline def build[NewType](extractValue: NewType => Long) =
  inline def build[NewType](extractValue: NewType => Short) =
  inline def build[NewType](extractValue: NewType => Double) =
  inline def build[NewType](extractValue: NewType => Float) =
  inline def build[NewType](extractValue: NewType => Byte) =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => LocalDate) =
  inline def build[NewType](extractValue: NewType => LocalTime) =
  inline def build[NewType](extractValue: NewType => LocalDateTime) =
  inline def build[NewType](extractValue: NewType => Month) =
  inline def build[NewType](extractValue: NewType => DayOfMonth) =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => BigDecimal) =
  inline def build[NewType](extractValue: NewType => BigInt) =
  ...
}
```

Just implement a custom data-builder!

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber](_.value :Int)

implicit val customerIdDeriver: DataBuilder[CustomerId] =
  NewtypeEnum.build[CustomerId](_.value:Int)

implicit val productIdDeriver: DataBuilder[ProductId] =
  NewtypeEnum.build[ProductId](_.value:String)

implicit val productTypeDeriver: DataBuilder[ProductType] =
  NewtypeEnum.build[ProductType](_.value:Char)
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

Overload Then

```
object NewtypeEnum {
  inline def build[NewType](ex
  inline def build[New pe](ex

  rinse... repeat!
  inline def build[New pe](ex
  inline def build[NewTy e](ex
  inline def build[NewType](extractValue: NewType => Short) =
  inline def build[NewType](extractValue: NewType => Double) =
  inline def build[NewType](extractValue: NewType => Float) =
  inline def build[NewType](extractValue: NewType => Byte) =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => LocalDate) =
  inline def build[NewType](extractValue: NewType => LocalTime) =
  inline def build[NewType](extractValue: NewType => LocalDateTime) =
  inline def build[NewType](extractValue: NewType => Month) =
  inline def build[NewType](extractValue: NewType => DayOfMonth) =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => BigDecimal) =
  inline def build[NewType](extractValue: NewType => BigInt) =
  ...
}
```

```
object NewtypeEnum {
  @targetName("deriveInt")
  inline def build[NewType](extractValue: NewType => Int) =
    new DataBuilder[NewType] {
      val wrapper = MakeNewtypeEnum(typeName[NewType], Schema.Int)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(Data.Int(extractValue(value)))
      override def schema: Schema                    =
        wrapper.schema
    }
}
```

Just implement a custom data-builder!

```
NumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber](_.value :Int)

implicit val customerIdDeriver: DataBuilder[CustomerId] =
  NewtypeEnum.build[CustomerId](_.value:Int)

implicit val productIdDeriver: DataBuilder[ProductId] =
  NewtypeEnum.build[ProductId](_.value:String)

implicit val productTypeDeriver: DataBuilder[ProductType] =
  NewtypeEnum.build[ProductType](_.value:Char)
```

val data: Data.Enum =
DataBuilder.*build*[AccountNumber](*acctNum*)

Overload Them!

```
object NewtypeEnum {
  inline def build[NewType](extractValue: NewType => Int) =
  inline def build[NewType](extractValue: NewType => String) =

  rinse... repeat!
  inline
  inline
  inline
  inline
  inline
  inline

  rinse..
  inline                                                    =
  inline                                                    =
  inline                                                 me) =
  inline
  inline                                                    =

  rinse... repeat!
  inline def build[NewType](extractValue: NewType => BigDecimal) =
  inline def build[NewType](extractValue: NewType => BigInt) =
  ...
}
```

Just implement a custom data-builder!

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber](_.value :Int)

implicit val customerIdDeriver: DataBuilder[CustomerId] =
  NewtypeEnum.build[CustomerId](_.value:Int)

implicit val productIdDeriver: DataBuilder[ProductId] =
  NewtypeEnum.build[ProductId](_.value:String)

implicit val productTypeDeriver: DataBuilder[ProductType] =
  NewtypeEnum.build[ProductType](_.value:Char)
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

Pathodental **Lie 01**

Boilerplate is Always Bad

```
object NewtypeEnum {
 @targetName("deriveInt")
  inline def build[NewType](extractValue: NewType => Int) =
    new DataBuilder[NewType] {
      val wrapper = MakeNewtypeEnum(typeName[NewType], Schema.Int)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(Data.Int(extractValue(value)))
      override def schema: Schema                   =
        wrapper.schema
    }

  @targetName("deriveString")
  inline def build[NewType](extractValue: NewType => String) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], Schema.String)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(Data.String(extractValue(value)))
      override def schema: Schema =
        wrapper.schema
    }

  ...
}
```

DataBuilder for **AccountNumber** Type

```
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber](_.value)
```

```
val data: Data.Enum =
  DataBuilder.build[AccountNumber](acctNum)
```

Using **Type Classes** 01

```scala
trait FromPrimitive[MorphirType, Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  new FromPrimitive[Data.Boolean, Boolean] {
    override def shape = Schema.Boolean
    override def fromPrimitive(value: Boolean) = Data.Boolean(value)
  }

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  new FromPrimitive[Data.Int, Int] {
    override def shape = Schema.Int
    override def fromPrimitive(value: Int) = Data.Int(value)
  }
```

-50% **Boilerplate**

```scala
object NewtypeEnum {
  inline def build[NewType, MorphirType, Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber, Data.Int, Int](_.value)
```

Using **Type Classes** 01

```scala
trait FromPrimitive[MorphirType, Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  new FromPrimitive[Data.Boolean, Boolean] {
    override def shape = Schema.Boolean
    override def fromPrimitive(value: Boolean) = Data.Boolean(value)
  }

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  new FromPrimitive[Data.Int, Int] {
    override def shape = Schema.Int
    override def fromPrimitive(value: Int) = Data.Int(value)
  }
```

-50% Boilerplate

```scala
object NewtypeEnum {
  inline def build[NewType, MorphirType, Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

Bleeds into the user API!

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum.build[AccountNumber, Data.Int, Int](_.value)
```

Using **Type Classes + Type-Search**

02

```scala
trait FromPrimitive[MorphirType <: Data.Basic[Primitive], Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  new FromPrimitive[Data.Boolean, Boolean] {
    override def shape = Schema.Boolean
    override def fromPrimitive(value: Boolean) = Data.Boolean(value)
  }

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  new FromPrimitive[Data.Int, Int] {
    override def shape = Schema.Int
    override def fromPrimitive(value: Int) = Data.Int(value)
  }
```

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(_.value)
```

Using **Type Classes + Type-Search**

02

```scala
trait FromPrimitive[MorphirType <: Data.Basic[Primitive], Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  new FromPrimitive[Data.Boolean, Boolean] {
    override def shape = Schema.Boolean
    override def fromPrimitive(value: Boolean) = Data.Boolean(value)
  }

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  new FromPrimitive[Data.Int, Int] {
    override def shape = Schema.Int
    override def fromPrimitive(value: Int) = Data.Int(value)
  }
```

-60% **Boilerplate**

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(_.value)
```

Using **Type Classes + Type-Search** 02

