SSCI 2015 Tutorial
http://github.com/cirg-up/cilib
G. Pamparà and A.P. Engelbrecht

Please interrupt at any time if you have questions
Things we fight, daily
Things we fight, daily

Programming is all about managing finite processing resources and trying to describe solutions to possibly complex problems.
Programming is all about managing finite processing resources and trying to describe solutions to possibly complex problems.

Things that add complexity:
Things we fight, daily

Programming is all about managing finite processing resources and trying to describe solutions to possibly complex problems.

Things that add complexity:

- Parallelism, Threading and concurrency
Things we fight, daily

Programming is all about managing finite processing resources and trying to describe solutions to possibly complex problems.

Things that add complexity:

- Parallelism, Threading and concurrency
- Managing application state
Things we fight, daily

Programming is all about managing finite processing resources and trying to describe solutions to possibly complex problems.

Things that add complexity:

- Parallelism, Threading and concurrency
- Managing application state
- Complex object interactions
Things we fight, daily

Programming is all about managing finite processing resources and trying to describe solutions to possibly complex problems.

Things that add complexity:

- Parallelism, Threading and concurrency
- Managing application state
- Complex object interactions
- Different kinds of OO semantics (X and Y have similar but different OO implementations)
Things we fight, daily

Programming is all about managing finite processing resources and trying to describe solutions to possibly complex problems.

Things that add complexity:

- Parallelism, Threading and concurrency
- Managing application state
- Complex object interactions
- Different kinds of OO semantics (X and Y have similar but different OO implementations)
- Pseudo-randomness on computers
Things we fight, daily

Programming is all about managing finite processing resources and trying to describe solutions to possibly complex problems.

Things that add complexity:

- Parallelism, Threading and concurrency
- Managing application state
- Complex object interactions
- Different kinds of OO semantics (X and Y have similar but different OO implementations)
- Pseudo-randomness on computers
- Discrepancies between algorithm descriptions (within and out of published works) and the actual implementations.
99 little bugs in the code.
99 little bugs in the code.
Take one down, patch it around.

127 little bugs in the code...
What do we want?
We want:

- A way to talk about algorithms and their implementations, without necessarily troubling ourselves with the complexities that make implementation difficult.
We want:

- A way to talk about algorithms and their implementations, without necessarily troubling ourselves with the complexities that make implementation difficult.
- We want a declarative process to describe things, worrying about how to “do” them at a later stage.
Cllib: Making sense of complexity

Cllib is a library designed to help with CI research by providing a base set of functionality to make usage simpler.
Clib: Making sense of complexity

Clib is a library designed to help with CI research by providing a base set of functionality to make usage simpler.

Furthermore, we want to make implementation as simple as possible by removing the boring parts that usually take up far too much time and are tricky to get right.
Clib: Making sense of complexity

Clib is a library designed to help with CI research by providing a base set of functionality to make usage simpler.

Furthermore, we want to make implementation as simple as possible by removing the boring parts that usually take up far too much time and are tricky to get right.

Several things, however, are very important to get right:
Clib: Making sense of complexity

Clib is a library designed to help with CI research by providing a base set of functionality to make usage simpler.

Furthermore, we want to make implementation as simple as possible by removing the boring parts that usually take up far too much time and are tricky to get right.

Several things, however, are very important to get right:

- **Correctness**
  - *Principled* design to promote intuitive usage

- **Experimental reproducability**
  - Not being able to reproduce experimental results is utterly useless

- **Type safety**
  - The ability to exploit types to prevent errors by making the construction of error states impossible
Clib: Design changes

Library first: No assumptions are made about how the code in this project is used
Cllib: Design changes

Library first: No assumptions are made about how the code in this project is used

Reliance on pre-existing libraries to simplify implementations:

- Scalaz: FP abstractions (lets see what happens with this and Cats)
- Spire: Numeric abstractions
- Monocle: Lens library
Clib: Design changes

Library first: No assumptions are made about how the code in this project is used

Reliance on pre-existing libraries to simplify implementations:

- Scalaz: FP abstractions (lets see what happens with this and Cats)
- Spire: Numeric abstractions
- Monocle: Lens library

Huge benefits:

- User has choice of persistence for results: memory, files, database or cloud storage
- Portions of library can simply be ignored if not needed
- Greater focus on algorithms
- Focused tests
- Opt-in parallelism
About Scala

We decided on Scala for a very simple reason: familiarity with the JVM. It's not that different from Java (on the runtime that is)

Scala features:
About Scala

We decided on Scala for a very simple reason: familiarity with the JVM. It’s not that different from Java (on the runtime that is)

Scala features:

- Object-orientated / Functional hybrid language
About Scala

We decided on Scala for a very simple reason: familiarity with the JVM. It's not **that** different from Java (on the runtime that is)

Scala features:

- Object-orientated / Functional hybrid language
- Static typing
About Scala

We decided on Scala for a very simple reason: familiarity with the JVM. It's not *that* different from Java (on the runtime that is)

Scala features:

- Object-orientated / Functional hybrid language
- Static typing
- Supports local type inference
About Scala

We decided on Scala for a very simple reason: familiarity with the JVM. It's not *that* different from Java (on the runtime that is)

**Scala features:**

- Object-orientated / Functional hybrid language
- Static typing
- Supports local type inference
- Extensible (e.g: syntax)
About Scala

We decided on Scala for a very simple reason: familiarity with the JVM.
It's not **that** different from Java (on the runtime that is)

Scala features:

- Object-orientated / Functional hybrid language
- Static typing
- Supports local type inference
- Extensible (e.g.: syntax)
- Allows for symbolic method names, including UTF-8

```scala
def Ꜩ (x: String) =
"This is probably not a good idea, but it's completely valid, " + x
```
Quick intro to Scala syntax

There are some difference in the syntax between Java and Scala, but these are in truth, minimal. All types are defined after variable names

```scala
def foo(first: String, last: String): Person =
...
```

Everything is an expression, which yields a value. In blocks, the last expression is the value of the entire block

```scala
val x: Int = {
  println("I'm a statement that returns Unit")
  List(1,2,3)
  4
}
```
**Syntax: Class definitions**

Java

```java
public class Person {
    private final String first;
    private final String last;

    public Person(String name, String last) {
        this.first = first;
        this.last = last;
    }

    public String getFirst() {
        return this.first;
    }

    public String getLast() {
        return this.last;
    }
}
```

Scala

```scala
case class Person(
    first: String, last: String)
```

There is always a primary constructor in Scala. This constructor is the class definition.

