

An Introduction to Functional Imperative Programming in **Flix**

GOTO Aarhus 2023

MAGNUS MADSEN

Outline

An introduction to the Flix programming language:

- ① Effect system
- ② Region-based memory
- ③ Iterators in action
- ④ Purity reflection
- ⑤ Ecosystem and tooling

The Flix Principle

In Flix the primary building block is a **function**. A function maps an input to an output.

Flix allows functions to be written in the most natural and/or efficient style ...

- Functionally (i.e. with immutable data structures)
- Imperatively (i.e. with mutable data structures) ← **Today**
- Declaratively (i.e. as a collection of logic constraints)

... without revealing these implementation to the clients.

①

Effect System

Type and Effect System

Flix has a **type and effect system** based on Hindley-Milner.

- The system supports type classes, higher-kinded types, and complete type inference.

The effect system *separates* **pure**, **impure**, and **effect polymorphic** expressions.

- The effect system is the basis for *purity reflection*.

Tracking **purity** has several benefits:

- It enables programmers to know when equational reasoning holds.
- It enables the Flix inliner to make more aggressive, but sound, choices.
- It enables the Flix standard library to know when it is safe to parallelize code (more on this later).

Purity

We can express that a function is *pure*:

```
def add(x: Int32, y: Int32): Int32 \ { } = ...  
                                     ^^^ empty effect
```

Here the implementation of `add` cannot have any side-effects.

We can also express that a higher-order function requires a *pure* function argument:

```
def count(f: a -> Bool \ { }, l: List[a]): Int32 \ { } = ...  
               ^^^ empty effect set           ^^^ empty effect
```

Here neither `f` nor `count` can have any side-effects.

Impurity

We can also express that a function is impure:

```
def sayHello(name: String): Unit \ { IO } =  
  println("Hello ${name}!")           ^^ printing is impure
```

It is a **type error** to annotate an **impure** function as **pure**:

```
def illegal() : Unit \ { } =  
  println("I am impure!")
```

```
✗ -- Type Error -----  
>> Impure function declared as pure.  
  
1 | def illegal() : Unit \ { } =  
  |     ^^^^^^^  
  |     impure function.
```

Effect Polymorphism

We can express that the effect of a higher-order function depends on its argument:

```
def map(f: a -> b \ ef, l: List[a]): List[b] \ ef = ...
```

 ^^ effect variable ^^ effect variable

The effect of map is the same as the effect of f.

```
// Pure use of map
List.map(x -> x * x + 42, l)
```

```
// Impure use of map
List.map(x -> println(x), l)
```


Catching Bugs

We can use the effect system to catch programming mistakes:

```
def main(): Unit \ { IO } =  
    checkPermission();  
    println("Access Granted")
```

✗ -- Redundancy Error ---

>> Useless expression: It has no side-effect(s) and its result is discarded.

```
4 |      checkPermission();  
   |      ^^^^^^^^^^^^^^^  
   |      useless expression.
```

The expression has type 'Bool'

① Summary: Effect System

The type and effect system enables us to write *pure*, *impure*, and *effect polymorphic* functions.

We can use the type and effect system to track and enforce purity:

- The Flix standard library enforces that the `eq`, `hash`, `compare`, and `toString` functions are pure.
- The Flix compiler uses purity information during variable and function inlining.

Reasoning about purity helps us reason and understand programs.

②

Region-based Memory

Region-based Memory

We have seen that Flix tracks purity.

- As soon as a function touches mutable memory it gets **tainted** with impurity.
- This is cumbersome because impurity then proliferates through the program.
- But what if the use of mutation is in some sense “local”.

💡 We associate all mutable data with a **region**:

- Reads and writes to data in a region are precisely tracked by the effect system.
- All effects related to a region vanish when the region goes out-of-scope.