```scala
trait FromPrimitive[MorphirType <: Data.Basic[Primitive], Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  new FromPrimitive[Data.Boolean :Basic[Boolean], Boolean] {
    override def shape = Schema.Boolean
    override def fromPrimitive(value: Boolean) = Data.Boolean(value)
  }

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  new FromPrimitive[Data.Int :Basic[Int], Int] {
    override def shape = Schema.Int
    override def fromPrimitive(value: Int) = Data.Int(value)
  }
```

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
case class Int(value: scala.Int) extends Basic[scala.Int]
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(extractValue: _.value :Int)
```

Using **Type Classes + Type-Search**

02

```scala
trait FromPrimitive[MorphirType <: Data.Basic[Primitive], Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  new FromPrimitive[Data.Boolean :Basic[Boolean], Boolean] {
    override def shape = Schema.Boolean
    override def fromPrimitive(value: Boolean) = Data.Boolean(value)
  }

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  new FromPrimitive[Data.Int :Basic[Int], Int] {
    override def shape = Schema.Int
    override def fromPrimitive(value: Int) = Data.Int(value)
  }
```

-60% **Boilerplate**

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
case class Int(value: scala.Int) extends Basic[scala.Int]
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(_.value :Int)
```

Using **Type Classes + Type-Search + Type Class Constructors**

03

```scala
trait FromPrimitive[MorphirType <: Data.Basic[Primitive], Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
object FromPrimitive {
  def apply[MorphirType <: Data.Basic[Primitive], Primitive](
    morphirShape: Schema.Basic[Primitive],
    buildData: Primitive => MorphirType
  ): FromPrimitive[MorphirType, Primitive] =
    new FromPrimitive[MorphirType, Primitive] {
      override def shape = morphirShape
      override def fromPrimitive(value: Primitive) = buildData(value)
    }
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  new FromPrimitive[Data.Boolean, Boolean] {
    override def shape = Schema.Boolean
    override def fromPrimitive(value: Boolean) = Data.Boolean(value)
  }

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  new FromPrimitive[Data.Int, Int] {
    override def shape = Schema.Int
    override def fromPrimitive(value: Int) = Data.Int(value)
  }
```

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(_.value)
```

Using **Type Classes + Type-Search + Type Class Constructors**

03

```scala
trait FromPrimitive[MorphirType <: Data.Basic[Primitive], Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
object FromPrimitive {
  def apply[MorphirType <: Data.Basic[Primitive], Primitive](
    morphirShape: Schema.Basic[Primitive],
    buildData: Primitive => MorphirType
  ): FromPrimitive[MorphirType, Primitive] =
    new FromPrimitive[MorphirType, Primitive] {
      override def shape = morphirShape
      override def fromPrimitive(value: Primitive) = buildData(value)
    }
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  FromPrimitive[Data.Boolean, Boolean](Schema.Boolean, Data.Boolean(_))

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  FromPrimitive[Data.Int, Int](Schema.Int, Data.Int(_))
```

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(_.value)
```

Using **Type Classes + Type-Search + Type Class Constructors**

03

```scala
trait FromPrimitive[MorphirType <: Data.Basic[Primitive], Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
object FromPrimitive {
  def apply[MorphirType <: Data.Basic[Primitive], Primitive](
    morphirShape: Schema.Basic[Primitive],
    buildData: Primitive => MorphirType
  ): FromPrimitive[MorphirType, Primitive] =
    new FromPrimitive[MorphirType, Primitive] {
      override def shape = morphirShape
      override def fromPrimitive(value: Primitive) = buildData(value)
    }
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  FromPrimitive[Data.Boolean, Boolean](Schema.Boolean, Data.Boolean(_))

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  FromPrimitive[Data.Int, Int](Schema.Int, Data.Int(_))
```

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(_.value)
```

-80% **Boilerplate**

Using **Type Classes + Type-Search + Type Class Constructors**

03

```scala
trait FromPrimitive[MorphirType <: Data.Basic[Primitive], Primitive] {
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MorphirType
}
object FromPrimitive {
  def apply[MorphirType <: Data.Basic[Primitive], Primitive](
    morphirShape: Schema.Basic[Primitive],
    buildData: Primitive => MorphirType
  ): FromPrimitive[MorphirType, Primitive] =
    new FromPrimitive[MorphirType, Primitive] {
      override def shape = morphirShape
      override def fromPrimitive(value: Primitive) = buildData(value)
    }
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Data.Boolean, Boolean] =
  FromPrimitive[Data.Boolean, Boolean](Schema.Boolean, Data.Boolean(_))

implicit val intFromPrimitive: FromPrimitive[Data.Int, Int] =
  FromPrimitive[Data.Int, Int](Schema.Int, Data.Int(_))
```

-80% **Boilerplate**

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(_.value)
```

Using **Type Classes** + **Type-Search** + **Type Class Constructors** + **Type-Level Function**

04

```scala
trait FromPrimitive[Primitive] {
  type MType <: Data.Basic[Primitive]
  def shape: Schema.Basic[Primitive]
  def fromPrimitive(value: Primitive): MType
}
object FromPrimitive {
  def apply[MorphirType <: Data.Basic[Primitive], Primitive](
    morphirShape: Schema.Basic[Primitive],
    buildData: Primitive => MorphirType
  ): FromPrimitive[Primitive] =
    new FromPrimitive[Primitive] {
      override type MType = MorphirType
      override def shape = morphirShape
      override def fromPrimitive(value: Primitive) = buildData(value)
    }
}
```

```scala
implicit val booleanFromPrimitive: FromPrimitive[Boolean] =
  FromPrimitive(Schema.Boolean, Data.Boolean(_))

implicit val intFromPrimitive: FromPrimitive[Int] =
  FromPrimitive(Schema.Int, Data.Int(_))
```

-85% Boilerplate

```scala
class NewtypeEnum[NewType] {
  inline def build[MorphirType <: Data.Basic[Primitive], Primitive](
    extractValue: NewType => Primitive
  )(implicit repr: FromPrimitive[MorphirType, Primitive]) =
    new DataBuilder[NewType] {
      val wrapper =
        MakeNewtypeEnum(typeName[NewType], repr.shape)
      override def build(value: NewType): Data.Enum =
        wrapper.construct(repr.fromPrimitive(extractValue(value)))
      override def schema: Schema = wrapper.schema
    }
}
```

