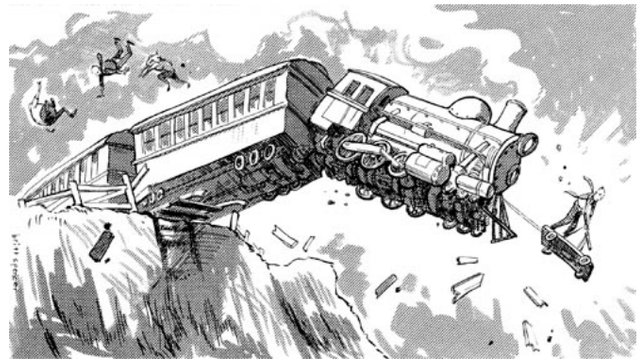
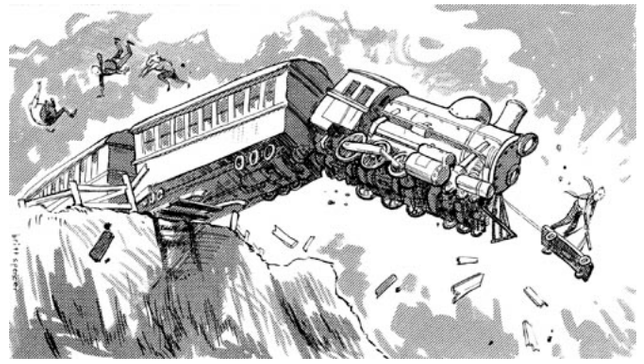
Life in a Post-Functional World

Definitions



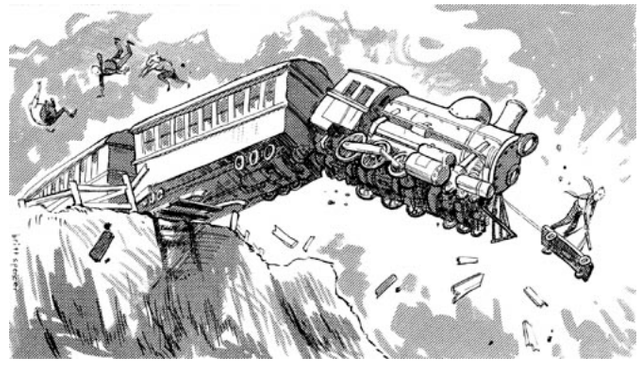
- Functional Programming

Definitions



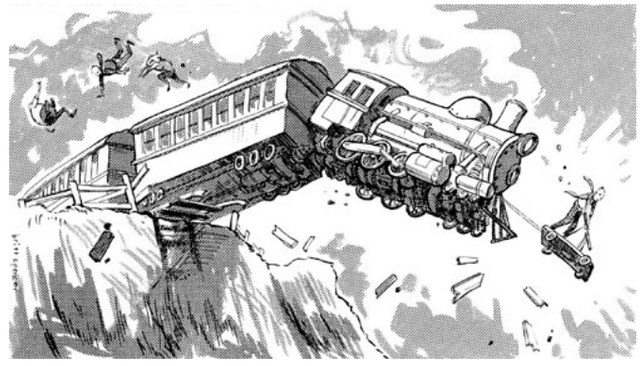
- Functional Programming
  - Programming…with functions!

Definitions



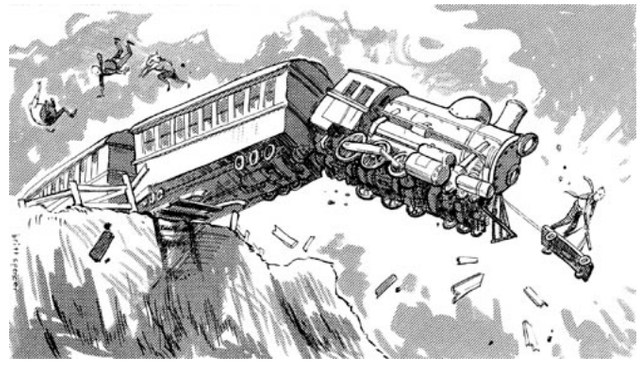
- Functional Programming
  - ~~Programming…with functions!~~
  - Functions as primary abstraction unit