Example: Sorting

```
///  
/// Sort the given list `l` so that elements are ordered from low to  
/// high according to their `Order` instance.  
///  
def sort(l: List[a]): List[a] with Order[a] =  
  region rc {  
    let arr = List.toArray(l, rc);  
    Array.sort!(arr);  
    Array.toList(arr)  
  }
```

1. Introduce a new region.
2. Allocate (mutable) data in the region.
3. Do imperative programming.
4. Return immutable data.

💡 Using an array-based (in place) sort is much faster than any list-based sort.

Example: Adding Two Numbers

```
///  
/// Returns the sum of `x` and `y`.  
///  
def sum(x: Int32, y: Int32): Int32 \ { } =  
    region rc {  
        let a = Array#{x, y} @ rc;  
        Array.swap!(0, 1, a);  
        a[1] + a[0]  
    }
```

Example: Swapping Elements

```
///  
/// Swap the elements at `i` and `j` in the array `a`.  
///  
def swap!(i: Int32, j: Int32, a: Array[t, r]): Unit \ { r } =  
    let x = a[i];  
    let y = a[j];  
    a[i] = y;  
    a[j] = x
```

👉 The region is part of the array type.

💡 The type and effect system tracks reads and writes to regions.

Example: ToString

```
///  
/// Returns a String representation of the given list `l`.  
///  
/// The returned String is of the form x1 :: x2 :: .. :: Nil.  
///  
def toString(l: List[a]): String with ToString[a] =  
  region r {  
    let sb = new StringBuilder(r);  
    List.foreach(x -> StringBuilder.appendString!("${x} :: ", sb), l);  
    StringBuilder.appendString!("Nil", sb);  
    StringBuilder.toString(sb)  
  }
```

💡 Using StringBuilders in toString functions is intuitive and efficient.

② Summary: Region-based Memory

Local mutation enables us to write **pure** functions that use **local mutable state**.

- We can implement functions in imperative style.
- We can use imperative style when it is more natural and/or more efficient.
- The type and effect system continues to ensure separation of pure and impure code!

💡 We can “pretend” to be **functional** programmers but use **imperative** style when we want!

We get the best of both worlds!

③

Iterators in Action

Iterators in Action

We illustrate the Flix type and effect system by showing how to express iterators.

An **iterator** is essentially a **mutable data structure** that represents a stream of elements.

- Iterators are useful because they allow **efficient** traversal through **homogenous collections**.

Iterators typically support both terminal (*eager*) and non-terminal (*lazy*) operations.

- Terminal operations include functions like count, fold, and sum.
- Non-terminal operations include functions like map and filter.

💡 the type and effect system clarifies *when* an effects happens.

The Iterator Data Type

```
/// An iterator has three type parameters:  
///  
/// a - the type of its elements.  
/// ef - the latent effect of the iterator.  
/// r - the region associated with the iterator.  
///  
enum Iterator[a: Type, ef: Eff, r: Region] { ... }
```

Latent Effects: What happens when this iterator is evaluated?

Associated Region: What memory does this iterator use?

Iterator API

```
///  
/// terminal (eager) iterator operations:  
///  
def sum(i: Iterator[Int32, ef, r]): Int32 \ { ef, r }  
  
def toList(i: Iterator[a, ef, r]): List[a] \ { ef, r }  
  
def forEach(f: a -> Unit \ ef2, i: Iterator[a, ef1, r]): Unit \ { ef1, ef2, r }  
  
///  
/// non-terminal (lazy) iterator operations:  
///  
def filter(f: a -> Bool \ ef2, i: Iterator[a, ef1, r]): Iterator[a, {ef1, ef2}, r]  
  
def map(f: a -> b \ ef2, i: Iterator[a, ef1, r]): Iterator[b, {ef1, ef2}, r]
```

Using Iterators

```
def main(): List[Int32] =  
  let s = Set#{1, 2, 3};  
  region rc {  
    Set.iterator(rc, s) |>  
    Iterator.map(x -> { x + 1 }) |> // Lazy  
    Iterator.toList                // Eager  
  }  
  
  /// Returns 2 :: 3 :: 4 :: Nil
```