```scala
implicit val accountNumBuilder: DataBuilder[AccountNumber] =
  NewtypeEnum[AccountNumber].build(_.value)
```

WARNING

The next slide portrays vulgar Guesstimation that is inappropriate for some audiences.

Viewer discretion is Advised

Who **can** maintain this?
(a.k.a the Hiring Pool)

Function **per** Implementation	00	(27m) Most Programmers
Using **Type Classes**	01	(700k) Most Scala (**-Spark**), Some (Professional) Rust, and All Haskell Programmers
Using **Type Classes** **+ Type-Search**	02	Many Scala (**-Spark**) and and Haskell Programmers **(200k?)**
Using **Type Classes** **+ Type-Search** **+ Type Class Constructors**	03	Some Scala (**-Spark**) and and Haskell Programmers **(100k?)**
Using **Type Classes** **+ Type-Search** **+ Type Class Constructors** **+ Type-Level Function**	04	Few Scala (**-Spark**) and and Haskell Programmers **(60k?)**

WARNING

This slide uses Guesstimation

1.5% of developers are **actually** on the job market?

Function **per** Implementation — 00 — (400k) Most Programmers

Using **Type Classes** — 01 — (10.5k) Most Scala (**-Spark**), Some (Professional) Rust, and All Haskell Programmers

Using **Type Classes** **+ Type-Search** — 02 — Many Scala (**-Spark**) and and Haskell Programmers **(3k?)**

Using **Type Classes** **+ Type-Search** **+ Type Class Constructors** — 03 — Some Scala (**-Spark**) and and Haskell Programmers **(1.5k?)**

Using **Type Classes** **+ Type-Search** **+ Type Class Constructors** **+ Type-Level Function** — 04 — Few Scala (**-Spark**) and and Haskell Programmers **(900?)**

WARNING

This slide uses Guesstimation

Function **per** Implementation	00	Baseline
Using **Type Classes**	01	Complexity is intrinsic, too much library boilerplate otherwise!
Using **Type Classes** + **Type-Search**	02	Complexity is intrinsic, too much library boilerplate otherwise!
Using **Type Classes** + **Type-Search** + **Type Class Constructors**	03	Complexity is intrinsic, too much library boilerplate otherwise!
Using **Type Classes** + **Type-Search** + **Type Class Constructors** + **Type-Level Function**	04	Complexity is intrinsic, too much library boilerplate otherwise!

WARNING

This slide uses Guesstimation

Lean on **Copilot?**

On the User - No!

```
implicit val customerIdBuilder: CustomDataBuilder[CustomerId] =
  new CustomDataBuilder[CustomerId] {
    val wrapper = MakeNewtypeEnum("CustomerId", Schema.Int)
    override def build(value: CustomerId): Data.Enum =
      wrapper.construct(Data.Int(value.value))
    override def schema: Schema.Enum =
      wrapper.schema
  }
```

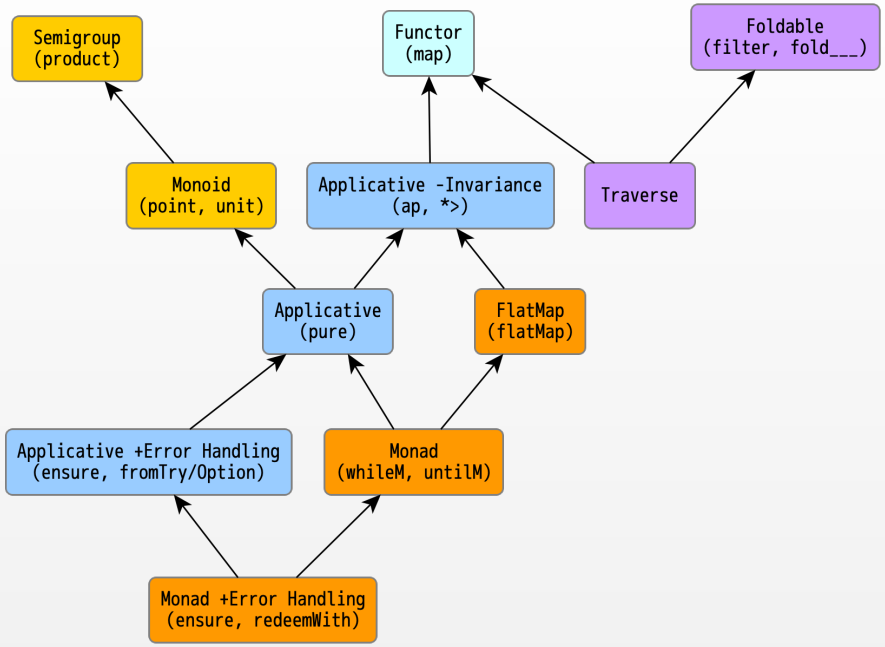
On the Library Author - Why Not?

No complex logic inside the body of the primitive data-builder classes
it makes code maintainable by 100x more people

```
@targetName("buildString")
inline def build[NewType](extractValue: NewType => String): CustomDataBuilder[NewType] =
  new CustomDataBuilder[NewType] {
    val wrapper                                    =
      MakeNewtypeEnum(typeName[NewType], Schema.String)
    override def build(value: NewType): Data.Enum =
      wrapper.construct(Data.String(extractValue(value)))
    override def schema: Schema                    =
      wrapper.schema
  }
```

Pathodental **Lie 02**

Functional Programming is Simple

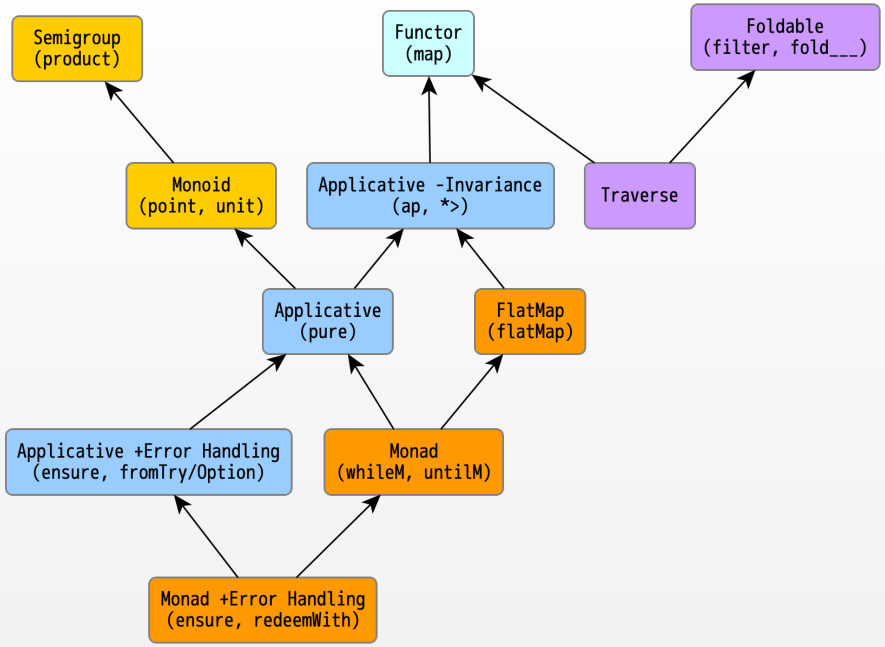


FP is all **about?**

These are design patterns?

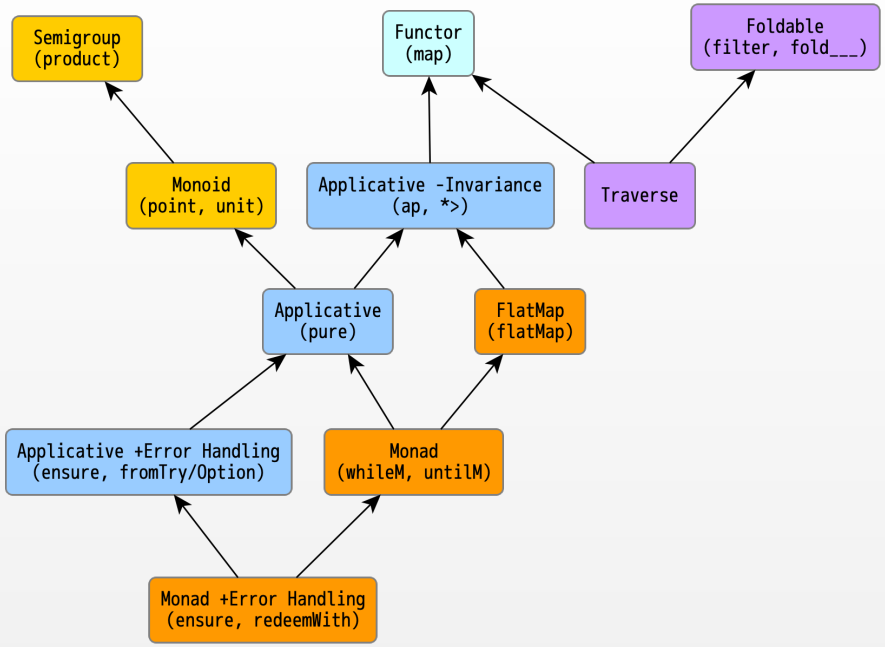
They look more like axioms...

FP is all **about?**