Auxiliary constructors are required to call the primary constructor.
Syntax: General

- Everything is an object and has a top type of Any, with a bottom type of Nothing
- The end of line terminator:
  - ; is not needed as the normal EOL character is \n
- Multiple expressions on the same line, however, use ; to separate individual expressions

- Linearization is used for multiple inheritance

```scala
class Dog extends Animal
  with Loyalty
  with Woofing
  with Mammal
```

- Variance annotations:
  - List[+A] - +A indicates covariance (i.e: subtypes)
  - Array[A] - A indicates the type is invariant
  - class Contravariant[-A] - -A indicates contravariance (i.e: supertypes)
• The normal OO style access modifiers (*private, protected, public*) exist, plus a few extra
• The default access modifier is *public*
• Immutable values are defined using *val*. E.g: *val* `x = 3`
• Mutable variables are defined using *var*
• Methods are defined using *def* `foo()`
• There are several uses for `_`:
  ○ *Imports*: `import scala.collection._`
  ○ *Hiding imports*: `import scala.Prelude.{ any2stringadd => _, _ }`
  ○ *Placeholder syntax*: `x => (_, Int) + x + 3`
  ○ *Ignoring a value*: `val (x, _) = tuple`
  ○ *Catch-all in pattern match*
Scala has **traits**, which are like Java interfaces but can contain concrete method implementations.

```scala
trait Adder {
  def add(x: Int, y: Int) = x + y
}
```

```scala
trait T[A] {
  def foo(a: A): String
}
```

Trait like interfaces now exist in Java 8, but it's not quite enough
• Multiple parameter groups are completely valid: `def foo(a: Int)(b: Int)(c: Int): Double`

• *Implicit parameters* are also allowed. These are parameters that the user need not provide, the compiler will insert the "nearest" scoped implicit value. Must be the last parameter group:

  `def bar(x: String)(implicit X: Monoid[String]) = ...`
Syntax: General (5)

An important tool we need is parametricity

Type parameters can be added to `class`, `def` and type aliases: `type C[A] = List[A]`
Syntax: General (5)

An important tool we need is parametricity

Type parameters can be added to `class`, `def` and type aliases: `type C[A] = List[A]

Generic programming means we can write less code to do more
Syntax: General (5)

An important tool we need is parametricity

Type parameters can be added to `class`, `def` and type aliases: `type C[A] = List[A]`

Generic programming means we can write less code to do more

Scala also allows us to abstract over `type constructors`, which is also known as `higher-kinded polymorphism`

```scala
def fooHigher[F[_],A](fa: F[A]) =
...
```

The above function works for types like `List`, `Set`, `Vector`, but not for `Map[A,B]`
Syntax: Pattern matching

Functional languages allow for "pattern matching", whereby a data structure can be deconstructed, without the need for reflection and in a way that the compiler can type-check for you.

Pattern matching uses the `match` syntax:

```scala
scala> List(1,2,3) match {
      | case x :: xs => 4
      | }
<console>:8: warning: match may not be exhaustive.
It would fail on the following input: Nil
List(1,2,3) match {
      ^
res0: Int = 4
```

Just note that the first matching case in the list of patterns will be used. We'll look at pattern matching in depth when we discuss Algebraic Data Types (ADTs)
REPL: Read Evaluate Print Loop

Scala, like most functional languages, has a REPL within which experimentation can happen without needing to create an entire compiling program. It aids thinking dramatically!
Functions
The ultimate tool
Functions: First class values

Functions are first class values in Scala, meaning that they can be passed around as you would pass around an `Int`.

```scala
val f: Int => String = (x: Int) => x.toString
```

Functions can be returned as a result and/or can be passed in as input.

```scala
def foo(x: Int => String): Int => List[Int] =
...
```

Functions that take and return functions are called Higher Order Functions (HOF).
Functions: Partial application

Functions of many parameters may have some of these parameters bound to values, resulting in a new function that requires the remaining parameters only.

This is known as *partial function application*

```scala
// Using methods, this is the way
def add(x: Int)(y: Int) = x + y // Multiple parameter groups
def add5 = add(5)(_)

// Using functions
val add = (x: Int) => (y: Int) => x + y
val add5 = add(5)
```

It should be obvious that there is a similarity between the forms. Note that the latter is more flexible.
Functions: Curried and uncurried

Currying (named after Haskell Curry) is the process where a function of many arguments is transformed into a sequence of functions, taking a single argument each.

Uncurrying is the “dual” process of currying, whereby a curried function is transformed into a function taking a tuple of arguments.

```plaintext
val curried: Int => Int => Int => Double =
    x => y => z => (x + y + z) / 0.5

val uncurried: (Int, Int, Int) => Double =
    (x, y, z) => (x + y + z) / 0.5
```
Functions: Curried and uncurried

Currying (named after Haskell Curry) is the process where a function of many arguments is transformed into a sequence of functions, taking a single argument each.

Uncurrying is the "dual" process of currying, whereby a curried function is transformed into a function taking a tuple of arguments.

```scala
val curried: Int => Int => Int => Double =
  x => y => z => (x + y + z) / 0.5

val uncurried: (Int, Int, Int) => Double =
  (x, y, z) => (x + y + z) / 0.5
```

Functions in Scala are right associative, so the curried function `Int => Int => Double` is isomorphic (exactly the same) as `Int => (Int => Double)`, where the latter has parenthesis applied to make the binding explicit.
Algebraic Data Types (ADTs)

You already know a number of types, such as: Int, Double, List[Int]

Composite types are created in two forms, namely Product or Sum types

Product types multiply types together, producing types that are the combination of the sub types. Examples are: normal classes, tuples and records

Sum types are also called disjoint types or discriminated unions. Example: Either[String, Int]
Algebraic Data Types (ADTs)

You already know a number of types, such as: `Int`, `Double`, `List[Int]`

Composite types are created in two forms, namely **Product** or **Sum** types

Product types multiply types together, producing types that are the combination of the sub types. Examples are: normal classes, tuples and records

Sum types are also called disjoint types or discriminated unions. Example: `Either[String, Int]`

ADTs are known as "algebraic" types because they form a **closed** algebra which cannot be altered
ADTs: Definition

Languages such as Haskell support the creation of ADTs directly in the language. Scala does support them, but the encoding is done using `case` classes / objects and `sealed`

```scala
sealed trait Option[A]
case class Some[A](a: A) extends Option[A]
case object None extends Option[Nothing]
```
ADTs: Definition

Languages such as Haskell support the creation of ADTs directly in the language. Scala does support them, but the encoding is done using case classes / objects and sealed

```scala
sealed trait Option[A]
case class Some[A](a: A) extends Option[A]
case object None extends Option[Nothing]
```

There is a **VERY** important distinction here that we need to make upfront:

Because of how Scala encodes ADTs, using subtyping, direct instantiating of `Some` and `None` are possible because Scala regards them as unique types themselves. These instances are of the types `Some.type` and `None.type` but they really should be of the type: `Option[A]`
ADTs: Definition

Languages such as Haskell support the creation of ADTs directly in the language. Scala does support them, but the encoding is done using case classes / objects and sealed

```scala
sealed trait Option[A]
case class Some[A](a: A) extends Option[A]
case object None extends Option[Nothing]
```

There is a **VERY** important distinction here that we need to make upfront:

Because of how Scala encodes ADTs, using subtyping, direct instantiating of Some and None are possible because Scala regards them as unique types themselves. These instances are of the types Some.type and None.type but they really should be of the type: Option[A]

When using ADT structures, like Option you always talk about things being Option and not a Some / None. Some(...) and None are called "data constructors" as they create data, but the type is Option
ADTs: Why use them?

ADTs allow us to have a fixed set of operations.

They cannot be extended further and encode something specific into the type system.

ADTs are what is missing in many languages, which then introduce more language features to try address this missing feature. ADTs are **simple values** and **not** language level semantics.

Examples of language "things" added due to missing ADTs:
ADTs: Why use them?

ADTs allow us to have a fixed set of operations.

They cannot be extended further and encode something specific into the type system.

ADTs are what is missing in many languages, which then introduce more language features to try address this missing feature. ADTs are simple values and not language level semantics.

Examples of language "things" added due to missing ADTs:

- Missing values -> the billion dollar mistake: null
ADTs: Why use them?