Definitions



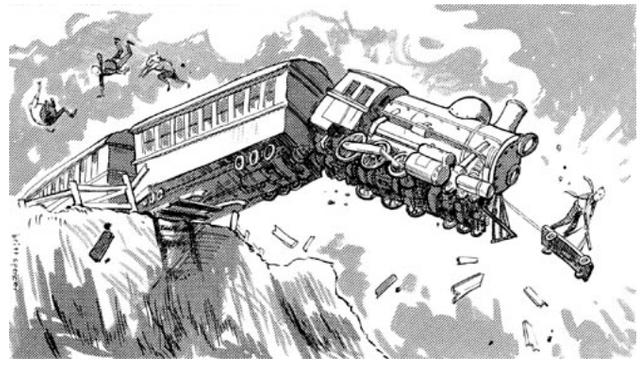
- Functional Programming
  - ~~Programming…with functions!~~
    - Functions as primary abstraction unit
- Object-Oriented Programming

Definitions



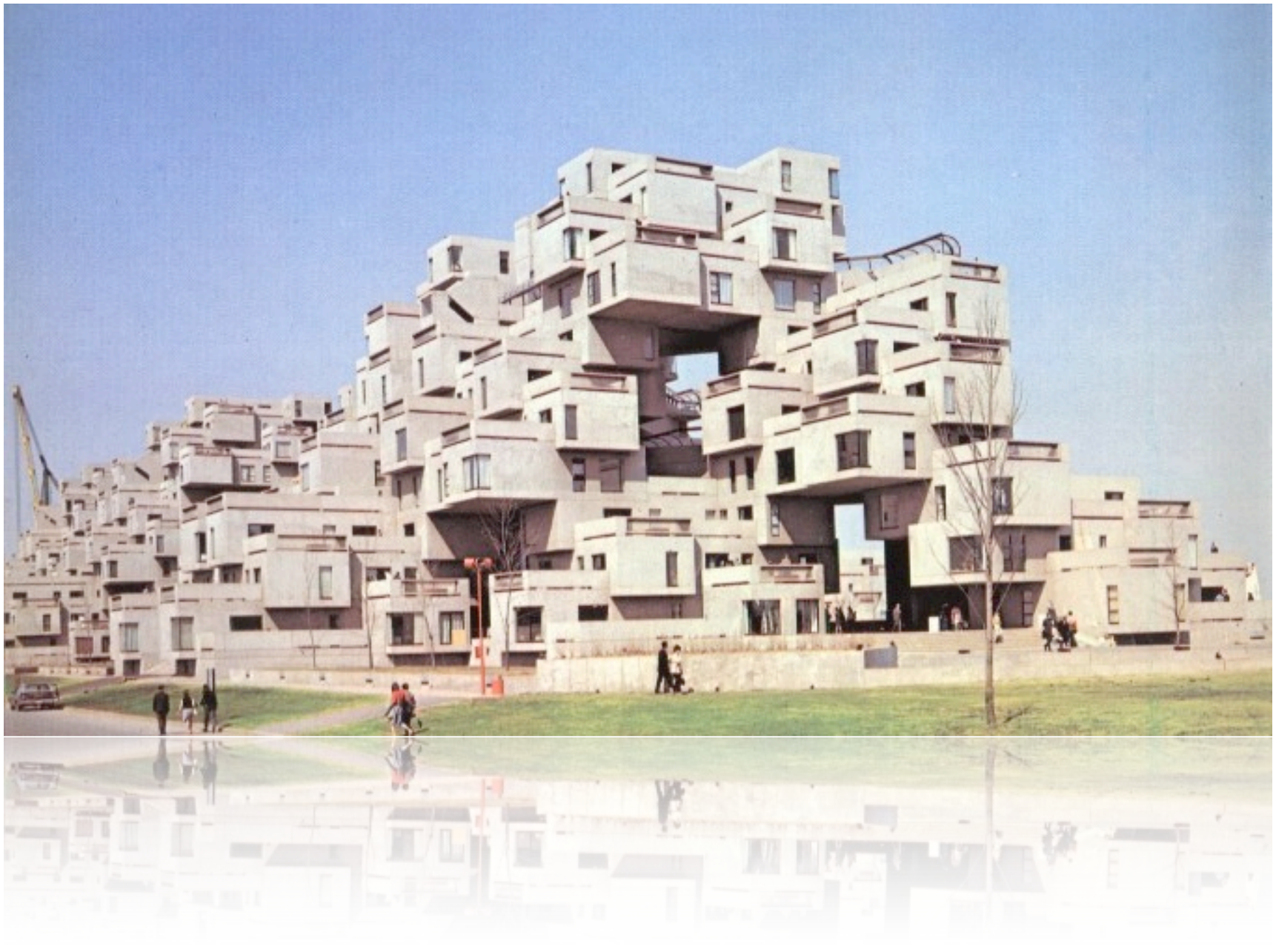
- Functional Programming
  - ~~Programming…with functions!~~
    - Functions as primary abstraction unit
- Object-Oriented Programming
  - Solipsisms

Definitions



- Functional Programming
  - ~~Programming…with functions!~~
    - Functions as primary abstraction unit
- Object-Oriented Programming
  - ~~Solipsisms~~
  - Objects as primary abstraction unit

Modules



```
def deviation(ints: List[Int]) = {
  def meanOpt(ints: List[Int]) = {
    val sumOpt = ints reduceOption { _ + _ }
    sumOpt map { _ / ints.length }
  }

  meanOpt(ints) flatMap { mean =>
    meanOpt(ints map { _ - mean })
  }
}
```

Scaling

- Real systems have many such functions

Scaling

- Real systems have many such functions
- A *set* of functions is not a function!