```scala
def traverse[F[_] : Applicative, A, B](
  as: List[A]
)(
  f: A => F[B]
): F[List[B]] =
  as.foldRight(Applicative[F].pure(List.empty[B])) {
    (a: A, acc: F[List[B]]) =>
    val fb: F[B] = f(a)
    Applicative[F].map2(fb, acc)(_ :: _)
  }
```

```scala
def groupBy[K, T, G, P](f: E => K)(implicit kshape: Shape[? <: FlatShapeLevel, K, T, G],
  vshape: Shape[? <: FlatShapeLevel, E, ?, P]
): Query[(G, Query[P, U, Seq]), (T, Query[P, U, Seq]), C] = {
  val sym = new AnonSymbol
  val key = ShapedValue(f(shaped.encodeRef(Ref(sym)).value), kshape).packedValue
  val value = ShapedValue(pack.to[Seq], RepShape[FlatShapeLevel, Query[P, U, Seq], Query[P, U, Seq]])
  val group = GroupBy(sym, toNode, key.toNode)
  new WrappingQuery[(G, Query[P, U, Seq]), (T, Query[P, U, Seq]), C](group, key.zip(value))
}
```



What are the actual design patterns?

OO pattern/principle	**FP pattern/principle**
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory Pattern	• Yes, functions
• Strategy Pattern	• Oh my, functions again!
• Decorator Pattern	• Functions
• Visitor Pattern	• Functions

...wait, there **are none?**

OO pattern/principle	FP pattern/principle
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory Pattern	• Yes, functions
• Strategy Pattern	• Oh my, functions again!
• Decorator Pattern	• Functions
• Visitor Pattern	• Functions

...wait, there **are none?**

OO pattern/principle	FP pattern/principle
• Single Responsibility Principle	• Functions
• Open/Closed principle	• Functions
• Dependency Inversion Principle	• Functions, also
• Interface Segregation Principle	• Functions
• Factory Pattern	• Yes, functions
• Strategy Pattern	• Oh my, functions again!
• Decorator Pattern	• Functions
• Visitor Pattern	• Functions

Products

Typeclasses

OO pattern/principle	**FP pattern/principle**
• Single Responsibility Principle	• ~~Functions~~ Coproducts/Modules
• Open/Closed principle	• ~~Functions~~ Inner Defs/Types, Delegation
• Dependency Inversion Principle	• ~~Functions, also~~ Reader/Writer Monads
• Interface Segregation Principle	• ~~Functions~~ Same idea!
• Factory Pattern	• ~~Yes, functions~~ things with .apply(...) function
• Strategy Pattern	• ~~Oh my, functions again!~~ HOFs
• Decorator Pattern	• ~~Functions~~ Mixins
• Visitor Pattern	• ~~Functions~~ Recursive Pattern Matching

The Reality...!

What do you some monad XYZ for?

What do you use some semigroup XYZ for?

OOP pattern/principle	FP pattern/principle
Single Responsibility Principle	Coproducts data modeling Careful Module scoping
Open/Closed Principle	Delegation instead of Inheritance Inner functions, Inner types
Dependency Inversion Principle	Reader Monad, Writer Monad ZLayers and R parameter, Also ZIO Service Pattern constructor inputs
Interface Segregation Principle	**Same! Used in:** Typeclass Traits, ZIO Service Pattern
Factory Pattern	Executable objects and classes i.e. the apply(…) function.
Strategy Pattern	Higher-Order Functions, Functors, Monads, etc…
Decorator Pattern	Self-types/Mixins
Visitor Pattern	Recursive pattern matching

OOP pattern/principle	FP pattern/principle
Single Responsibility Principle	Coproducts data modeling Careful Module scoping
Open/Closed Principle	Delegation instead of Inheritance Inner functions, Inner types
Dependency Inversion Principle	Reader Monad, Writer Monad ZLayers and R parameter, Also ZIO Service Pattern constructor inputs
Interface Segregation Principle	**Same! Used in:** Typeclass Traits, ZIO Service Pattern
Factory Pattern	Executable objects and classes i.e. the apply(…) function.
Strategy Pattern	Higher-Order Functions, Functors, Monads, etc…
Decorator Pattern	Self-types/Mixins
Visitor Pattern	Recursive pattern matching

and some more!

Adapter Pattern	Contramap (i.e. Contravariant Functor)
Bridge Pattern	**Same! Used in:** live/test in ZIO Service Pattern
Composite	Monoid, Semigroup
Façade	**Same! Used in:** Chunk and Chain

OOP pattern/principle	FP pattern/principle
Single Responsibility Principle	Coproducts data modeling Careful Module scoping
Open/Closed Principle	Delegation instead of Inheritance Inner functions, Inner types
Dependency Inversion Principle	Reader Monad, Writer Monad ZLayers and R parameter, Also ZIO Service Pattern constructor inputs
Interface Segregation Principle	**Same! Used in:** Typeclass Traits, ZIO Service Pattern
Factory Pattern	Executable objects and classes i.e. the apply(…) function.
Strategy Pattern	**Higher-Order Functions**, Functors, Monads, etc…
Decorator Pattern	Self-types/Mixins
Visitor Pattern	**Recursive pattern matching**

and some more!

Adapter Pattern	**Contramap** (i.e. Contravariant Functor)
Bridge Pattern	**Same! Used in:** live/test in ZIO Service Pattern
Composite	Monoid, Semigroup
Façade	**Same! Used in:** Chunk and Chain

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
Gson gson = new GsonBuilder().create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
Gson gson = new GsonBuilder().create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":{"value":"123"},"firstName":"John","lastName":"Doe"}
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
Gson gson = new GsonBuilder().create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":{"value":"123"},"firstName":"John","lastName":"Doe"}
```