ADTs allow us to have a fixed set of operations

They cannot be extended further and encode something specific into the type system

ADTs are what is missing in many languages, which then introduce more language features to try address this missing feature. ADTs are simple values and not language level semantics

Examples of language "things" added due to missing ADTs:

- Missing values -> the billion dollar mistake: null
- Descriptive errors -> exceptions
ADTs: Why use them?

ADTs allow us to have a fixed set of operations
They cannot be extended further and encode something specific into the type system
ADTs are what is missing in many languages, which then introduce more language features to try address this missing feature. ADTs are simple values and not language level semantics

Examples of language "things" added due to missing ADTs:

- Missing values -> the billion dollar mistake: null
- Descriptive errors -> exceptions
- Comparison semantics -> Using Int to compare values. \(2^{32}\) possible values instead of the needed 3
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that invariants can be encoded into the type system, giving you greater guarantees.

Clib itself uses a number of ADTs:
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that in-variants can be encoded into the type system, giving you greater guarantees.

Clilib itself uses a number of ADTs:

- *Fit* - Fitness value, either *Valid* or *Penalty*
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that in-variants can be encoded into the type system, giving you greater guarantees.

Clilib itself uses a number of ADTs:

- Fit - Fitness value, either Valid or Penalty
- Position[A] - Position in the search space. Point or Solution
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that invariants can be encoded into the type system, giving you greater guarantees.

Clib itself uses a number of ADTs:

- **Fit** - Fitness value, either Valid or Penalty
- **Position[A]** - Position in the search space. Point or Solution
- **Bound[A]** - Search space limits, either Open or Closed
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that invariants can be encoded into the type system, giving you greater guarantees.

Clib itself uses a number of ADTs:

- **Fit** - Fitness value, either *Valid* or *Penalty*
- **Position[A]** - Position in the search space. *Point* or *Solution*
- **Bound[A]** - Search space limits, either *Open* or *Closed*
- **Constraint[A,B]** - Position constraints: <, >, <=, >=, == and *InInterval*
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that invariants can be encoded into the type system, giving you greater guarantees.

Clib itself uses a number of ADTs:

- **Fit** - Fitness value, either Valid or Penalty
- **Position[A]** - Position in the search space. Point or Solution
- **Bound[A]** - Search space limits, either Open or Closed
- **Constraint[A,B]** - Position constraints: <, >, <=, >=, == and InInterval
- **Opt** - Optimization scheme, either Min or Max
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that invariants can be encoded into the type system, giving you greater guarantees.

ClLib itself uses a number of ADTs:

- **Fit** - Fitness value, either **Valid** or **Penalty**
- **Position[A]** - Position in the search space. **Point** or **Solution**
- **Bound[A]** - Search space limits, either **Open** or **Closed**
- **Constraint[A,B]** - Position constraints: `<`, `>`, `<=`, `>=`, `==` and **InInterval**
- **Opt** - Optimization scheme, either **Min** or **Max**
- **Eval[A]** - Evaluation scheme for a given **Position**, can be **Constrained** or **Unconstrained**
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that in-variants can be encoded into the type system, giving you greater guarantees.

Clib itself uses a number of ADTs:

- **Fit** - Fitness value, either Valid or Penalty
- **Position[A]** - Position in the search space. Point or Solution
- **Bound[A]** - Search space limits, either Open or Closed
- **Constraint[A,B]** - Position constraints: <, >, <=, >=, == and InInterval
- **Opt** - Optimization scheme, either Min or Max
- **Eval[A]** - Evaluation scheme for a given Position, can be Constrained or Unconstrained
- **RNG** - type of PRNG, limited to CMWC
ADTs: Custom algebras

ADTs also play extremely nicely with pattern matching. Exhaustiveness checking from the pattern matcher will also apply to ADTs.

The closed nature means that invariants can be encoded into the type system, giving you greater guarantees.

Clib itself uses a number of ADTs:

- **Fit** - Fitness value, either Valid or Penalty
- **Position[A]** - Position in the search space. Point or Solution
- **Bound[A]** - Search space limits, either Open or Closed
- **Constraint[A,B]** - Position constraints: <, >, <=, >=, == and InInterval
- **Opt** - Optimization scheme, either Min or Max
- **Eval[A]** - Evaluation scheme for a given Position, can be Constrained or Unconstrained
- **RNG** - type of PRNG, limited to CMWC
- **Generator** - random generators based on RNG, produce Int, Double or Boolean
Typeclasses

Staying classy
Typeclasses: Open-world interfaces
Typeclasses: Open-world interfaces

- Typeclasses allow for adhoc polymorphism
Typeclasses: Open-world interfaces

- Typeclasses allow for adhoc polymorphism
- Serve to create a categorization of types
Typeclasses: Open-world interfaces

- Typeclasses allow for adhoc polymorphism
- Serve to create a categorization of types
- Extend the functionality of a given type
Typeclasses: Open-world interfaces

- Typeclasses allow for adhoc polymorphism
- Serve to create a categorization of types
- Extend the functionality of a given type
- Does not involve inheritance, yay!
Typeclasses: Open-world interfaces

- Typeclasses allow for adhoc polymorphism
- Serve to create a categorization of types
- Extend the functionality of a given type
- Does not involve inheritance, yay!
- Makes the compiler work for you and acts like a kind of proof-assistant
Typeclasses: Definition

Typeclasses are a kind of pattern in Scala, whereas they are first-class in languages like Haskell. Typeclasses in Scala are defined using a parameterized trait:

```scala
trait Show[A] {
  def show(a: A): String
}
```

It's important to recognize that the type in question, `A`, is passed to the function directly and is not obtained from somewhere "magical"
Typeclasses: Usage
Typeclasses: Usage

- Typeclasses are made useful by having "instances" defined for the class
Typeclasses: Usage

- Typeclasses are made useful by having "instances" defined for the class
- Each instance of the typeclass is an instance of the typeclass trait, with a given value for the type parameter "hole"
Typeclasses: Usage

- Typeclasses are made useful by having "instances" defined for the class
- Each instance of the typeclass is an instance of the typeclass trait, with a given value for the type parameter "hole"
- Each instance is defined as an implicit value
Typeclasses: Usage

- Typeclasses are made useful by having "instances" defined for the class
- Each instance of the typeclass is an instance of the typeclass trait, with a given value for the type parameter "hole"
- Each instance is defined as an implicit value
- Compiler will lookup typeclass instances based on the needed type, looking at the current scoping
Typeclasses: Usage

- Typeclasses are made useful by having "instances" defined for the class
- Each instance of the typeclass is an instance of the typeclass trait, with a given value for the type parameter "hole"
- Each instance is defined as an implicit value
- Compiler will lookup typeclass instances based on the needed type, looking at the current scoping

- **Missing typeclass instance for a given type: failure in compilation**
Typeclasses: Usage

- Typeclasses are made useful by having "instances" defined for the class.
- Each instance of the typeclass is an instance of the typeclass trait, with a given value for the type parameter "hole".
- Each instance is defined as an implicit value.
- Compiler will lookup typeclass instances based on the needed type, looking at the current scoping.
- Missing typeclass instance for a given type: failure in compilation.

```scala
implicit val intShow = new Show[Int] {
  def show(a: Int) = a.toString
}
```
Typeclasses: Usage (2)

