```
Core Syntax: no inference, no sugar
L  ::= D[Xs] : D1[Ts1] ... Dn[Tsn] { 'x  Ms }
M  ::= sig, | sig -> e,
e  ::= x | e m[Ts](es) | L
sig::= m[Xs](x1:T1, ..., xn:Tn):T
T  ::= X | D[Ts]
```

Syntactic sugar:
- Can omit empty parenthesis {} [] or ()
- Can omit () on 1 argument method (becomes left associative operator)
- Top level Declarations omitted ‘x = ‘this’. Omitted ‘x for declarations in methods is fresh
- Omitted declaration name D of a literal inside of an expression is fresh

Concrete syntax:
- Declaration overloading based on generics arity,
- Method overloading on parameter arity, method names both as .lowercase or operators

Inference:
- The implemented types Ts of a declaration are inferred when the target type is known.
- An overridden sig can omit the types
- Can omit even the method name if there is only 1 abstract method


The Fearless Journey

2021: Publication on how to code in Java without ‘new’, ‘class’ and ‘static’

```
interface Bob{
  String name();
  default String greet(){ return "Hi, my name is "+this.name();}
}


..
Bob bob= ()->"Bob";//lambda implementing the method ‘name’
bob.greet();//can call any range of methods


//no explicit new: JVM in this case recognize this as a valid singleton


Bob factory(String bobName){ return ()->bobName; }
//no explicit fields. Object is allocated and will store ‘bobName’
//No fields== no difference between fields and methods,
//no super fields, no field shadowing, no constructors and superconstructors,
//no field initialization order, etc etc..
```

That is, a simple and elegant language is hiding inside of Java 8, but the java syntax is not designed to be used in this way...

2021: Publication on how to code in Java without ‘new’, ‘class’ and ‘static’

Now: Fearless: A minimalistic nominally typed pure OO language where
- There are no fields and all the state is captured by closures.
- Objects are described by method implementations and captured state
- Only 3 concepts: types, methods and expressions.

Types: types can be generics, so a type name denote a family of types:
The ‘List’ type name denotes types ‘List[String]’, ‘List[Int]’, ‘List[List[Int]]’ …

Methods: methods can be generic, and can be abstract or implemented with a single expression. Fearless is expression based, so there are no statements

Expressions: there are 3 kinds of expressions: local bindings, method calls and object literals.
Object literals offer a few syntactic variations, in the same way Java lambdas do

Crucially, types are just object literals that can have abstract methods.

```
Person[]: { .age[](): Num, .name[](): Str } //a Person with age and name
```

```
Person: { .age: Num, .name: Str }  //a Person with age and name
```

```
Person: { .age: Num, .name: Str }  //a Person with age and name
```

Not a record with two fields, but a type with two methods taking zero arguments.
Not behave like fields: trigger (potentially non terminating) computations.
No guarantee that any storage space is used by those methods.

Example: .name captures a string, .age is the length of the same string.

Making a person directly:

```
Bob: Person{ .age:Num -> 42, .name:Str -> "Bob" }
Bob: Person{ .age -> 42, .name -> "Bob" }//infer inherited method type
Person{ .age -> 42, .name -> "Bob" }//can be of an anonymous type
```

Making a person with a factory:

```
PersonF.of(42, "Bob")
```

Defined as:

```
PersonF:{.of(age: Num, name: Str): Person ->
  Anon:Person{.age:Num->age, .name:Num->name} }

PersonF:{.of(age: Num, name: Str): Person ->{.age->age, .name->name} }
```

But… wait...

```
PersonF.of(42, "Bob")    //making a person with a factory
```

What are 42 and “Bob”?

- `42 desugared as Anon:42[]{}`
- `"Bob" desugared as Anon:"Bob"[]{}`
- `PersonF == Anon:PersonF[]{}`

The singleton instance of any top level declaration with no abstract method
can be ‘summoned’ by just writing the name

There is no ‘static method call’ in Fearless, when we write

```
PersonF.of(42, "Bob")
```

we are writing a normal method call on the singleton instance of the type ‘PersonF’.