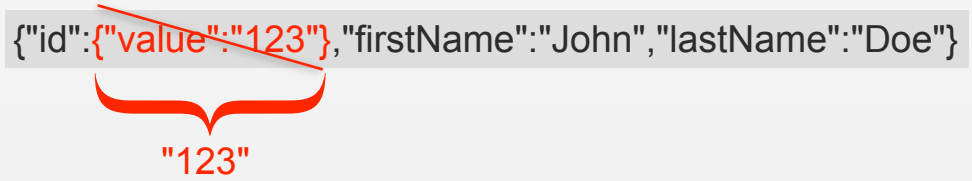
The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
Gson gson = new GsonBuilder().create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```



The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value :String);
    }
}
```

```
Gson gson = new GsonBuilder().create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":{"value":"123"},"firstName":"John","lastName":"Doe"}
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":{"value":"123"},"firstName":"John","lastName":"Doe"}
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":"123","firstName":"John","lastName":"Doe"}
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":"123","firstName":"John","lastName":"Doe"}
```

Model

```
case class UserId(String value)
case class Person(UserId id, String firstName, String lastName)
```

OOP FP

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":"123","firstName":"John","lastName":"Doe"}
```

Model

```
case class UserId(String value)
case class Person(UserId id, String firstName, String lastName)
```

Serializer

```
implicit val userIdEncoder: JsonEncoder[UserId] =
  DeriveJsonEncoder.gen[UserId]
implicit val personEncoder: JsonEncoder[Person] =
  DeriveJsonEncoder.gen[Person]
val id = UserId("123")
val person = Person(id, "John", "Doe")

println(person.toJson)
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":"123","firstName":"John","lastName":"Doe"}
```

Model

```
case class UserId(String value)
case class Person(UserId id, String firstName, String lastName)
```

Serializer

```
implicit val userIdEncoder: JsonEncoder[UserId] =
  DeriveJsonEncoder.gen[UserId]
implicit val personEncoder: JsonEncoder[Person] =
  DeriveJsonEncoder.gen[Person]
val id = UserId("123")
val person = Person(id, "John", "Doe")

println(person.toJson)
```

```
{"id":{"value":"123"},"firstName":"John","lastName":"Doe"}
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":"123","firstName":"John","lastName":"Doe"}
```

Model

```
case class UserId(String value)
case class Person(UserId id, String firstName, String lastName)
```

Serializer

```
implicit val personEncoder: JsonEncoder[Person] =
  DeriveJsonEncoder.gen[Person]
val id = UserId("123")
val person = Person(id, "John", "Doe")

println(person.toJson)
```

```
{"id":{"value":"123"},"firstName":"John","lastName":"Doe"}
```

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":"123","firstName":"John","lastName":"Doe"}
```

Model

```
case class UserId(String value)
case class Person(UserId id, String firstName, String lastName)
```

Serializer

```
implicit val userIdEncoder: JsonEncoder[UserId] =
  JsonEncoder.string.contramap(
    (src: UserId) => src.value :String
  )
```

```
implicit val personEncoder: JsonEncoder[Person] =
  DeriveJsonEncoder.gen[Person]
val id = UserId("123")
val person = Person(id, "John", "Doe")

println(person.toJson)
```

```
{"id":{"value":"123"},"firstName":"John","lastName":"Doe"}
```

OOP
FP

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

```
{"id":"123","firstName":"John","lastName":"Doe"}
```

Model

```
case class UserId(String value)
case class Person(UserId id, String firstName, String lastName)
```

Serializer

```
implicit val userIdEncoder: JsonEncoder[UserId] =
  JsonEncoder.string.contramap(
    (src: UserId) => src.value
  )
```

```
implicit val personEncoder: JsonEncoder[Person] =
  DeriveJsonEncoder.gen[Person]
val id = UserId("123")
val person = Person(id, "John", "Doe")

println(person.toJson)
```

```
{"id":"123","firstName":"John","lastName":"Doe"}
```

OOP FP

The Great **Adapter Pattern**

Model

```
record UserId(String value) {}
record Person(UserId id, String firstName, String lastName) {}
```

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            java.lang.reflect.Type typeOfSrc,
            JsonSerializationContext context) {
        return new JsonPrimitive(src.value);
    }
}
```

```
Gson gson = new GsonBuilder()
  .registerTypeAdapter(UserId.class, new UserIdAdapter())
  .create();
UserId id = new UserId("123");
Person person = new Person(id, "John", "Doe");
System.out.println(gson.toJson(person));
```

{"id":"123","firstName":"John","lastName":"Doe"}

Model

```
case class UserId(String value)
case class Person(UserId id, String firstName, String lastName)
```

Serializer

```
implicit val userIdEncoder: JsonEncoder[UserId] =
  JsonEncoder.string.contramap(
    (src: UserId) => src.value
  )
```

```
implicit val personEncoder: JsonEncoder[Person] =
  DeriveJsonEncoder.gen[Person]
val id = UserId("123")
val person = Person(id, "John", "Doe")

println(person.toJson)
```

{"id":"123","firstName":"John","lastName":"Doe"}

OOP FP

The Great **Adapter Pattern**

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            ...) {
        return new JsonPrimitive(src.value);
    }
}
```

Serializer

```
implicit val userIdEncoder: JsonEncoder[UserId] =
  JsonEncoder.string.contramap(
    (src: UserId) => src.value
  )
```

The Great **Adapter Pattern**

Serializer

```
public static class UserIdAdapter implements JsonSerializer<UserId> {
    @Override
    public JsonElement serialize(
            UserId src,
            ...) {
        return new JsonPrimitive(src.value);
    }
}
```

Serializer

```
implicit val userIdEncoder: JsonEncoder[UserId] =
  JsonEncoder.string.contramap(
    (src: UserId) => src.value
  )
```

OOP
FP

Adapter =:= Contramap

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  ...
}
```

OOP
FP

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

API

```
showTitle("Hello", ???)
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

OOP

FP

API

```
showTitle("Hello", ???)
```

Implementation

```
def showTitle(title: String, color: (AnsiColor.type) => String) =
  s"======= ${color(AnsiColor)}${title}${RESET} ======="


trait AnsiColor {
  final val BLACK      = "\u001b[30m"
  final val RED        = "\u001b[31m"
  final val GREEN      = "\u001b[32m"
  final val YELLOW     = "\u001b[33m"
  final val BLUE       = "\u001b[34m"
  ...
}

object AnsiColor extends AnsiColor {}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

OOP

FP

API

```
showTitle("Hello", colors => colors.BLUE)
```

Implementation

```
def showTitle(title: String, color: (AnsiColor.type) => String) =
  s"======= ${color(AnsiColor)}${title}${RESET} ======="


trait AnsiColor {
  final val BLACK      = "\u001b[30m"
  final val RED        = "\u001b[31m"
  final val GREEN      = "\u001b[32m"
  final val YELLOW     = "\u001b[33m"
  final val BLUE       = "\u001b[34m"
  ...
}

object AnsiColor extends AnsiColor {}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

OOP

FP

API

```
showTitle("Hello", _.BLUE)
```

Implementation

```
def showTitle(title: String, color: (AnsiColor.type) => String) =
  s"======= ${color(AnsiColor)}${title}${RESET} ======="


trait AnsiColor {
  final val BLACK      = "\u001b[30m"
  final val RED        = "\u001b[31m"
  final val GREEN      = "\u001b[32m"
  final val YELLOW     = "\u001b[33m"
  final val BLUE       = "\u001b[34m"
  ...
}

object AnsiColor extends AnsiColor {}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

OOP

FP

API

```
showTitle("Hello", _.BLUE)
```

Implementation

```
def showTitle(title: String, color: (AnsiColor.type) => String) =
  s"======= ${color(AnsiColor)}${title}${RESET} ======="


trait AnsiColor {
  final val BLACK      = "\u001b[30m"
  final val RED        = "\u001b[31m"
  final val GREEN      = "\u001b[32m"
  final val YELLOW     = "\u001b[33m"
  final val BLUE       = "\u001b[34m"
  ...
}

object AnsiColor extends AnsiColor {}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  ...
}
```

API

```
showTitle("Hello", _.BLUE)
```

Implementation

```
def showTitle(title: String, color: (AnsiColor.type) => String) =
  s"======= ${color(AnsiColor)}${title}${RESET} ======="
```

OOP

FP

$$\lambda(x) \rightarrow userLogic(x)$$

is the most powerful strategy possible!