Functions that work with a given typeclass, expressly define it as an implicit parameter or context bound, resulting in a function that is “constrained”

\[
def \text{show}[A](a: A)(\text{implicit shower: Show}[A]) = \text{shower}.\text{show}(a)
\]

Context bounds are the same as an implicit parameter, except that the typeclass is not bound to a name and the constraint is provided in the type parameter:

\[
def \text{show}[A: \text{Show}](a: A) = \text{implicitly}[\text{Show}[A]].\text{show}(a)
\]
Typeclasses: Usage (2)

Functions that work with a given typeclass, expressively define it as an implicit parameter or context bound, resulting in a function that is “constrained”

```scala
def show[A](a: A)(implicit shower: Show[A]) = shower.show(a)
```

*Context bounds* are the same as an implicit parameter, except that the typeclass is not bound to a name and the constraint is provided in the type parameter:

```scala
def show[A: Show](a: A) = implicitly[Show[A]].show(a)
```

Of course, there is no limit to the number of constraints a function may have. Each typeclass constraint adds a level of "strictness" to the function.
Typeclasses: Usage (2)

Functions that work with a given typeclass, expressly define it as an implicit parameter or context bound, resulting in a function that is “constrained”

```scala
def show[A](a: A)(implicit shower: Show[A]) = shower.show(a)
```

*Context bounds* are the same as an implicit parameter, except that the typeclass is not bound to a name and the constraint is provided in the type parameter:

```scala
def show[A: Show](a: A) = implicitly[Show[A]].show(a)
```

Of course, there is no limit to the number of constraints a function may have. Each typeclass constraint adds a level of "strictness" to the function.

Syntax may also be used to achieve the same result. Scalaz adds such syntax for a set of types

```scala
3.shows // Syntax that uses Show[Int]
```
Typeclasses: So what is the point?

- So we have declared functionality for a type, without actually changing the implementation of the type
  - The type `A` knows **nothing** about `Show`

- This is known as the *open world assumption*

- **Very important:** this is not subtyping
  - Subtyping requires an implementation of a method for each extending class
  - Typeclasses (aka adhoc polymorphism) allow the omission of implementations, which is fantastic if a certain type *shouldn’t* have an instance defined because the functionality either doesn’t make sense or if it is incorrect to have an instance defined (violation of laws)
The Typeclassopedia

As taken from research done in Haskell:

- These structures allow better structure to functional programs
- Typeclasses are not magical, but do provide a common nomenclature
- Defined in the scala library
Typeclasses: Semigroup

Imagine having a set of objects, of type $A$, and a function that can combine two values of $A$ into a new value of type $A$.

This binary, two element, operator must always adhere to the associative law in order to behave in a consistent manner

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

where $\cdot$ is the binary operator and is also called append
Typeclasses: Semigroup

Imagine having a set of objects, of type $A$, and a function that can combine two values of $A$ into a new value of type $A$.

This binary, two element, operator must always adhere to the associative law in order to behave in a consistent manner

$$a \cdot (b \cdot c) = (a \cdot b) \cdot c$$

where $\cdot$ is the binary operator and is also called append

You can imagine the number of possible instances that the semigroup structure may have
Typeclasses: Semigroup (2)

The representation in code:

```scala
trait Semigroup[A] {
  def append(x: A, y: A): A
}
```
Typeclasses: Semigroup (2)

The representation in code:

```scala
trait Semigroup[A] {
  def append(x: A, y: A): A
}
```

Sample implementation

```scala
implicit val stringSemigroup = new Semigroup[String] {
  def append(x: String, y: String) = x + y
}
```
Typeclasses: Monoid

Monoid is a structure that builds on Semigroup by adding an identity element to the abstraction. This operation is also called zero or identity. This means that the following laws need to be adhered to

\[
\begin{align*}
    a \cdot (b \cdot c) &= (a \cdot b) \cdot c \\
    a \cdot \text{id} &= \text{id} \cdot a
\end{align*}
\]
Typeclasses: Monoid (2)

In code:

```scala
trait Monoid[A] extends Semigroup[A] {
  def zero: A
}
```
Typeclasses: Monoid (2)

In code:

```scala
trait Monoid[A] extends Semigroup[A] {
  def zero: A
}
```

Sample implementation

```scala
implicit val intMonoid = new Monoid[Int] {
  def zero: Int = 0
  def append(x: Int, y: Int) = x + y
}
```
Typeclasses: Monoid (3)

There are many monoids. Far more than what seem obvious at first.

Very important to remember that monoids beget monoids! Given a Monoid[Int] and a Monoid[List], Monoid[List[Int]] can be constructed.

Monoid examples

- Int Addition: \((0,+)\)
- Int Multiplication: \((1,\ast)\)
- String concatenation: \("\),+\)
- Function composition: \((\text{identity}, \text{compose})\)
- List concatenation: \((\text{Nil},++)\)
- etc
**Functor** represents a structure that allows you to transform (map) a value in the structure into a value of another type (but still in the same structure), given a transformation function.

The structure is often called a "computational context", which is an abstract notion, but there is nothing else that really fits here.
Typeclasses: Functor (2)

Functors are structures that define a single function `map`

```scala
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

An instance for the `Option ADT`

```scala
implicit val optionFunctor = new Functor[Option] {
  def map[A,B](fa: Option[A])(f: A => B): Option[B] =
  fa match {
    case Some(a) => Some(f(a))
    case None    => None
  }
}
```
Typeclasses: Functor (3)

Functor instances, like normal functions, can be composed to create new Functor instances. In essence, a Functor "lifts" a regular function into the context of the Functor.

Functors operate on kinds of the shape \( * \rightarrow * \), where a type parameter is needed to create the type, like:

- Lists
- Trees
- Maps

Functor laws:

- \( \text{map}(\text{id})(fa) == fa \)
- \( \text{map}(fa)(f \circ g) == \text{map}(\text{map}(fa)(g))(f) \)

These laws prevent ridiculous definitions of \( \text{map} \), and make it possible to predict how functions behave.
Typeclasses: Functor (3)

1. UNWRAP VALUE FROM CONTEXT
2. APPLY FUNCTION
3. REWRAP VALUE IN CONTEXT
Aside: Parametricity

```scala
def fun[A](a: A): A = ...
```

What is `fun`?

```scala
val x : List[Int] = ...
def foo[F[_] : Functor](fs : F[Int]): F[Int] = ...
```

What is `foo(x).length`?
Typeclasses: Applicative functor

You can do a lot with `map`, but sometime you need more power

Consider

```scala
def parseInt(s: String): Option[Int] = ...
```

Now if you want to add two `Ints` together:

```scala
parseInt("5").map((x: Int) => (y: Int) => x + y)
```
Typeclasses: Applicative functor

You can do a lot with `map`, but sometime you need more power.

Consider

```scala
def parseInt(s: String): Option[Int] = ...
```

Now if you want to add two `Ints` together:

```scala
parseInt("5").map((x: Int) => (y: Int) => x + y)
```

So we get `Option[Int => Int]`, but what now?

We need a *more* powerful `map-like` operation, one that allows us to kind of "map in the context"
Typeclasses: Applicative functor (2)

It can be achieved by adding more power to Functor:
Typeclasses: Applicative functor (2)

It can be achieved by adding more power to `Functor`:

```scala
trait Applicative[F[_]] extends Functor[F] {
  def <*>[A, B](fa: F[A])(f: F[A => B]): F[B] // This is also called "apply" or "ap"
}
```
Typeclasses: Applicative functor (2)

It can be achieved by adding more power to `Functor`:

```scala
trait Applicative[F[_]] extends Functor[F] {
  def <*>[A, B](fa: F[A])(f: F[A => B]): F[B] // This is also called "apply" or "ap"
}
```

Again, laws must be adhered to (Haskell-ish pseudo-code for space reasons):

- **Identity**: `pure identity <*> u === u`
- **Composition**: `pure (.) <*> u <*> v <*> w == u <*> (v <*> w)`
- **Homomorphism**: `pure f <*> pure x == pure (f x)`
- **Interchange**: `u <*> pure x == pure (\f -\> f x) <*> u`
Typeclasses: Applicative functor (3)

So this allows for the composition we wanted:

```plaintext
(pthread("5") . <*> (pthread("Nope") . map (x => (y: Int) => x + y))
```

But that's obviously horrible...

Sadly, the syntax of scala is actually working against us. As an example, the same example in Haskell would be

```plaintext
(+) <$> parseInt("3") <*> parseInt("Nope")
```

Applicative functors are a generalization of `map`, allowing us to work with functions with multiple inputs.

As a result, additional syntax is introduced to `smooth` out the usage in Scala, which Scalaz provides known as the `ApplicativeBuilder` which allows us to use the code in a nicer manner.
Typeclasses: Applicative functor (3)

So this allows for the composition we wanted:

```scala
(parseInt("5")). <*> (parseInt("Nope").map(x => y: Int => x + y))
```

But that’s obviously horrible...

Sadly, the syntax of scala is actually working against us. As an example, the same example in Haskell would be:

```scala
(+) <$> parseInt("3") <*> parseInt("Nope")
```

Applicative functors are a generalization of map, allowing us to work with functions with multiple inputs.

As a result, additional syntax is introduced to smooth out the usage in scala, which Scalaz provides known as the ApplicativeBuilder which allows us to use the code in a nicer manner:

```scala
(parseInt("5") |@| parseInt("Nope")) { _ + _ } // Option[Int]
```
Typeclasses: Applicative functor (4)
Typeclasses: Applicative functor (5)

So now operations in some context can be done with multi argument functions

What happens if you have a value that isn't in the context?

```haskell
(partialInt("5") @@ 4) { _ + _ } // error: this will not compile
```

Just like how we have `map`, that lifts a function into the context, we need a way to lift a `value` into the context.
Typeclasses: Applicative functor (6)

One small addition to the typeclass:

```scala
trait Applicative[F[_]] extends Functor[F] {
  def <*>[A, B](fa: F[A])(f: F[A => B]): F[B]
  def point[A](a : A): F[A]
}
```

With `point` defined, we can use the applicative as expected (using some syntax):

```scala
(parseInt("5") |@| 4.point[Option]) { _ + _ }
```

Additionally notice that the way `Applicative` receives its arguments means that all the arguments need to be available, and ready, before the composition function is applied. This actually can be exploited, allowing applicatives to do things in `parallel` before composition.
Typeclasses: Applicative functor (7)

1. Function wrapped in a context

2. Value in a context

3. Unwrap both and apply the function to the value

4. New value in a context
Typeclasses: Applicative functor (8)
Typeclasses: Monad

Let's take a step back and see how something else might go wrong.

Assume for a moment, that we needed to obtain a value from an associative Map, which then would need to be parsed into an integer value.
Typeclasses: Monad

Let's take a step back and see how something else might go wrong

Assume for a moment, that we needed to obtain a value from an associative Map, which then would need to be parsed into an integer value

Because getting the value from the Map, might fail (i.e. the value is not present), the return type is a Option[String]
Typeclasses: Monad

Let's take a step back and see how something else might go wrong.

Assume for a moment, that we needed to obtain a value from an associative Map, which then would need to be parsed into an integer value.

Because getting the value from the Map, might fail (i.e. the value is not present), the return type is a Option[String].

So we have:

```scala
val x: Option[String] = variables.get("x")
```

to which we apply our trusty hammer, map:

```scala
x.map(parseInt) // Result is Option[Option[Int]]
```
Typeclasses: Monad (2)

We need to be able to reach into a computational context, pull out the value and apply a function which results in a new computational context (which is the same as the original), then having the nested structure flattened
Typeclasses: Monad (2)

We need to be able to reach into a computational context, pull out the value and apply a function which results in a new computational context (which is the same as the original), then having the nested structure flattened

```
trait Monad[F[_]] extends Applicative[F] {
}
```

This weird operator `>>=` is pronounced "bind" and there are a set of laws, naturally:

- **Left identity:** `return a >>= f` · `f a`
- **Right identity:** `m >>= return` · `m`
- **Associativity:** `(m >>= f) >>= g` · `m >>= (\x -> f x >>= g)`
Some people talk about a "shoving" values though the monad
Typeclasses: Monad (4)

So, we can now do what we wanted, letting the monad correctly flatten the structure for us:

```haskell
variables.get("x") >>= parseInt // Option[Int]
```

This can be repeated, feeding the result of one action into a function that produces the next action:

```haskell
monadicX >>= (x => monadicY >>= (y => monadicZ map (z => x+y+z)))
```

The pattern here is a group of nested binds followed by a single map, but its no fun to read.
Typeclasses: Monad (4)

So, we can now do what we wanted, letting the monad correctly flatten the structure for us

```haskell
variables.get("x") >>= parseInt // Option[Int]
```

This can be repeated, feeding the result of one action into a function that produces the next action:

```haskell
monadicX >>= (x => monadicY >>= (y => monadicZ map (z => x+y+z)))
```

The pattern here is a group of nested binds followed by a single map, but it's no fun to read.

This brings up an important concept: Monads allow for the sequencing of effects. This allows the lazy, purely functional language Haskell to do things in order.
If the syntax is formatted a little:

```haskell
monadicX >>= ( x =>
  monadicY >>= ( y =>
    monadicZ map ( z =>
      x + y + z
    )))
```
Typeclasses: Monad (5)

If the syntax is formatted a little:

```haskell
monadicX >>= ( x =>
  monadicY >>= ( y =>
    monadicZ map ( z =>
      x + y + z
    )
  )
)
```

That does look better, but because the pattern is used so often, the compiler provides some syntax sugar which the programmer can use, but the compiler de-sugars the syntax during compilation.