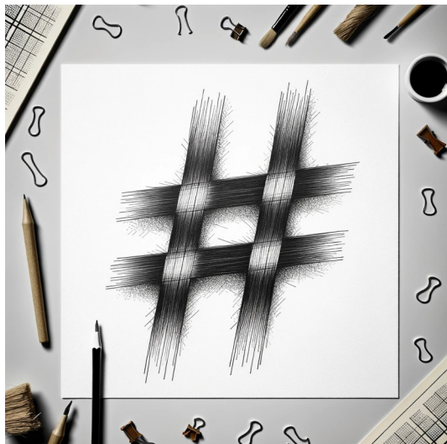
```
Person: { .age: Num, .name: Str }  //a Person with age and name
```

Functions can just be generic top level declarations

```
F[R]:{ #: R }           //Note: # is a valid method name just like .of
F[A, R]:{ #(a: A): R }
F[A, B, R]:{ #(a: A, b: B): R }
F[A, B, C, R]:{ #(a: A, b: B, c: C): R }
```

Now we PersonF can be a kind of function

```
PersonF:F[Num, Str, Person]{ age, name -> {.age->age, .name->name} }

PersonF#(42, "Bob")    //making a person with a function/factory
```



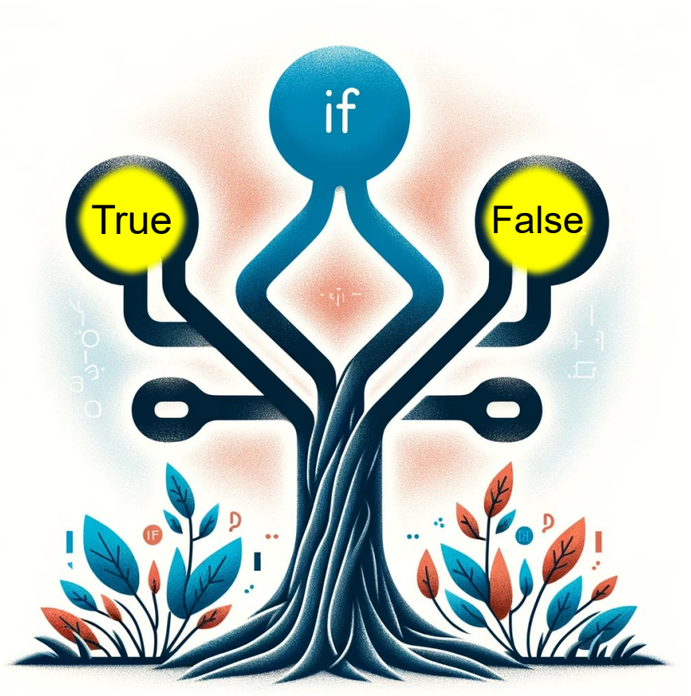
```
Person: { .age: Num, .name: Str }  //not declared at top level
```

Person as an internally declared trait instead

```
PersonF:F[Num, Str, Person]{ age, name -> Person:{
   .age:Num->age,
   .name:Str->name
   } }

PersonF#(42, "Bob")
```

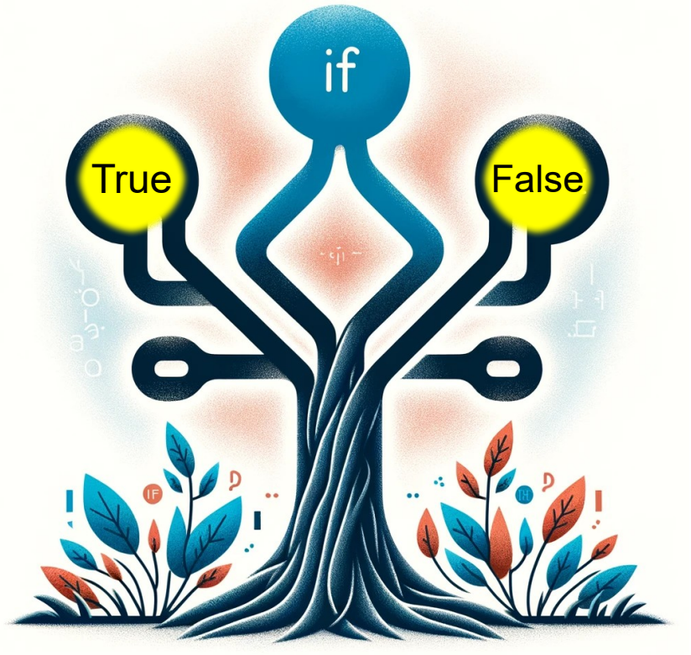
Same usage as before, but now this is guaranteed to be the only way to make a Person: A declaration name introduced inside a method body is ‘final’ and can not be inherited.
Writing ‘Person{…}’ anywhere else would be a type error.

```
Bool: {
   .and(other: Bool): Bool,
   .or(other: Bool): Bool,
   .not: Bool,

   }


True: Bool{
   .and(other) -> other,
   .or(other) -> this,
   .not -> False,

   }
False:Bool{
   .and(other) -> this,
   .or(other) -> other,
   .not -> True,

   }
```



```
Bool: {
   .and(other: Bool): Bool,
   .or(other: Bool): Bool,
   .not: Bool,
   .if[R](m: ThenElse[R]): R
   }
ThenElse[R]:{ .then: R, .else: R }

True: Bool{
   .and(other) -> other,
   .or(other) -> this,
   .not -> False,
   .if(m) -> m.then,
   }
False:Bool{
   .and(other) -> this,
   .or(other) -> other,
   .not -> True,
   .if(m) -> m.else,
   }
```