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  ...
}
```

API

```
showTitle("Hello", _.BLUE)
```

Implementation

```
def showTitle(title: String, color: (AnsiColor.type) => String) =
  s"======= ${color(AnsiColor)}${title}${RESET} ======="
```

OOP
FP

$$\lambda(x) \rightarrow userLogic(x)$$

is the most powerful strategy possible!

too powerful? **Do this...**

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

Hybrid Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
  case class Custom(colorFunc: (AnsiColor.type) => String)
    extends ColorStrategy
}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

Implementation

```
def showTitle(title: String, color: ColorStrategy): String = {
  val text = color match {
    case ColorStrategy.Red                => showTitle(title, Custom(_.RED))
    case ColorStrategy.Green              => showTitle(title, Custom(_.GREEN))
    case ColorStrategy.Blue               => showTitle(title, Custom(_.BLUE))
    case ColorStrategy.Custom(colorFunc)  =>
      s"${AnsiColor.BLUE}${colorFunc(AnsiColor)}${RESET}"
  }
  s"======= ${text} ======="
}
```

Hybrid Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
  case class Custom(colorFunc: (AnsiColor.type) => String)
    extends ColorStrategy
}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
}
```

API

```
println(
  showTitle("Hello", ColorStrategy.Blue) +
  showTitle("How are you", ColorStrategy.Custom(_.MAGENTA))
)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy): String = {
  val text = color match {
    case ColorStrategy.Red                 => showTitle(title, Custom(_.RED))
    case ColorStrategy.Green               => showTitle(title, Custom(_.GREEN))
    case ColorStrategy.Blue                => showTitle(title, Custom(_.BLUE))
    case ColorStrategy.Custom(colorFunc)   =>
      s"${AnsiColor.BLUE}${colorFunc(AnsiColor)}${RESET}"
  }
  s"======= ${text} ======="
}
```

Hybrid Strategy

```
sealed trait ColorStrategy
object ColorStrategy {
  case object Red extends ColorStrategy
  case object Green extends ColorStrategy
  case object Blue extends ColorStrategy
  case class Custom(colorFunc: (AnsiColor.type) => String)
    extends ColorStrategy
}
```

The Great **Strategy Pattern**

API

```
showTitle("Hello", ColorStrategy.Blue)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy) = {
  val text = color match {
    case ColorStrategy.Red   => s"${AnsiColor.RED}${title}${RESET}"
    case ColorStrategy.Green => s"${AnsiColor.GREEN}${title}${RESET}"
    case ColorStrategy.Blue  => s"${AnsiColor.BLUE}${title}${RESET}"
  }
  s"======= ${text} ======="
}
```

API

```
println(
  showTitle("Hello", ColorStrategy.Blue) +
  showTitle("How are you", ColorStrategy.Custom(_.MAGENTA))
)
```

Implementation

```
def showTitle(title: String, color: ColorStrategy): String = {
  val text = color match {
    case ColorStrategy.Red                => showTitle(title, Custom(_.RED))
    case ColorStrategy.Green              => showTitle(title, Custom(_.GREEN))
    case ColorStrategy.Blue               => showTitle(title, Custom(_.BLUE))
    case ColorStrategy.Custom(colorFunc)  =>
      s"${AnsiColor.BLUE}${colorFunc(AnsiColor)}${RESET}"
  }
  s"======= ${text} ======="
}
```

Strategy <:< Higher Order Function

The Great **Visitor Pattern**

User Code

```
data class Person(val name: Name, val age: Int)
data class Name(val first: String)

val p = Person(Name("Joe"), 123)
val v = CapturedBlock {
  p.name.first
}
```

Kotlin IR (expression tree)

```
IrCall(
  IrCall(
    IrGetValue("p")
    "name"
  ),
  "first"
)
```

The Great **Visitor Pattern**

User Code

```
data class Person(val name: Name, val age: Int)
data class Name(val first: String)

val p = Person(Name("Joe"), 123)
val v = CapturedBlock {
  p.name.first
}
```

Kotlin IR (expression tree)

```
IrCall(
  IrCall(
    IrGetValue("p")
    "name"
  ),
  "first"
)
```

Our IR (expression tree)

```
Ast.Property(
  Ast.Property(
    Ast.Ident("p")
    "name"
  ),
  "first"
)
```

The Great **Visitor Pattern**

User Code

```
data class Person(val name: Name, val age: Int)
data class Name(val first: String)

val p = Person(Name("Joe"), 123)
val v = CapturedBlock {
  p.name.first
}
```

Kotlin IR (expression tree)

```
IrCall(
  IrCall(
    IrGetValue("p")
    "name"
  ),
  "first"
)
```

Our IR

```
Ast.Property(
  Ast.Property(
    Ast.Ident("p")
    "name"
  ),
  "first"
)
```

```
interface IrElementVisitor<out R, in D> {
  fun visitElement(element: IrElement, data: D): R

  fun visitValueParameter(declaration: IrValueParameter, data: D): R
  fun visitClass(declaration: IrClass, data: D): R
  fun visitAnonymousInitializer(declaration: IrAnonymousInitializer, data: D): R

  fun visitCall(expression: IrCall, data: D): R
  fun visitGetValue(expression: IrGetValue, data: D): R

  ...
}
```

The Great **Visitor Pattern**

User Code

```
data class Person(val name: Name, val age: Int)
data class Name(val first: String)

val p = Person(Name("Joe"), 123)
val v = CapturedBlock {
  p.name.first
}
```

Kotlin IR (expression tree)

```
IrCall(
  IrCall(
    IrGetValue("p")
    "name"
  ),
  "first"
)
```

Our IR

```
Ast.Property(
  Ast.Property(
    Ast.Ident("p")
    "name"
  ),
  "first"
)
```

```
interface IrElementVisitor<out R, in D> {
  fun visitElement(element: IrElement, data: D): R

  fun visitValueParameter(declaration: IrValueParameter, data: D): R
  fun visitClass(declaration: IrClass, data: D): R
  fun visitAnonymousInitializer(declaration: IrAnonymousInitializer, data: D): R

  fun visitCall(expression: IrCall, data: D): R
  fun visitGetValue(expression: IrGetValue, data: D): R

  ...
}
```