```haskell
for {
  x <- monadicX
  y <- monadicY
  z <- monadicZ
} yield x + y + z
```
Typeclasses: Monad (6)

Monads provide an embedded language:

```plaintext
for {
  x <- monadicX
  y <- monadicY
} yield x + y
```
Typeclasses: Monad (6)

Monads provide an embedded language:

```plaintext
for {
    x <- monadicX
    y <- monadicY
} yield x + y
```

What does the above for-comprehension (scala parlance), or even better monad-comprehension do?
Typeclasses: Monad (6)

Monads provide an embedded language:

```plaintext
for {
    x <- monadicX
    y <- monadicY
} yield x + y
```

What does the above for-comprehension (scala parlance), or even better monad-comprehension do?

The answer is quite simple: it depends on what the monad is defined to do, while maintaining the laws

<table>
<thead>
<tr>
<th>Monad</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Option</td>
<td>Anonymous exceptions / missing values</td>
</tr>
<tr>
<td>Reader</td>
<td>Read-only / immutable environment</td>
</tr>
<tr>
<td>Validation</td>
<td>Descriptive exceptions</td>
</tr>
<tr>
<td>List</td>
<td>Non-deterministic computation</td>
</tr>
<tr>
<td>Future</td>
<td>Valid computed asynchronously</td>
</tr>
</tbody>
</table>
Typeclasses: Signatures

Have a look at the following, notice the difference in the structures

```scala
```

It should also be noted again that, as per the typeclassopedia diagram:

- All Applicative instances are also valid Functor instances, but not all Functors are Applicative
- All Monad instances are valid Applicatives (and by extension Functors), but not vice versa
Always use the relevant structure to achieve your desired goal. For example:

```haskell
for {
    aa <- a
    bb <- b
} yield f(aa, bb)
```

does not need monad sequencing in order to call the function \( f \). An applicative functor is better:

```haskell
(a ± b) { f( _, _ ) }
```
Typeclasses: Monad transformers

In programming we need to deal with a variety of multiple kinds of effects:

- A program needs some configuration (Reader)
- A connection to an external system may fail (Validation or Either)
- Computation needs to be asynchronous (Future)
- Computation may produce zero or multiple results (List)
- We want to log some additional data during a computation (Writer)
Typeclasses: Monad transformers

In programming we need to deal with a variety of multiple kinds of effects:

- A program needs some configuration (Reader)
- A connection to an external system may fail (Validation or Either)
- Computation needs to be asynchronous (Future)
- Computation may produce zero or multiple results (List)
- We want to log some additional data during a computation (Writer)

Unfortunately, combining these effects gets messy and the reason for this is that it is not possible to compose any two random monads - you simply cannot write the >>= operation in a completely generic way. Please feel free to try it :)

```scala
implicit val ComposedMonad[M: Monad, N: Monad] = new Monad[...]
{
}
```
Typeclasses: Monad transformers (2)

Let's see why stacking monads gets messy

```scala
type Env = Map[String, Int]
def lookup(s: String): Reader[Env, Option[Int]] = Reader(env => env.get(s) orElse parseInt(s))
```

Remember that you cannot "mix" monads! If you do, you completely invalidate the laws and as a result, it is impossible to ensure that the computation will be consistent.

Say we now add two numbers, using Reader to prevent repetition:

```scala
for {
  xOpt <- lookup("3")
  yOpt <- lookup("y")
} yield (for {
  x <- xOpt
  y <- yOpt
} yield x + y)
```
Typeclasses: Monad transformers (3)

This switch from monad to monad is often referred to as "stair-stepping" and results in very nested, verbose code. It's turtles all the way down.
Transformers are wrappers that exploit the knowledge of the inner monad instance of the given monad

```scala
case class OptionT[F[_, _], A](run: F[Option[A]]) {
  ...
}
```
Transformers are wrappers that exploit the knowledge of the inner monad instance of the given monad

```scala
case class OptionT[F[_], A](run: F[Option[A]]) {
  ...
}
```

```scala
OptionT(List(Some(1), None, Some(3), Some(4))).getOrElse(0) == List(1,0,3,4)
```

It’s important to realize that the transformer is a monad itself, therefore you can nest transformers arbitrarily deep, but the order of the stacking is significant
Typeclasses: Monad transformers (4)

It is the convention that the monad transformer instances mirror the inner monad’s name, with a T suffix

- OptionT
- ReaderT, aka Kleisli (yes, from category theory: the Kleisli category)
- WriterT
- EitherT
- ListT

Now we can cleanup our previous stair-stepping example:
Typeclasses: Monad transformers (4)

It is the convention that the monad transformer instances mirror the inner monad’s name, with a T suffix

- OptionT
- ReaderT, aka Kleisli (yes, from category theory: the Kleisli category)
- WriterT
- EitherT
- ListT

Now we can cleanup our previous stair-stepping example:

```haskell
for {
    x <- Kleisli(lookup("3"))
    y <- Kleisli(lookup("y"))
} yield x + y
```
Typeclasses: Monad transformers (5)

Again, a monad transformer is defined by a typeclass:

```scala
trait MonadTrans[F[___], ___] {
  def liftM[G[___]: Monad, A](a: G[A]): F[G,A]
}
```
Typeclasses: Monad transformers (5)

Again, a monad transformer is defined by a typeclass:

```scala
traitMonadTrans[F[[_], _]] {
  def liftM[G[_]: Monad, A](a: G[A]): F[G, A]
}
```

`liftM` allows us to "lift" computations in the base monad into the transformer:

```scala
(for {
  x <- OptionT(List(Some(3), None))
  y <- List(1, 2).liftM[OptionT]
} yield x + y).run // List(Some(4), None, Some(5), None)
```
Lenses
Lenses are, in a very simplified sense, functional getters and setters that allow composition. They are part of a greater idea; that of Optics (which we won't get into today).
Lenses

Lenses are, in a very simplified sense, functional getters and setters that allow composition. They are part of a greater idea; that of **Optics** (which we won’t get into today).

Assume a nested structure of case classes, to update a value deep inside, code ends up looking like:

```scala
val employee: Employee = ...

employee.copy(
  company = employee.company.copy(
    address = employee.company.address.copy(
      street = employee.company.address.street.copy(
        name = employee.company.address.street.name.capitalize // capitalize exists
      )
    )
  )
)
```

This need not be case classes, any nested structure is applicable.
Using lenses, we can define a composition to zoom down into a structure, handling the updates of all the immutable data by the lenses as we "zoom out"

```scala
val _name : Lens[Street , String] = ...  // we'll see later how to build Lens
val _street : Lens[Address , Street] = ...
val _address: Lens[Company , Address] = ...
val _company: Lens[Employee, Company] = ...

(company composeLens _address
  composeLens _street
  composeLens _name).modify(_.capitalize)(employee)
```

There are a host of other abstractions that Lenses result in, but they are not in the scope for this talk - we do use lenses a lot in CIBlib though
Lenses (3)

Creating a lens is very simple. You need to provide the following for the types in question:

- **getter** function
- **setter** function
Lenses (3)

Creating a lens is very simple. You need to provide the following for the types in question:

- **getter** function
- **setter** function

```scala
case class Mem[A](b: Position[A], v: Position[A])

trait Memory[S,A] {
  def _memory: Lens[S, Position[A]]
}

object Memory {
  implicit def memMemory = new Memory[Mem[Double],Double] {
    def _memory = Lens[Mem[Double],Position[Double]](_.b)(b => a => a.copy(b = b))
  }
}
```
Clilib 2.0
We've spoken a lot about abstractions and the like before getting to this point. It was necessary as we build on these abstractions within Cllib

Now let's start going through the core concepts and abstractions we have:

- Entity
- Position
- RVar
- Dist
- Step
- Iteration
Looking back on Clib 1.0, there were many *entity* types:

- Individual
- Harmony
- Particle
  - StandardParticle
  - ChargedParticle
  - DynamicParticle
  - etc
- Bee
- etc

And various permutations and adaptations of the above
Looking at all the entity instances, a general structure emerged
Looking at all the entity instances, a general structure emerged. **Every** entity can be described / represented as a 2-tuple consisting of:

- State specific to the entity
- Position that the entity represents of some point in the search space
Clilib: Entity (2)

Looking at all the entity instances, a general structure emerged

*Every* entity can be described / represented as a 2-tuple consisting of:

- State specific to the entity
- Position that the entity represents of some point in the search space