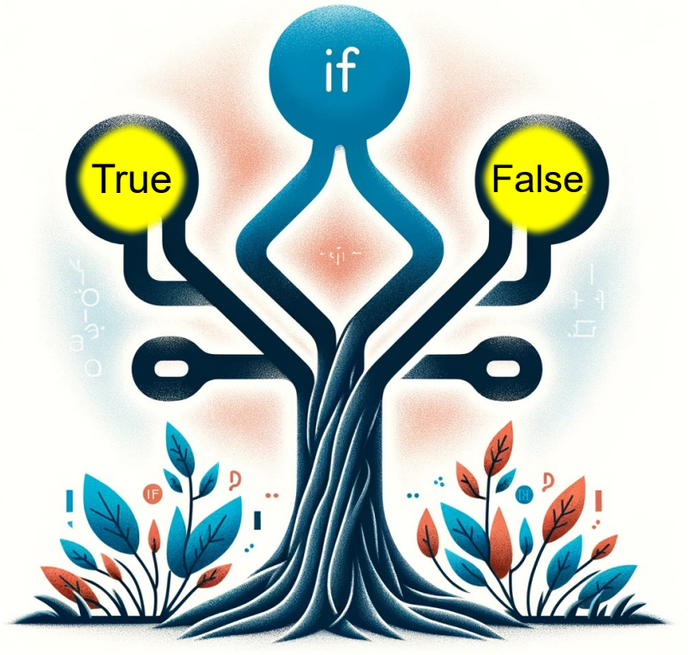
```
//usage example
True.and(False).if({
   .then->/*code for the then case*/,
   .else->/*code for the else case*/,
   })

True.and False.if{
   .then->/*code for the then case*/,
   .else->/*code for the else case*/,
   }
```



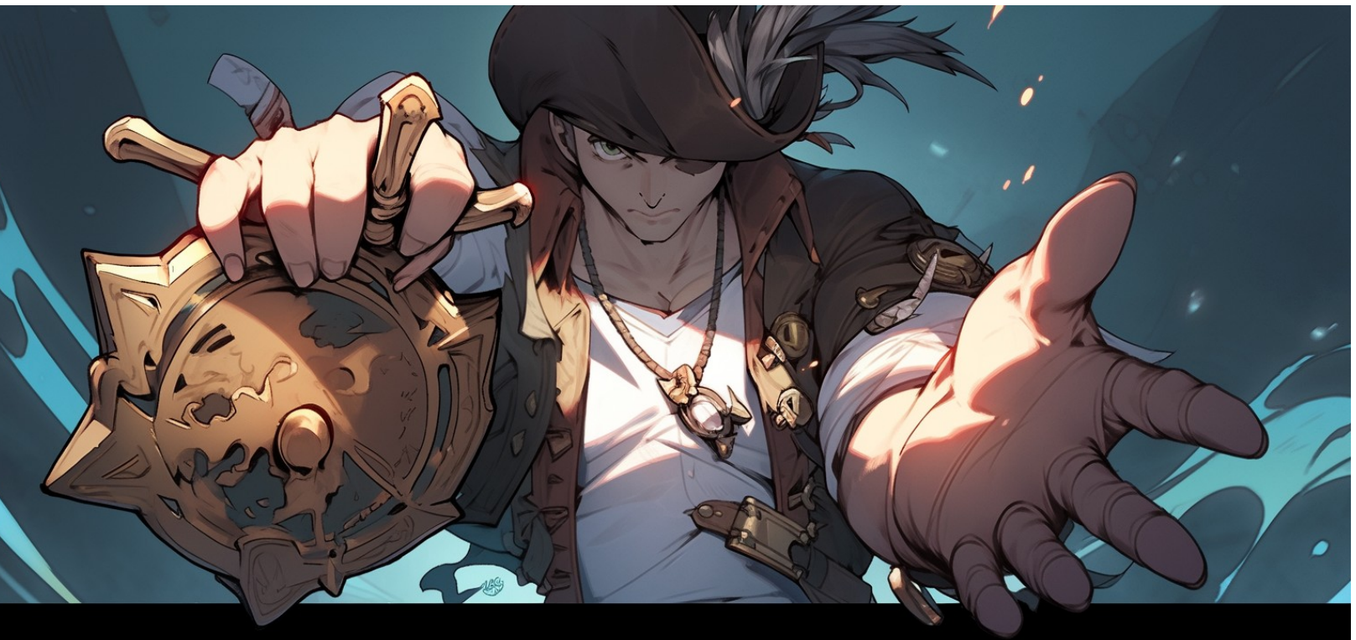
```
Bool: {..
   &&(other: F[Bool]): Bool,
   ||(other: F[Bool]): Bool,
   }

True: Bool{..
   &&(other) -> other#,
   ||(other) -> this,
   }
False:Bool{..
   &&(other) -> this,
   ||(other) -> other#,
   }
```

```
//usage example
My.stuff && {More.stuff}
//only executes More.stuff
//if My.stuff is True.
```