Using Iterators with Latent Effects

```
def main(): List[Int32] \ IO =  
  let s = Set#{1, 2, 3};  
  region rc {  
    Set.iterator(rc, s) |>  
    Iterator.map(x -> { println("Hi!"); x + 1}) |>  
    Iterator.toList  
  }  
  
// Prints "Hi!", "Hi!", "Hi!" and then returns 2 :: 3 :: Nil
```

The Collectable Type Class

```
///  
/// A class for collections that can be produced from an `Iterator`.  
///  
class Collectable[m: Type -> Type] {  
  
    ///  
    /// Collect the elements of `iter` into `m[a]`.  
    ///  
    def collect(iter: Iterator[a, ef, r]): m[a] \ {ef, r} with Order[a]  
  
}
```

Instances for: List, Set, Map, etc...

Using Collectable

```
def desserts(): Set[(String, String)] =  
  let fruits = List#{"Apple", "Pear", "Mango"};  
  let creams = List#{"Vanilla", "Stracciatella"};  
  region rc {  
    let iter1 = List.iterator(rc, fruits);  
    let iter3 = Iterator.flatMap(fruit -> {  
      let iter2 = List.iterator(rc, creams);  
      Iterator.map(cream -> (fruit, cream), iter2)}, iter1);  
    Collectable.collect(iter3)  
  }  
  
  /// Set#{(Apple, Stracciatella), (Apple, Vanilla),  
  ///      (Mango, Stracciatella), (Mango, Vanilla),  
  ///      (Pear, Stracciatella), (Pear, Vanilla)}
```

Syntactic Sugar: Foreach-Yield

```
def desserts(): Set[(String, String)] =  
  let fruits = List#{"Apple", "Pear", "Mango"};  
  let creams = List#{"Vanilla", "Stracciatella"};  
  foreach(fruit <- fruits; cream <- creams)  
    yield (fruit, cream)
```

1. Regions are implicit.
2. Iterators are flatMap'ed.
3. Uses **Collectable** to build a new collection.

A “Real-World” Example

```
def solve(rows: Int32, cols: Int32, pieces: List[Piece]): Set[Solution] =  
  match pieces {  
    case Nil => Set#{Set#{}}  
    case piece :: rest =>  
      foreach (  
        solution <- solve(rows, cols, rest);  
        x <- Set.range(0, rows);  
        y <- Set.range(0, cols);  
        pos <- Some((x, y));  
        if allowed(piece, pos, solution)  
      ) yield Set.insert((piece, pos), solution)  
  }
```

Thanks to Paul Butcher:

<https://github.com/paulbutcher/chess-flux/blob/master/src/Main.flix>

③ Summary: Iterators in Action

We can express iterators in Flix.

- Iterators support both terminal (*eager*) and non-terminal (*lazy*) operations.
- The Flix type and effect system precisely describes *when* effects happens.

We can use **Collectable** to describe collections that can be built from iterators.

A bit of **syntactic sugar** makes the **medicine go down**:

- We can hide the region capability, the creation of iterators, and the use of Collectable.
- **Upshot:** We get clean syntax, excellent performance, and code that is *pure* to the outside world.

4

Purity Reflection

Purity Reflection

Program evaluation in Flix (and most programming languages) is eager and sequential.

- But it would be useful if library authors could take advantage of lazy and/or parallel evaluation.
- But when is it safe to evaluate a function lazily or in parallel?

Many programming languages have **streams**:

- But the combination of streams and side-effects is a dangerous cocktail.
- We risk race conditons, deadlocks, lost and/or re-ordered side effects!

Can we do better?

💡 What if we allow data structure operations (map, filter, etc.) to vary their behavior depending on the purity of their function arguments?