```scala
case class Entity[S, A](state: S, pos: Position[A])
```

Furthermore, we needed to change our terminology: there is no need to talk about swarms, populations, colonies, etc. Instead we can just refer to a *collection* of entities
The interesting part of the Entity is actually the state represented by the $S$ type parameter Entity is a general data structure, so how do we represent that this Entity contains various "pieces of state"?

This is where some structure comes in, in the form of:
Cllib: Entity (3)

The interesting part of the Entity is actually the state represented by the S type parameter. Entity is a general data structure, so how do we represent that this Entity contains various "pieces of state"?

This is where some structure comes in, in the form of:

- An Entity composes different pieces of state together
The interesting part of the `Entity` is actually the state represented by the `S` type parameter. `Entity` is a general data structure, so how do we represent that this `Entity` contains various "pieces of state"?

This is where some structure comes in, in the form of:

- An `Entity` composes different pieces of state together
- The state of an `Entity` is extracted using `Lens` instances
The interesting part of the `Entity` is actually the state represented by the `S` type parameter. Entity is a general data structure, so how do we represent that this `Entity` contains various "pieces of state"?

This is where some structure comes in, in the form of:

- An `Entity` composes different pieces of state together
- The state of an `Entity` is extracted using `Lens` instances
- `Lens` instances also are used to update `Entity` state
The interesting part of the Entity is actually the state represented by the S type parameter Entity is a general data structure, so how do we represent that this Entity contains various "pieces of state"?

This is where some structure comes in, in the form of:

- An Entity composes different pieces of state together
- The state of an Entity is extracted using Lens instances
- Lens instances also are used to update Entity state

The most important aspect is now that because the Lens instances constrain an Entity, they also then become the constraints that algorithms specify, thereby preventing the wrong "kind" of Entity being used in a given function.
def updatePBest[S](// S is the entity state type
   p: Particle[S,Double]
)(implicit M: Memory[S,Double]): Step[Double,Particle[S,Double]] = {
    val pbestL = M._memory
    Step.liftK(Fitness.compare(p.pos, (p.state applyLens pbestL).get).map(x =>
       Entity(p.state applyLens pbestL set x, p.pos)
    ))
}

def updateVelocity[S](
   p: Particle[S,Double],
   v: Position[Double]
)(implicit V: Velocity[S,Double]): Step[Double,Particle[S,Double]] =
Step.pointR(RVar.point(Entity(p.state applyLens V._velocity set v, p.pos)))
Position is a data structure that is at the core of the library. Position is encoded as an ADT and represents two distinct cases:

- **Point** - A point within the problem search space
- **Solution** - A point within the search space that has been quantified to have both a Fit and a List[Constraint[A,B]] for constraint violations

Position is parameterized on the type of the individual dimension elements and are created by using Bound instances that define the search space.
Syntax has been added to allow for Vector based mathematics on `Position` instances, and any change to the position correctly translates it back to a `Point` instance.

- `Position + Position`
- `Position - Position`
- `scalar_value *: Position`

Additionally, `Position` is also a valid `Monad`, so `Position` instances can be composed in all the usual ways.
This is, without question, the **most important abstraction** within Clib.
This is, without question, the **most important abstraction** within Clib

**RVar** **tracks** the effect of randomness, and as a result allows for actions that use randomness to be declared and used without concerning ourselves with:

- How the randomness is applied
- Which PRNG to use
Clib: RVar

This is, without question, the most important abstraction within Clib.

RVar tracks the effect of randomness, and as a result allows for actions that use randomness to be declared and used without concerning ourselves with:

- How the randomness is applied
- Which PRNG to use

RVar is a declaration that the resulting type has randomness applied to it, and has the following guarantees:

- Using RVar is completely stack safe (i.e: no stack overflows)
- Providing the same seeded PRNG, you will get the same result
Clib: RVar (2)

Internals:

- RVVar internally is itself a monad transformer stack
- The base monad is a State monad that concerns itself with managing the state of the PRNG
- Around the base monad is a Trampoline, which provides a way for the computation to suspend itself and then be resumed later
- RVVar is the base monad for Clib

RVVar is constrained to only produce elements that are defined using the Generator typeclass, which are currently:

- Double
- Int
- Boolean
RVar requires as a parameter, an instance of a RNG

There is only a single RNG provided by Clib to users, the CMWC which is currently the suggested RNG for scientific computing. It has a period of $2^{131104}$ whereas the Mersenne Twister (which is also much more difficult to implement correctly) has a period of $2^{19937} - 1$
RVr requires as a parameter, an instance of a RNG.

There is only a single RNG provided by CliLib to users, the CMWC which is currently the suggested RNG for scientific computing. It has a period of $2^{131104}$ whereas the Mersenne Twister (which is also much more difficult to implement correctly) has a period of $2^{19937} - 1$.

RVr makes use of the standard derived operations on the Monad typeclass for most of the provided functionality:

```haskell
def ints(n: Int) = 
next[Int](Generator.IntGen) replicateM n
```

Additional functions made available by RVr, that return RVr instances include:

- `choose` - randomly select a value out of a NonEmptyList $\rightarrow$ RVr[A]
- `shuffle` - randomly shuffle the given List $\rightarrow$ RVr[List[A]]
- `sample` - randomly sample / choose n elements from the provided List $\rightarrow$ RVr[List[A]]
Building on RVar is Dist

Dist contains several algorithms that implement probability distributions using the primitives that RVar provides.

The probability distributions currently supported:

- `uniformInt(from, to)` - Uniform integer from the provided range (inclusive)
- `uniform(from, to)` - Uniform number from the given range (upper bound excluded)
- `cauchy(θ, δ)`
- `gamma(k, θ)`
- `exponential(l)`
- `lognormal(θ, δ)`
- `dirichlet(θ*)`
- `gaussian(θ, δ)` - Gaussian numbers using Doornik's improved ziggurat method
Clib: Dist (2)

There are several "standard" distributions also available so that the parameters need not be provided each time:

- stdUniform
- stdNormal
- stdCauchy
- stdExponential
- stdGamma
- stdLaplace
- stdLognormal

The algorithms implemented for the probability distributions are, as far as possible, are based on published works and aim to try an minimize the usage of the RNG.

For example: the JDK standard library implementation uses the "polar method" which samples two double values each time before testing if it will succeed the internal checks before being emitted. The ziggurat method used, in contrast, only samples a single double a time.
The next abstraction is that of a Step. Step represents a single operation within an algorithm definition.

The algorithms we are interested in are non-deterministic, and as a result the algorithms eventually ends up making use of randomness.
The next abstraction is that of a **Step**. **Step** represents a single operation within an algorithm definition.

The algorithms we are interested in are non-deterministic, and as a result the algorithms eventually ends up making use of randomness.

Randomness, encapsulated in **RVar**, is not the only concern that algorithms have. There are two other important pieces of information that govern the algorithm behavior:
The next abstraction is that of a Step. Step represents a single operation within an algorithm definition.

The algorithms we are interested in are non-deterministic, and as a result the algorithms eventually ends up making use of randomness.

Randomness, encapsulated in \( RVar \), is not the only concern that algorithms have. There are two other important pieces of information that govern the algorithm behavior:

- The optimization scheme: Minimization (Min) or maximization (Max)
The next abstraction is that of a Step. Step represents a single operation within an algorithm definition.

The algorithms we are interested in are non-deterministic, and as a result the algorithms eventually ends up making use of randomness.

Randomness, encapsulated in RVa, is not the only concern that algorithms have. There are two other important pieces of information that govern the algorithm behavior:

- The optimization scheme: Minimization (Min) or maximization (Max)
- A mechanism to quantify the provided candidate solution: Eval[A]
The `Eval[A]` instance, takes a `Position[A]` and yields a new `Position[A]`, except that the new `Position[A]` should contain a `Fit` and a list of all violated constraints.
The Eval[A] instance, takes a Position[A] and yields a new Position[A], except that the new Position[A] should contain a Fit and a list of all violated constraints.

Therefore, based on these requirements, Step instances need to provide both an Opt and an Eval[A] instance, which then finally yield RVar[A] values.
The `Eval[A]` instance takes a `Position[A]` and yields a new `Position[A]`, except that the new `Position[A]` should contain a `Fit` and a list of all violated constraints.

Therefore, based on these requirements, `Step` instances need to provide both an `Opt` and an `Eval[A]` instance, which then finally yield `RVar[A]` values.

Q: What can we do to make such a formulation possible?
Clilib: Step (2)

The Eval[A] instance, takes a Position[A] and yields a new Position[A], except that the new Position[A] should contain a Fit and a list of all violated constraints.

Therefore, based on these requirements, Step instances need to provide both an Opt and an Eval[A] instance, which then finally yield RVar[A] values.

Q: What can we do to make such a formulation possible?

A: We define a ReaderT (aka Kleisli) monad transformer (because RVar is the base monad in Clilib), which provides a read-only environment of (Opt, Eval[A])