F[Bool] has a single abstract method with no parameters, so we can implement it by writing {..}



```
Opt[T]: {
   .match[R](m: OptMatch[T,R]): R -> m.empty
   }
OptMatch[T,R]: {
   .empty: R,
   .some(t: T): R
   }
Opt: {
   #[T](t:T):Opt[T] -> {  .match(m) -> m.some(t)  }
   }
```



```
//usage example
Opt#bob       //Bob is here
Opt[Person]   //no one is here
```

```
List[T]: {
   .match[R](m: ListMatch[T,R]): R -> m.empty
  +(e: T): List[T] -> { .match(m) -> m.elem(this, e) },
}
ListMatch[T,R]: {
   .empty: R,
   .elem(list: List[T], e: T): R
}

//usage examples
List[Num]+1+2+3
List[Opt[Num]]+{}+{}+(Opt#3)
List[List[Num]]+{}+{}+(List[Num]+3)

Example: {
   .sum(ns: List[Num]): Num -> ns.match{
      .empty -> 0,
      .elem(list, e) -> this.sum(list) + e
   }
}
```

```
List[T]: {
   .match[R](m: ListMatch[T,R]): R -> m.empty
  +(e: T): List[T] -> { .match(m) -> m.elem(this, e) },
   .map[R](f: F[T, R]): List[R] -> this.match{
     .empty -> {},
     .elem(list, e) -> list.map(f) + (f#e)
    }
}
ListMatch[T,R]: {
   .empty: R,
   .elem(list: List[T], e: T): R
}
```

```
//Visitor pattern in 1 slide
Html:{ .match[R](m: HtmlMatch[R]): R }
HtmlMatch[R]:{
   .h1(text: Str): R,
   .h5(text: Str): R,
   .a(link: Str, text: Str): R,
   .div(es: List[Html]): R,
}
HtmlF:HtmlMatch[Html]{ //the factory is a shallow clone visitor
   .h1(text) -> {m -> m.h1 text},
   .h5(text) -> {m -> m.h5 text},
   .a(link,text) -> {m -> m.a(link, text)},
   .div(es) -> {m->m.div es},
}
//HtmlClone == deep clone visitor
HtmlClone:HtmlF{.div(es) -> HtmlF.div(es.map{e -> e.match(this)}) }
CapitalizeTitles:HtmlClone{ .h1(text) -> Fhtml.h1(text.upperCase) }

...
myHtml.match(CapitalizeTitles)//usage
```

Hello world + Block

```
MyApp:Main{sys->Block#
   .let fs = {sys.fileSystem}
   .let content = {fs.read("data.txt")}
   .if {content.size > 5} .return {sys.println("Big")}
   .return {sys.println("Small")}
   }
```

A main type is simply a type implementing ‘Main’.
‘Main’ has a single abstract method taking a System object.
The System object has methods to do all kinds of IO effects.