Scaling

- Real systems have many such functions
- A *set* of functions is not a function!
- We need a first-class organizational unit

Scaling

- Real systems have many such functions
- A *set* of functions is not a function!
- We need a first-class organizational unit
- Reason about sets of functions

Real Modules

- Namespacing alone is not sufficient

Real Modules

- Namespacing alone is not sufficient
- Collision is a symptom, not the problem!

Real Modules

- Namespacing alone is not sufficient
  - Collision is a symptom, not the problem!
- Dependencies should be expressed

Real Modules

- Namespacing alone is not sufficient
  - Collision is a symptom, not the problem!
- Dependencies should be expressed
- Concrete binding should be deferred

SML

- Signatures are abstract modules

SML

- Signatures are abstract modules
  - Functions

SML

- Signatures are abstract modules
  - Functions
  - Types

SML

- Signatures are abstract modules
  - Functions
  - Types
- Structures implement signatures

SML

- Signatures are abstract modules
  - Functions
  - Types
- Structures implement signatures
- Functors allow module dependencies

SML

- Signatures are abstract modules
  - Functions
  - Types
- Structures implement signatures
- Functors allow module dependencies
  - Note: not Haskell's *applicative functor*

```
signature QUEUE =
sig
  type 'a queue
  val empty     : 'a queue
  val isEmpty   : 'a queue -> bool
  val singleton : 'a -> 'a queue
  val insert    : 'a * 'a queue -> 'a queue
  val peek      : 'a queue -> 'a
  val remove    : 'a queue -> 'a * 'a queue
end
```

```
structure ListQueue :> QUEUE =
struct
  type 'a queue = 'a list * 'a list

  val empty = ([], [])

  fun isEmpty ([], []) = true
    | isEmpty _ = false

  (* ... *)
end
```

```
functor STUFF (Q: QUEUE) =
struct
  fun foo (q: int Q.queue) =
    Q.isEmpty q
end

structure Cake = STUFF (ListQueue)

Cake.foo ([], [42])
```

Scala

- Traits are modules

Scala

- Traits are modules
- Type abstraction

Scala

- Traits are modules
- Type abstraction
- Dependencies via inheritance

Scala

- Traits are modules
- Type abstraction
- Dependencies via inheritance
- Instantiated traits are values of kind $*$

```scala
trait QueueModule {
  type Queue[+A] <: QueueLike[A]

  def Queue[A](a: A*): Queue[A]

  trait QueueLike[+A] {
    def isEmpty: Boolean
    def insert[B >: A](b: B): Queue[B]
    def peek: A
    def remove: (A, Queue[A])
  }
}
```

```
trait ListQueueModule extends QueueModule {
  def Queue[A](a: A*) =
    new Queue(a.toList, Nil)

  class Queue[+A](front: List[A],
                  back: List[A])
      extends QueueLike[A] {

    def isEmpty =
      front.isEmpty && back.isEmpty

    // ...
  }
}
```

```
trait Stuff extends QueueModule {
  def foo(q: Queue[Int]) = q.isEmpty
}
```

```
val cake = new Stuff with ListQueueModule {}
cake.foo(cake.Queue(42))
```

```
def foo(QM: QueueModule)(q: QM.Queue[Int]) =
  q.isEmpty
```

```
val cake = new ListQueueModule {}
foo(cake)(cake.Queue(42))
```

Noteworthy

- In SML, the function is first-class

Noteworthy

- In SML, the function is first-class
- In Scala, the *module* is first-class
  - Functions are modules

Noteworthy

- In SML, the function is first-class
- In Scala, the *module* is first-class
  - Functions are modules
- Opens up a whole realm of awesome!

```scala
trait Stuff {
  def foo(i: Int): Int
}

def build(f: Int => Int) = new Stuff {
  def foo(i: Int) = f(i)
}
```

Objects

Expression Problem

- Define a datatype by cases

Expression Problem

- Define a datatype by cases
- Add new cases to the datatype

Expression Problem

- Define a datatype by cases
- Add new cases to the datatype
- Add new functions over the datatype

Expression Problem

- Define a datatype by cases
- Add new cases to the datatype
- Add new functions over the datatype
- Don’t recompile!