Selective Parallelism

We can write a map function that uses selective parallelism:

```
def map(f: a -> b \ ef, t: Map[k, a]): Map[k, b] \ ef =  
  typematch f {  
    case g: (k, a) -> b \ {} => parMap(g, t)  
    case g: (k, a) -> b \ ef => seqMap(g, t)  
  }
```

Example I

```
def main(): Map[Int32, Int32] \ IO =  
  let keys    = List.range(1, 1_000);  
  let values  = List.range(1, 1_000);  
  let m = List.toMap(List.zip(keys, values));  
  
  Map.map(v -> {println(v); v + 1}, m)  
  
  // Prints 1, 2, 3, ... and returns a Map#{1 -> 2, 2 -> 3, ...}
```



Evaluation is sequential (since the function passed to map is impure).

This ensures that the order of side-effects is preserved (i.e. we print 1, 2, 3).

Example II: Automatic Parallelization

```
def main(): Map[Int32, Int32] =  
  let keys    = List.range(1, 1_000);  
  let values  = List.range(1, 1_000);  
  let m = List.toMap(List.zip(keys, values));  
  
  Map.map(v -> {v + 1}, m)  
  
  // Returns a Map#{1 -> 2, 2 -> 3, ...}
```



Evaluation is parallel (since the function passed to map is pure).

Upshot: the Map is rebuilt using all cores of the machine.

Selective Laziness

We can also write a map function that uses selective laziness:

```
def map(f: a -> b \ ef, l: DelayList[a]): DelayList[b] \ ef =  
  typematch f {  
    case g: a -> b \ {} => mapL(g, l)  
    case g: a -> b \ ef => mapE(g, l)  
  }
```

If f is pure then we use `mapL` to apply it lazily over the list.

If f is impure then we use `mapE` to apply it eager over the list (materializing all effects).

A Fresh Take on Data Transformations

Principle: Data structure operations (such as map, filter, ...)

- Use lazy and/or parallel evaluation when given pure function arguments.
- Use eager sequential evaluation when given impure function arguments.

This ensures that side-effects are not lost and that the order of side-effects is preserved.

④ Summary: Purity Reflection

Purity reflection enables higher-order functions to *inspect the purity* of their function argument(s) and to *vary their behavior* based on this information.

We can exploit this information to parallelize certain operations on sets and maps.

- Examples are count, map, or mapWithKeys.

We can also use this information to implement new and novel data structures:

- **DelayList** a list that is maximally lazy except when given impure functions.
- **DelayMap** a map that is lazy in its values and uses parallel evaluation for bulk operations.

5

Ecosystem & Tooling

Visual Studio Code Support

We support most Visual Studio Code features, including:

- ✓ syntax highlighting
- ✓ inline diagnostics
- ✓ auto-complete
- ✓ type and effect hover
- ✓ find references
- ✓ jump to definition
- ✓ rename
- ✓ code hints
- ✓ code lenses (e.g. “click to run”)
- ✓ document symbols
- ✓ workspace symbols
- ✓ highlight related symbols
- ✓ incremental compilation

File

Edit

Selection

View

Go

Run

Terminal

Help

EXPLORER

...

FLIX

> lib

> src

Main.flix

> target

> test

HISTORY.md

LICENSE.md

README.md

src > Main.flix > ...

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

/// An example using Datalog constraints enriched with lattice semantics to

/// compute the delivery date of a part based on delivery dates of its components.

Run | Run with args... | Run (in new terminal) | Run with args... (in new terminal)

def main(): Unit & Impure =

let p = #[

/// Parts and the components they depend on.

PartDepends("Car", "Chassis").

PartDepends("Car", "Engine").

PartDepends("Engine", "Piston").

PartDepends("Engine", "Ignition").

/// The time required to assemble a part from its components.

AssemblyTime("Car", 7).

AssemblyTime("Engine", 2).

/// The expected delivery date for certain components.

DeliveryDate("Chassis"; 2).