The Great **Visitor Pattern**

User Code

```
data class Person(val name: Name, val age: Int)
data class Name(val first: String)

val p = Person(Name("Joe"), 123)
val v = CapturedBlock {
  p.name.first
}
```

Kotlin IR (expression tree)

```
IrCall(
  IrCall(
    IrGetValue("p")
    "name"
  ),
  "first"
)
```

Our IR

```
Ast.Property(
  Ast.Property(
    Ast.Ident("p")
    "name"
  ),
  "first"
)
```

```
interface IrElementVisitor<out R, in D> {
  fun visitElement(element: IrElement, data: D): R
  fun visitCall(call: IrCall, data: D): R = visitElement(call, data)
  fun visitGetValue(getValue: IrGetValue, data: D): R = visitElement(getValue, data)
  ...
}
```

```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

The Great **Visitor Pattern**

User Code

```
data class Person(val name: Name, val age: Int)
data class Name(val first: String)

val p = Person(Name("Joe"), 123)
val v = CapturedBlock {
  p.name.first
}
```

Kotlin IR (expression tree)

```
IrCall(
  IrCall(
    IrGetValue("p")
    "name"
  ),
  "first"
)
```

Our IR

```
Ast.Property(
  Ast.Property(
    Ast.Ident("p")
    "name"
  ),
  "first"
)
```

```
interface IrElementVisitor<out R, in D> {
  fun visitElement(element: IrElement, data: D): R
  fun visitCall(call: IrCall, data: D): R = visitElement(call, data)
  fun visitGetValue(getValue: IrGetValue, data: D): R = visitElement(getValue, data)
}
```

```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult =
    expr.accept(ParserVisitor, Unit)
```

The Great **Visitor Pattern**

User Code

```
data class Person(val name: Name, val age: Int)
data class Name(val first: String)

val p = Person(Name("Joe"), 123)
val v = CapturedBlock {
  p.name.first
}
```

Kotlin IR (expression tree)

```
IrCall(
  IrCall(
    IrGetValue("p")
    "name"
  ),
  "first"
)
```

Our IR

```
Ast.Property(
  Ast.Property(
    Ast.Ident("p")
    "name"
  ),
  "first"
)
```

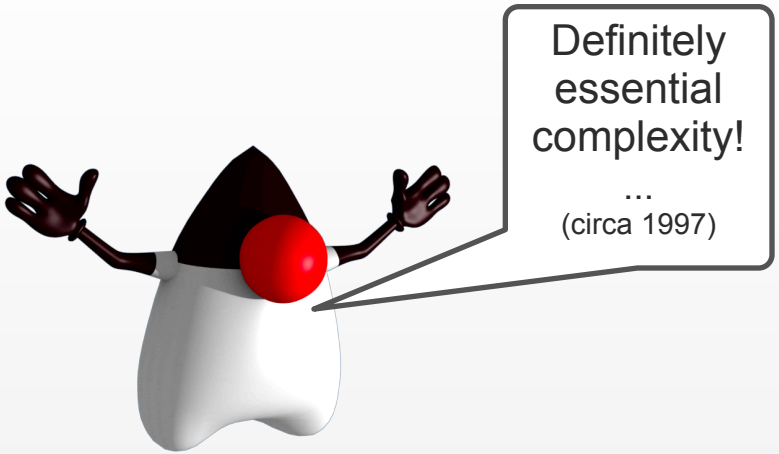
```
interface IrElementVisitor<out R, in D> {
  fun visitElement(element: IrElement, data: D): R
  fun visitCall(call: IrCall, data: D): R = visitElement(call, data)
  fun visitGetValue(getValue: IrGetValue, data: D): R = visitElement(getValue, data)
}
```

```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult: Ast =
    expr.accept(ParserVisitor, Unit)
```

The Great **Visitor Pattern**



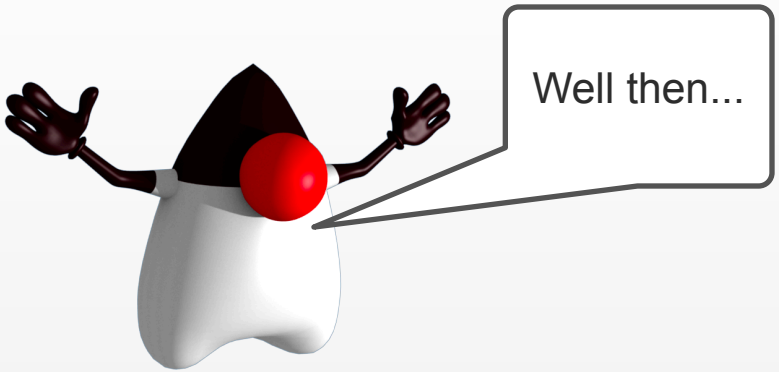

```
interface IrElementVisitor<out R, in D> {
  fun visitElement(element: IrElement, data: D): R

  fun visitValueParameter(declaration: IrValueParameter, data: D): R
  fun visitClass(declaration: IrClass, data: D): R
  fun visitAnonymousInitializer(declaration: IrAnonymousInitializer, data: D): R

  fun visitCall(expression: IrCall, data: D): R
  fun visitGetField(expression: IrGetField, data: D): R

  ...
}
```

Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult =
    expr.accept(ParserVisitor, Unit)
```

```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

The Great **Visitor Pattern**

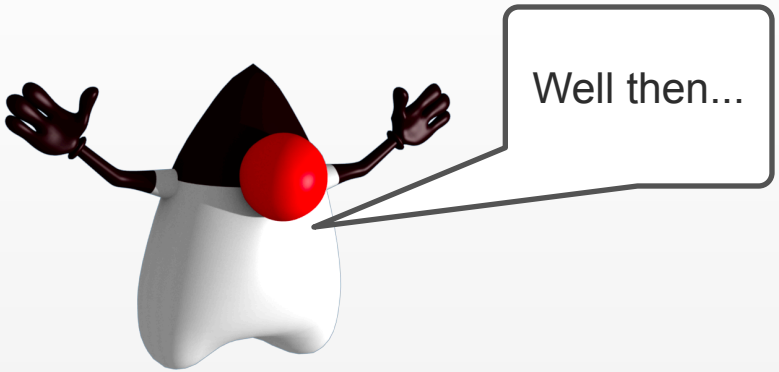


Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult: Ast =
    parse(expr)
```

```
def parse(expr: IrElement) =
  expr match {
    case IrGetValue(name) =>
      Ident(name)
    case IrCall(Some(expr), IrValueSymbol(name)) =>
      Property(parse(expr), name)
  }
```

The Great **Visitor Pattern**




```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

OOP

Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult =
    expr.accept(ParserVisitor, Unit)
```

```
def parse(expr: IrElement) =
  expr match {
    case IrGetValue(name) =>
      Ident(name)
    case IrCall(Some(expr), IrValueSymbol(name)) =>
      Property(parse(expr), name)
  }
```