Expression Problem

- Define a datatype by cases
- Add new cases to the datatype
- Add new functions over the datatype
- Don’t recompile!

(good luck!)

```scala
sealed trait Expr
case class Add(e1: Expr, e2: Expr) extends Expr
case class Sub(e1: Expr, e2: Expr) extends Expr
case class Num(n: Int) extends Expr

def value(e: Expr): Int = e match {
  case Add(e1, e2) =>
    value(e1) + value(e2)

  case Sub(e1, e2) =>
    value(e1) - value(e2)

  case Num(n) => n
}
```

Pattern Matching

- "Dumb" cases (*i.e.* empty case classes)

Pattern Matching

- "Dumb" cases (*i.e.* empty case classes)
- Every function enumerates full algebra

Pattern Matching

- "Dumb" cases (*i.e.* empty case classes)
- Every function enumerates full algebra
- Very easy to add new functions

Pattern Matching

- "Dumb" cases (*i.e.* empty case classes)
- Every function enumerates full algebra
- Very easy to add new functions
- Very difficult to add new cases

```scala
sealed trait Expr {
  def value: Int
}

case class Add(e1: Expr, e2: Expr) extends Expr {
  def value = e2.value + e2.value
}

case class Sub(e1: Expr, e2: Expr) extends Expr {
  def value = e1.value - e2.value
}

case class Num(n: Int) extends Expr {
  def value = n
}
```

Subtype Polymorphism

- "Smart" cases (*i.e.* non-empty classes)

Subtype Polymorphism

- "Smart" cases (*i.e.* non-empty classes)
- Every case enumerates all functions

Subtype Polymorphism

- "Smart" cases (*i.e.* non-empty classes)
- Every case enumerates all functions
- Very easy to add new cases

Subtype Polymorphism

- "Smart" cases (*i.e.* non-empty classes)
- Every case enumerates all functions
- Very easy to add new cases
- Very difficult to add new functions

Expression Problem

- Neither answer is correct!

Expression Problem

- Neither answer is correct!
- There is no "solution" to the problem

Expression Problem

- Neither answer is correct!
- There is no "solution" to the problem
- We need to pick and choose our technique

Expression Problem

- Neither answer is correct!
- There is no "solution" to the problem
- We need to pick and choose our technique
- Objects are essential!

Expression Problem

- Neither answer is correct!
- There is no "solution" to the problem
- We need to pick and choose our technique
- Objects are essential!
  - …or some near approximation

Expression Problem

- Neither answer is correct!
- There is no "solution" to the problem
- We need to pick and choose our technique
- Objects are essential!
  - …or some near approximation
- Pattern matching is also essential

Clojure

- Protocols unify typeclasses and classes
  - (sort of)
  - See: *Oliveira and Sulzmann, 2008*

Clojure

- Protocols unify typeclasses and classes
  - (sort of)
    - See: *Oliveira and Sulzmann, 2008*
- Originally done for performance reasons

Clojure

- Protocols unify typeclasses and classes
  - (sort of)
    - See: *Oliveira and Sulzmann, 2008*
- Originally done for performance reasons
- A little OO in your Clojure!

```
(defprotocol Eval
  (expr-value [this]))

(defrecord Add [e1 e2]
  Eval
  (expr-value [this]
    (+
      (expr-value e1)
      (expr-value e2))))
```

```
(defrecord Sub [e1 e2]
  Eval
  (expr-value [this]
    (-
      (expr-value e1)
      (expr-value e2))))

(defrecord Num [n]
  Eval
  (expr-value [this] n))
```

```
(expr-value (Add. (Num. 4) (Num. 5)))
```

Protocols

- A special case of multi-methods
- Part of the record definition!
- Support for all sorts of OO evil
- …and more besides!

```
(defprotocol StringFun
  (palindrome? [this]))

(extend-type String
  StringFun
  (palindrome? [this]
    (=
      this
      (apply str (reverse this)))))


(println (palindrome? "test"))
(println (palindrome? "madamimadam"))
```

Conclusion

- Modules are vital #haskellfail

Conclusion

- Modules are vital #haskellfail
- Objects help address expression problem

Conclusion

- Modules are vital #haskellfail
- Objects help address expression problem
- Not *everything* OO/procedural is evil

Conclusion

- Modules are vital #haskellfail
- Objects help address expression problem
- Not *everything* OO/procedural is evil
- Traditional notions of "FP" are naive

Conclusion

- Modules are vital #haskellfail
- Objects help address expression problem
- Not *everything* OO/procedural is evil
- Traditional notions of "FP" are naive
  - …but a *really* good starting point!

Questions?