DeliveryDate("Piston"; 1).

DeliveryDate("Ignition"; 7).

/// A part is ready when it is delivered.

ReadyDate(part; date) :-

DeliveryDate(part; date).

/// Or when it can be assembled from its components.

ReadyDate(part; assemblyTime + componentDate) :-

PartDepends(part, component),

AssemblyTime(part, assemblyTime),

ReadyDate(component; componentDate).

];

// Computes a map from parts to delivery dates.

let m = query p select (c, d) from ReadyDate(c; d) ▷ Array.toMap;

// Looks up the delivery date for the car and prints it.

Map.getWithDefault("Car", 0, m) ▷ println

PROBLEMS

OUTPUT

DEBUG CONSOLE

TERMINAL

Flix Compiler

≡

🔍

📄

⬆

⬇

⬇

⬇

LSP listening on: 'localhost/127.0.0.1:8888'.

Flix 0.28.0 Ready! (Extension: 0.74.0) (Using c:\Users\iostream\AppData\Roaming\Code\User\globalStorage\flix.flix\flix.jar)

OUTLINE

TIMELINE

0

0

0

Flix 0.28.0 Ready! (Extension: 0.74.0) (Using c:\Users\iostream\AppData...

39

Ln 36, Col 1

Spaces: 4

UTF-8

CRLF

Flix

🔍

📄

The Flix Programming Language

Next-generation reliable, safe, concise, and functional-first programming language.

Flix is a principled functional, imperative, and logic programming language developed at [Aarhus University](#), at the [University of Waterloo](#), and by a community of [open source contributors](#).

Flix is inspired by OCaml and Haskell with ideas from Rust and Scala. Flix looks like Scala, but its type system is based on Hindley-Milner. Two unique features of Flix are its polymorphic effect system and its support for first-class Datalog constraints.

Flix compiles to JVM bytecode, runs on the Java Virtual Machine, and supports full tail call elimination. A VSCode plugin for Flix is available.

Get Started Playground Documentation Library

Why Flix?

Flix aims to offer a **unique combination of features** that no other programming language offers, including: **algebraic data types and pattern matching** (like Haskell, OCaml), **extensible records** (like Elm), **type classes** (like Haskell, Rust), **higher-kinded types** (like Haskell), **typematch** (like Scala), **type inference** (like Haskell, OCaml), **structured channel and process-based concurrency** (like Go), **a polymorphic effect system** (a unique feature), **region-based local mutation** (a unique feature), **purity reflection** (a unique feature), **first-class Datalog constraints** (a unique feature), and **compilation to JVM bytecode** (like Scala).

Algebraic Data Types and Pattern Matching

Algebraic data types and pattern matching are the bread-and-butter of functional programming and are supported by Flix with minimal fuss.

```
def origin(): (Int32, Int32) = (0, 0)
```

Run Algebraic Data Types and Pattern Matching

```
/// An algebraic data type for shapes.
enum Shape {
  case Circle(Int32),           // circle radius
  case Square(Int32),           // side length
  case Rectangle(Int32, Int32) // height and width
}

/// Computes the area of the given shape using
/// pattern matching and basic arithmetic.
def area(s: Shape): Int32 = match s {
  case Shape.Circle(r)      => 3 * (r * r)
  case Shape.Square(w)       => w * w
  case Shape.Rectangle(h, w) => h * w
}

// Computes the area of a 2 by 4.
def main(): Unit \ IO =
  println(area(Shape.Rectangle(2, 4)))
```

```
enum Shape {
  case Circle(Int32),
  case Square(Int32),
  case Rectangle(Int32, Int32)
}

def area(s: Shape): Int32 = match s {
  case Circle(r)      => 3 * (r * r)
  case Square(w)       => w * w
  case Rectangle(h, w) => h * w
}
```

Tuples and Records

Fork me on GitHub



Design Principles

We believe that the development of a programming language should follow a set of principles. That is, when a design decision is made there should exist some rationale for why that decision was made. By outlining these principles, as we develop Flix, we hope to keep ourselves honest and to communicate the kind of language Flix aspires to be.