The Great **Visitor Pattern**

JEP440: Record Patterns in your face!
(circa 2023)

```
static Ast parse(IrElement expr) {
  switch (expr) {
    case IrGetValue(name)   ->
      Ident(name)
    case IrCall(IrExpr(_, _), IrValueSymbol(name)) ->
      Property(parse(expr), name)
  }
}
```

```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult =
    expr.accept(ParserVisitor, Unit)
```

```
def parse(expr: IrElement) =
  expr match {
    case IrGetValue(name) =>
      Ident(name)
    case IrCall(Some(expr), IrValueSymbol(name)) =>
      Property(parse(expr), name)
  }
```

The Great **Visitor Pattern**

JEP443: Unnamed Patterns and Variables

(circa Java 21 Preview)

```
static Ast parse(IrElement expr) {
  switch (expr) {
    case IrGetValue(name) ->
      Ident(name)
    case IrCall(IrExpr(_, _), IrValueSymbol(name)) ->
      Property(parse(expr), name)
  }
}
```

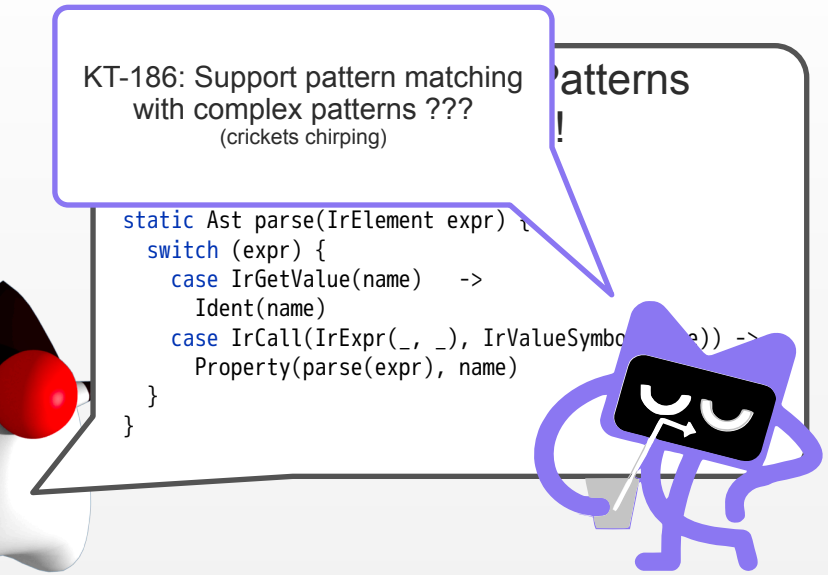
```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult =
    expr.accept(ParserVisitor, Unit)
```

```
def parse(expr: IrElement) =
  expr match {
    case IrGetValue(name) =>
      Ident(name)
    case IrCall(Some(expr), IrValueSymbol(name)) =>
      Property(parse(expr), name)
  }
```

The Great **Visitor Pattern**




```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

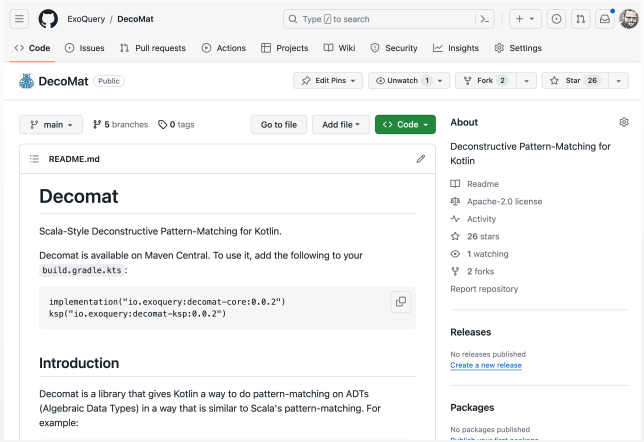
Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult =
    expr.accept(ParserVisitor, Unit)
```

```
def parse(expr: IrElement) =
  expr match {
    case IrGetValue(name) =>
      Ident(name)
    case IrCall(Some(expr), IrValueSymbol(name)) =>
      Property(parse(expr), name)
  }
```

The Great **Visitor Pattern**

Decomat Library for Kotlin



Library Entry Point

```
fun entryPoint(expr: IrExpression): String =
  val parseResult =
    expr.accept(ParserVisitor, Unit)
```

```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

```
def parse(expr: IrElement) =
  on(expr).match<Ast>(
    case(Ir.GetValue[Is()]).then { (name) ->  Ident(name) },
    case(Ir.Call[Is(), Ir.ValueSymbol[Is()]]).then { expr, name ->
      Property(parse(expr), name)
    }
  )
```

The Great **Visitor Pattern**

```
def parse(expr: IrElement) =
  expr match {
    case IrGetValue(name) =>
      Ident(name)
    case IrCall(Some(expr), IrValueSymbol(name)) =>
      Property(parse(expr), name)
  }
```

FP

```
def parse(expr: IrElement) =
  on(expr).match<Ast>(
    case(Ir.GetValue[Is()]).then { (name) ->  Ident(name) },
    case(Ir.Call[Is(), Ir.ValueSymbol[Is()]]).then { expr, name ->
      Property(parse(expr), name)
    }
  )
```

FP

```
object ParserVisitor: IrElementVisitor<Ast, Unit> {
    override fun visitElement(element: IrElement, data: Unit) =
        Fail.noVisitRoot()

    override fun visitCall(call: IrCall, data: Unit): Ast {
        val caller = call.dispatchReceiver?.accept(this, data) ?: Fail()
        val arg = call.symbol.safeName
        return Property(caller, arg)
    }

    override fun visitGetValue(getValue: IrGetValue, data: Unit): Ast =
        Ident(getValue.symbol.safeName)
}
```

OOP

Visitor <🤬< Recursive Pattern Match

Conclusions

Various **Pathodental Lies**

~~Boilerplate is Always Bad~~ 01

~~Functional Programming has no Design Patterns~~ 02

Various **Realities**

Boilerplate can significantly increase maintainability 01

Functional Programming has use-case specific design patterns (and we should talk about them) 02

Various other **Pathodental Lies**

Boilerplate can significantly increase maintainability 01

Functional Programming has use-case specific design patterns (and we should talk about them) 02

for-comprehensions / computation expressions are Referentially Transparent 03

Scala 3 Generic Derivation requires no knowledge of macros 04

Spring is Easy 05

All complexity is **intrinsic**
until you figure out how to **do it better**!

This Presentation

https://github.com/deusaquilus/fs2023

Scala-Style Deconstructive Pattern-Matching for Kotlin.

https://github.com/ExoQuery/DecoMat

```
someone match {
  case Customer(Name(first @ "Joe", last), Partner(id)) =>
    func(first, last, id)
  case Customer(Name(first @ "Jack", last), Organization("BigOrg")) =>
    func(first, last)
}
```

```
on(someone).match(
  case( Customer[Name[Is("Joe"), Is()], Partner[Is()]] )
    .then { first, last, id -> func(first, last, id) },
  case( Customer[Name[Is("Jack"), Is()], Organization[Is("BigOrg")]] )
    .then { first, last -> func(first, last) }
)
```

Upcoming changes in Quill and ProtoQuill

https://github.com/zio/zio-quill

https://github.com/zio/zio-protoquill