```haskell
```
The `Eval[A]` instance, takes a `Position[A]` and yields a new `Position[A]`, except that the new `Position[A]` should contain a `Fit` and a list of all violated constraints.

Therefore, based on these requirements, `Step` instances need to provide both an `Opt` and an `Eval[A]` instance, which then finally yield `RVar[A]` values.

Q: What can we do to make such a formulation possible?

A: We define a `ReaderT` (aka `Kleisli`) monad transformer (because `RVar` is the base monad in Cllib), which provides a read-only environment of `(Opt, Eval[A])`

```scala
```

The expanded form of the `Step`, as what the scala compiler will report in some scenarios, looks like this:

```scala
val stepAsFunc = (env: (Opt,Eval[A])) => RVar[A]
```
def gbest[S](
    w: Double,
    c1: Double,
    c2: Double,
    cognitive: Guide[S,Double],
    social: Guide[S,Double]
)(implicit M: Memory[S,Double], V: Velocity[S,Double], MO:
Module[Position[Double],Double]
): List[Particle[S,Double]] => Particle[S,Double] => Step[Double,Particle[S,Double]] =
collection => x => for {
    cog   <- cognitive(collection, x)         // Step
    soc   <- social(collection, x)           // Step
    v     <- stdVelocity(x, soc, cog, w, c1, c2) // Step
    p     <- stdPosition(x, v)               // Step
    p2    <- evalParticle(p)                 // Step
    p3    <- updateVelocity(p2, v)           // Step
    updated <- updatePBest(p3)               // Step
} yield updated
Clib: Step (4)

So, the entire `gbest` function, returns a new function of the shape

```
List[Particle[S,Double]] => Particle[S,Double] => Step[Double,Particle[S,Double]]
```

which states that, given a `List[Particle[S,Double]]` (the collection of entities) and the current `Particle[S,Double]` (which is the Entity), you get back a single `Step[Double,Particle[S,Double]]`
Clib: Step (4)

So, the entire \texttt{gbest} function, returns a new function of the shape

\[
\text{List[Particle[S,Double]]} \Rightarrow \text{Particle[S,Double]} \Rightarrow \text{Step[Double,Particle[S,Double]]}
\]

which states that, given a \texttt{List[Particle[S,Double]]} (the collection of entities) and the current \texttt{Particle[S,Double]} (which is the Entity), you get back a \textbf{single} \texttt{Step[Double,Particle[S,Double]]}

This \texttt{Step[Double,Particle[S,Double]]} may then be "executed" or "run" by providing the optimization environment (\texttt{Opt, Eval[Double]}) and then finally a \texttt{RNG}
So, the entire `gbest` function, returns a new function of the shape

```
List[Particle[S,Double]] => Particle[S,Double] => Step[Double,Particle[S,Double]]
```

which states that, given a `List[Particle[S,Double]]` (the collection of entities) and the current `Particle[S,Double]` (which is the Entity), you get back a **single** `Step[Double,Particle[S,Double]]`

This `Step[Double,Particle[S,Double]]` may then be "executed" or "run" by providing the optimization environment (`Opt,Eval[Double]`) and then finally a RNG

The final result for this entire algorithm definition is a new version of the passed in `Particle[S,Double]`, based on the current collection and the old version of the particle
So, the entire `gbest` function, returns a new function of the shape

\[
\text{List[Particle[S,Double]]} \Rightarrow \text{Particle[S,Double]} \Rightarrow \text{Step[Double,Particle[S,Double]]}
\]

which states that, given a `List[Particle[S,Double]]` (the collection of entities) and the current `Particle[S,Double]` (which is the Entity), you get back a single `Step[Double,Particle[S,Double]]`

This `Step[Double,Particle[S,Double]]` may then be "executed" or "run" by providing the optimization environment (`Opt,Eval[Double]`) and then finally a `RNG`

The final result for this entire algorithm definition is a new version of the passed in `Particle[S,Double]`, based on the current collection and the old version of the particle

Applying this function repeatedly on all Entity instances in the collection results in the creations of a new collection
ClLib has based all its algorithm execution around the notion of an Iteration.
Clilb: Iteration

Clilb has based all its algorithm execution around the notion of an Iteration.

An Iteration is a type that defines how an "algorithm" instance will be run, provided a collection of Entity.
Clib: Iteration

Clib has based all its algorithm execution around the notion of an Iteration

An Iteration is a type that defines how an "algorithm" instance will be run, provided a collection of Entity

This execution can either be:
Clib: Iteration

Clib has based all its algorithm execution around the notion of an Iteration.

An Iteration is a type that defines how an "algorithm" instance will be run, provided a collection of Entity.

This execution can either be:

- Synchronous (sync) where the new Entity instances are determined solely off the old collection.
Clib: Iteration

Clib has based all its algorithm execution around the notion of an Iteration.

An Iteration is a type that defines how an "algorithm" instance will be run, provided a collection of Entity.

This execution can either be:

- **Synchronous** *(sync)* where the new Entity instances are determined solely off the old collection.

- **Asynchronous** *(async)* where the new Entity instances are determined off the currently building collection and the remainder of the old collection that has not yet been processed.
Clib: Iteration

Clib has based all its algorithm execution around the notion of an Iteration.

An Iteration is a type that defines how an "algorithm" instance will be run, provided a collection of Entity.

This execution can either be:

- Synchronous (sync) where the new Entity instances are determined solely off the old collection.
- Asynchronous (async) where the new Entity instances are determined off the currently building collection and the remainder of the old collection that has not yet been processed.

It is important to note that the sync version of the Iteration may be implemented completely in parallel as the old immutable collection is the only requirement, whereas the async version cannot be parallelised so easily because it needs a combination of the old and new collections, effectively making the processing sequential.
Some Demos
Also, Thank you!