Many of these ideas and principles come from languages that have inspired Flix, including Ada, Elm, F#, Go, Haskell, OCaml, Rust, and Scala.

Update: The Flix Principles has been published in a paper at Onward! '22. Read it here: [The Principles of the Flix Programming Language](#).

Language Principles

Simple is not easy

We believe in Rich Hickey's creed: [simple is not easy](#). We prefer a language that gets things right to one that makes things easy. Such a language might take longer to learn in the short run, but its simplicity pays off in the long run.

Human-readable errors

In the spirit of [Elm](#) and [Rust](#), Flix aims to have human readable and understandable compiler messages. Messages should describe the problem in detail and provide information about the context, including suggestions for how to correct the problem.

No null value

Flix does not have the `null` value. The null value is now widely considered a mistake and languages such as C#, Dart, Kotlin and Scala are scrambling to adopt mechanisms to ensure non-nullness. In Flix, we adopt the standard solution from functional languages which is to represent the absence of a value using the `Option` type. This solution is simple to understand, works well, and guarantees the absence of dreaded `NullPointerExceptions`.

Everything is an expression

Flix is a functional language and embraces the idea that everything should be an expression. Flix has no local variable declarations or if-then-else statements, instead it has let-bindings and if-then-else expressions. However, Flix does not take this idea as far as the Scheme languages. Flix still has declarations, namespaces, and so forth that are not expressions.

Private by default

Flix embraces the principle of least privilege. In Flix, declarations are hidden by default (i.e. private) and cannot be accessed from outside of their namespace (or sub-namespaces). We believe it is important that programmers are forced to make a conscious choice about when to make a declaration publicly visible.

No implicit coercions

In Flix, a value of one type is never implicitly coerced or converted into a value of another type. For example,

- No value is ever coerced to a boolean.
- No value is ever coerced to a string.
- Integers and floating-point are never truncated or promoted.

Separate pure and impure code

Flix supports functional, imperative, and logic programming. The type and effect system of Flix cleanly and safely

Closed world assumption

Flix requires all code to be available at compile-time. This enables a range of compilation techniques, such as:

Flix Playground

play.flix.dev

Google

Share

Star

1

M

Compile & Run

[Website](#) [Documentation](#) [Standard Library](#) [Shareable Link](#)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16


17

18

19

```
1  /// An algebraic data type for shapes.
2  enum Shape {
3    case Circle(Int32),      // circle radius
4    case Square(Int32),      // side length
5    case Rectangle(Int32, Int32) // height and width
6  }
7
8  /// Computes the area of the given shape using
9  /// pattern matching and basic arithmetic.
10 def area(s: Shape): Int32 = match s {
11   case Shape.Circle(r)      => 3 * (r * r)
12   case Shape.Square(w)      => w * w
13   case Shape.Rectangle(h, w) => h * w
14 }
15
16 // Computes the area of a 2 by 4.
17 def main(): Unit \ IO =
18   println(area(Shape.Circle(2)))
19
```

Standard Output



Introduction to Flix - Programmir

doc.flix.dev

iosstream.dk

flix

flix

1. Introduction to Flix

2. Getting Started

2.1. Hello World!

2.2. Next Steps

3. Data Types

3.1. Primitives

3.2. Tuples

3.3. Enums

3.4. Type Aliases

4. Functions

5. Immutable Data

5.1. Lists

5.2. Chains and Vectors

5.3. Sets and Maps

5.4. Records

6. Mutable Data

6.1. Regions

6.2. References

6.3. Arrays

6.4. Collections

7. Control Structures

7.1. If-Then-Else

7.2. Pattern Matching

7.3. Foreach

7.4. Foreach-Yield

7.5. Monadic For-Yield

7.6. Applicative For-Yield

8. Structured Concurrency

9. Parallelism

10. Effect System

11. Modules

11.1. Declaring Modules

11.2. Using Modules

11.3. Companion Modules

12. Type Classes

Programming Flix

Introduction to Flix

Flix is a principled functional, logic, and imperative programming language developed at [Aarhus University](#) and by a community of [open source contributors](#) in collaboration with researchers from the [University of Waterloo](#), from the [University of Tübingen](#), and from the [University of Copenhagen](#).