Here we use ‘Block’: it allows us to write in a ‘statements-like’ style without having actual statements. ‘.let’, ‘.if’ and ‘.return’ are just methods of ‘Block’.
Similar to Java streams, where we chain ‘filter’, ‘map’ and so on to express behavior.
As for Java streams, all of the parameters of ‘block’ take lambdas/Object literals.

The ‘=’ sign is a new piece of syntax; it triggers a syntactic sugar to facilitate local variable declaration.

Hello world + Block

```
MyApp:Main{sys->Block#
   .let fs = {sys.fileSystem}
   .let content = {fs.read("data.txt")}
   .if {content.size > 5} .return {sys.println("Big")}
   .return {sys.println("Small")}
   }
-----------------------------
MyApp:Main{sys->Block#
   .let({sys.fileSystem},{fs,self1-> self1
     .let({fs.read("data.txt")},{content,self2-> self2
       .if {content.size > 5} .return {sys.println("Big")}
       .return {sys.println("Small")}
       })
     })
   }
```

Here is the version without the = sugar.
As you can see, local variables are encoded with nested object literals.


Braving Mutability:
Fearless's Journey
into mutability

Mutable state obtained by inserting a magic Var # implementation
Mutable state controlled with reference capabilities

```
Var[T]: {//First approximation
   .get: T,
   .set(x: T): Void,
   }

Var: { #[T](x: T): Var[T] -> Magic! }
```

Mutable state obtained by inserting a magic Var # implementation
Mutable state controlled with reference capabilities

```
Var[T]: {   //two .get in overloading
  mut  .get: T, //result: T as provided
  read .get: read/imm T, //result: T as read or imm
  mut  .set(x: T): Void,
  }

Var: { #[T](x: T): mut Var[T] -> Magic! }

Void: {}
```

Reference capabilities and grammar

```
R   ::= imm  | iso | read |  mut
L   ::= D[Xs] : D1[Ts1] ... Dn[Tsn] { 'x Ms }
M   ::= sig, | sig -> e,
e   ::= x | e.m[Ts](es) | R L
sig ::= R m[Xs](x1:T1, ..., xn:Tn):T
T   ::= R D[Ts] | X | R X | read/imm X
R   ::= imm | iso | read |  mut
```

Sugar: D[Ts] desugared as imm D[Ts], imm is also the default in sig
(all the code shown up to know still works and it is just all about imms)

Isolated iso

Immutable imm

Readable read

Mutable mut

Mutable mut

Easy to eat, easy to digest.
Hard to manage since they can tangle.

For hygenic reasons is better to eat your own food instead of having many strangers eating from the same plate.

Mut is like free Java objects,
not like rust mut or mut&

Isolated iso

The gold standard.
Affine: can be used only zero or one times.
It can be sold to a stranger.
It is often open only in private.

The whole MROG of the iso is only reachable from the iso reference itself.

Immutable objects can be freely shared

The unchanging eternal diamond.

Deep immutable objects as
in functional programming.

Can be shown to strangers, can be shared.

Immutable imm

The magnifying lens allows us to see stuff well, but we can not modify it or touch it using it.

You can look to any kind of things!

Readable read

Isolated iso

Immutable imm

Readable read

Mutable mut

Input output via Object Capabilities

Object Capabilities (OC) are not a type system feature, but a programming methodology that is greatly beneficial when it is embraced by the standard library.

The main idea is that instead of being able to make non deterministic actions like IO everywhere in the code by using static methods or public constructors, only certain specific objects have the 'capability' of doing those privileged actions, and access to those objects is kept under strict control.

In Fearless, this is done by having the main taking in input a mut System object. System is a normal type with all methods abstract. An instance of System with magically implemented methods is provided to the user as a parameter to the main method at the beginning of the execution.


safe passage


RC+OC = determinism

RC+OC = determinism

Note, with Vef[T], we need to distinguish identical by identity and structurally identical

(1) Any method taking in input only imm parameters is deterministic.
Pass structurally identical parameters and get a structurally identical result
Pass identity identical parameters and get a structurally identical result

(2) Any method taking in input only imm/read parameters is deterministic up to external mutation of its read parameters.
Pass structurally identical parameters and get a structurally identical result (and the parameters are not mutated)

This form of determinism is weaker that functional purity:
Non termination can happen (deterministically)
Exceptions can happen (deterministically and not)
Capturing exceptions without breaking this determinism property is possible but outside of the scope of this presentation


The Treasure:
Invariants, caching and automatic parallelism