Flix is inspired by OCaml and Haskell with ideas from Rust and Scala. Flix looks like Scala, but its type system is based on Hindley-Milner which supports complete type inference. Flix aims to offer a unique combination of features that no other programming language offers, including:

- algebraic data types and pattern matching.
- type classes with higher-kinded types.
- structured concurrency based on channels and light-weight processes.

In addition, Flix has several new powerful and unique features:

- A polymorphic type and effect system with region-based local mutation.
- Datalog constraints as first-class program values.
- Function overloading based on purity reflection.

Flix compiles to efficient JVM bytecode, runs on the Java Virtual Machine, and supports full tail call elimination. Flix has interoperability with Java and can use JVM classes and methods. Hence the entire Java ecosystem is available from within Flix.

Flix aims to have world-class Visual Studio Code support. The [Flix Visual Studio Code extension](#) uses the real Flix compiler hence there is always a 1:1 correspondence between the Flix language and what is reported in the editor. The advantages are many: (a) diagnostics are always exact, (b) code navigation "just works", and (c) refactorings are always correct.

Look'n Feel

Here are a few programs to illustrate the look and feel of Flix:

This program illustrates the use of **algebraic data types** and **pattern matching**:

```
/// An algebraic data type for shapes.
enum Shape {
  case Circle(Int32),           // circle radius
  case Square(Int32),           // side length
  case Rectangle(Int32, Int32) // height and width
}

/// Computes the area of the given shape using
/// pattern matching and basic arithmetic
```

Prelude

api.flix.dev

iosstream.dk

flix

flix

flix0.36.0

Filter

> Prelude

> Applicative

> Array

> Assert

> Benchmark

> BigDecimal

> BigInt

> Bool

> Boxable

> Chain

> Channel

> Char

> Choice

> Comparison

> Console

> DelayList

> DelayMap

> Environment

> File

> Fixpoint

> Fixpoint.Ast

> Float32

> Float64

> Foldable

> Functor

> GetOpt

> Graph

> Hash

> Int16

> Int32

> Int64

> Int8

> Iterator

> List

> Map

> Monad

> Monoid

> MultiMap

> MutDeque

> MutList

> MutMap

> MutSet

> Nec

> Nel

> Object

Prelude

Classes

classAdd[a : Type]A type class for addition.

Signatures (hide)def add(x: a, y: a): a with Add[a]

Instances (show)

classApplicative[m : Type → Type] with Functor[m]A type class for functors that support application, i.e. allow to:

- Make an applicative value out of a normal value (embed it into a default context), e.g. embed 5 into Some(5).
- Apply a function-type applicative to a matching argument-type applicative, resulting in an applicative of the function's result type.

The meaning of the application realized by the ap function is defined by the respective instance. Conceptually this can be understood as applying functions "contained" in the first applicative to arguments in the second applicative, where the possible quantity of functions/arguments depends on the type m. For example, an Option[a → b] can be None, or contain a function of type a → b, and only in the latter case a function is applied. A List[a → b] is an applicative that contains a list of functions, which are all to be applied to all arguments contained in the arguments list.

A minimal implementation must define point and at least one of ap and map2 (if map2 is implemented, ap can be defined based on map2 as shown below). If both ap and map2 are defined, they must be equivalent to their default definitions: ap(f: m[a → b & e], x: m[a]): m[b] \ ef = map2(identity, f, x) map2(f: a → b → c & e, x: m[a], y: m[b]): m[c] \ ef = ap(Functor.map(f, x), y)

Signatures (hide)def ap[a, ef, b](f: m[a → b \ ef], x: m[a]): m[b] \ ef with Applicative[m]def point[a](x: a): m[a] with Applicative[m]

Definitions (show)

Instances (show)

classCloseable[a : Type]A type class for types that can be closed.

Signatures (hide)def close(x: a): Unit \ IO with Closeable[a]

Instances (show)

master

7 branches

53 tags

Go to file

Add file


<> Code

sockmaster27 chore: remove re1 and lat (#5931)

b6562b5 54 minutes ago 8,272 commits

.github/workflows	ci: update community build (#5849)	2 weeks ago
docs	feat: release 0.36.0 (#5761)	2 weeks ago
examples	chore: remove re1 and lat (#5931)	54 minutes ago
gradle/wrapper	chore: update gradle: 7.6 → 8.1 (#5756)	3 weeks ago
lib	chore: Switch from jar files in lib to Gradle dependencies (#5211)	4 months ago
main	chore: remove re1 and lat (#5931)	54 minutes ago
.editorconfig	Add .editorconfig (#2676)	2 years ago
.gitattributes	chore: add .gitattributes (#2723)	2 years ago
.gitignore	chore: add crash_report_*.txt to .gitignore (#3655)	last year
AUTHORS.md	feat: add String.stripMarginWith (#5895)	4 days ago
LICENSE.md	Added license.	8 years ago
README.md	chore: update png height (#3225)	last year
build.gradle	fix: disable SLF4J warning (#5912)	5 days ago
gradlew	chore: update gradle: 7.6 → 8.1 (#5756)	3 weeks ago
gradlew.bat	chore: update gradle: 7.6 → 8.1 (#5756)	3 weeks ago

☰ README.md



About

The Flix Programming Language

flix.dev/

language programming-language functional jvm logic flx hacktoberfest imperative

Readme

View license

1.7k stars

22 watching

123 forks

Report repository

Releases 52

Version 0.36.0 Latest

2 weeks ago

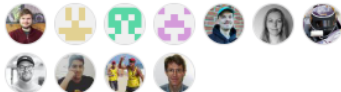
+ 51 releases

Packages

No packages published

[Publish your first package](#)

Contributors 60



⑥

Wrapping Up

Additional Resources

The Official Flix Website:	<u>https://flix.dev/</u>
The Programming Flix Book:	<u>https://doc.flix.dev/</u>
API Documentation:	<u>https://api.flix.dev/</u>
GitHub:	<u>https://github.com/flix/flix</u>
InfoQ Article:	<u>https://tinyurl.com/infoq-flix</u>
Happy Path Podcast:	<u>https://tinyurl.com/happypath-flix</u>

The Flix Team



Magnus



Ondřej



Jaco

... and all our open-source contributors!



Jonathan



Matt



Andreas



Jakob

... And thanks to our sponsors!



Summary (1/2)

Flix is a new **functional**, **imperative**, and **logic** programming language.

Flix aims to offer a **unique combination** of features that no other existing language offers.

- algebraic data types and pattern matching
 - tuples and extensible records
 - parametric polymorphism
 - higher-kinded types and type classes
 - a polymorphic effect system
 - purity reflection
 - region-based memory
- channel and process-based concurrency
 - first-class Datalog program values
 - Hindley-Milner style type inference
 - full tail call elimination
 - an extensive standard library
 - Visual Studio Code support
 - ... and more

Summary (2/2)

Flix is a new **functional**, **imperative**, and **logic** programming language.

Flix is ready for use! Try it out!

- Flix has a fully-featured Visual Studio Code extension.
- Flix has a website, online documentation, and a playground.
- The Flix Standard Library is extensive.

Thank You!

Flix is open source, freely available, and ready for use:

<https://flix.dev/>