An Introduction to the **Flix** Programming Language

Dahl-Nygaard Prize Talk

MAGNUS MADSEN

The Flix Team

Thanks to our sponsors:

Magnus

Ondřej

Jaco

Jonathan

Matt

Jakob

Nina

Outline

An introduction to the Flix programming language:

① First-class Datalog

② Effect system

③ Local mutation

④ Purity reflection

⑤ Design principles

⑥ 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)
- Declaratively (i.e. as a collection of logic constraints)

… without revealing these implementation to the clients.

①

First-class Datalog

Introduction to Datalog

Datalog is a simple, yet surprisingly powerful declarative logic programming language.

- A bit like SQL but with recursion.

Datalog has several important properties:

1. Every Datalog program eventually terminates.
2. Every Datalog program has a unique solution.
3. Datalog has efficient and parallel evaluation strategies.

Datalog has successfully been used in a wide-variety of applications:

- e.g., program analysis, security analysis, data analytics, bio-informatics.
- Datalog has often been used for very specialized applications that have high complexity and/or performance requirements.

Introduction to Datalog (cont'd)

A simple query over two relations:

```
Bird("Eagle").
Bird("Gentoo").
Flying("Eagle").

Penguin(x) :- Bird(x), not Flying(x).
```

A recursive query over two relations:

```
Direct("BLL", "FRA").
Direct("FRA", "YYZ").
Direct("YYZ", "YVR").

Connected(src, dst) :- Direct(src, dst).
Connected(src, dst) :-
  Connected(src, hop), Direct(hop, dst).
```

Vancouver
May 10, 2022
Canada
United States
United Kingdom
Deutschland
France
España
Ísland
México

Motivation

The Problem: Using Datalog is cumbersome!

- I don’t want to write my entire program in Datalog.
- I don’t want to generate a text file, pass it to a Datalog solver, and parse the result.
- I don’t want to use FFI for a 10-line program.
- I want all the conveniences of a real programming language (a type system, an IDE, tools, etc.)
- I want to write *modular* and *reusable* programs.

Key Idea:

Datalog programs as *first-class* values in a functional programming language.

Example: Datalog in Flix

```
def connected(routes: List[(String, String)]): List[(String, String)] =
    let db = project routes into Direct;
    let pr = #{
        Connected(src, dst) :- Direct(src, dst).
        Connected(src, dst) :- Connected(src, hop), Direct(hop, dst).
    };
    query db, pr select (x, y) from Connected(x, y) |> Array.toList
```

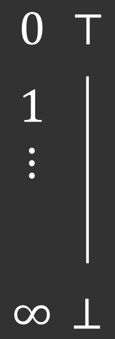
Example: Polymorphic Datalog

```
def reachable(edges: List[(t, t)], src: t, dst: t): Bool =
    let db = project edges into Edge;
    let pr = #{
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
        Reachable() :- Path(src, dst).
    };
    let result = query db, pr select () from Reachable();
    not Array.isEmpty(result)
```

Example: Lattice Semantics

```
def shortestDist(origin: t, edges: List[(t, Int32, t)]): Map[t, N] =
    let db = project edges into Edge;
    let pr = #{
        Dist(origin; N(0)).
        Dist(y; add(d1, d2)) :- Dist(x; d1), Edge(x, d2, y).
    };
    query db, pr select (x, d) from Dist(x; d) |> Array.toMap
```

```
enum N { case N(Int32) }

instance PartialOrder[N] {
    def lessEqual(x: N, y: N): Bool = ...
}

// LowerBound, JoinLattice, MeetLattice, ...
```



Here $N = (\infty, 0, \geq, \min, \max)$

① Summary: First-Class Datalog

First-class Datalog program values enable us to use Datalog where it shines: to declaratively express and solve fixed-point constraints.

- We can write *functions* that internally use Datalog program values.
- We can write modular and reusable Datalog programs using polymorphism.
- The type system ensures that our Datalog programs are consistent and stratified.

Flix supports *constraints on relations*, but also *constraints on lattices*.

- We can solve even more fixed-point problems, including program analyses and two-player games.

②

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.

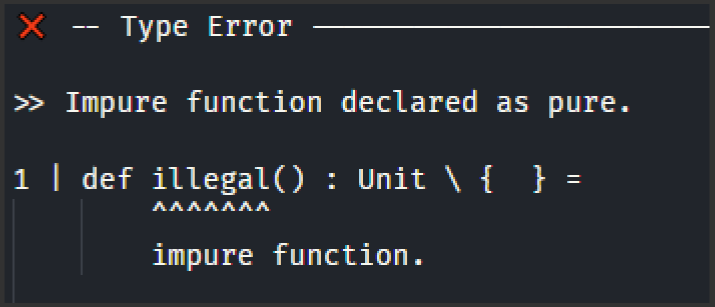
Impurity

We can also express that a function is impure:

```
def sayHello(name: String): Unit \ { Impure } =
    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)
```

First-class Functions

We support first-class functions. We can express function composition as:

```
   def >>(f: a -> b \ ef1, g: b -> c \ ef2): a -> c \ {ef1, ef2} = ...
                                                         ^^^^^^^^^ union effect
```

The effect of the returned function is pure if `f` and `g` are pure.

② 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.

Related Work:

- Polymorphic effect systems [**Lucassen et al., ‘88**]
- Region-based memory management [**Tofte et al., ‘97**]
- Programming with algebraic effects and handlers [**Plotkin et al., ‘15**]

③

Local Mutation

Local Mutation

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".

Can we do better?

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 r {
        let arr = List.toArray(l, r);
        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 r {
        let a = [x, y] @ r;
        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 \ { Read(r), Write(r) } =
    let x = a[i];                         ^ region      ^^^^^^^^^^^^^^^^^ effect
    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.

③ Summary: Local Mutation

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!

Related Work:

- Linear types can change the world! [**Wadler et al., '90**]
- Imperative functional programming [**Jones et al., '93**]
- Lazy functional state threads [**Jones et al., '94**]

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?

Example: Selective Laziness

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

```
def map(f: a -> b & ef, l: DelayList[a]): DelayList[b] & ef =
    reifyEff(f) {
        case Pure(g) => mapL(g, l)
        case _       => mapE(f, 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).

Example I

The Flix program fragment:

```
DelayList.range(1, 1_000_000_000) |>
DelayList.map(x -> {println("a"); x + 1}) |>
DelayList.map(x -> {println("b"); x * 2})
```

Prints one billion a’s followed by one billion b’s.

This takes a while, but ultimately the program terminates.

The a’s are printed before the b’s preserving the order of effects.

Example II

The Flix program fragment:

```
DelayList.range(1, 1_000_000_000) |>
DelayList.map(x -> x + 1) |>
DelayList.map(x -> x * 2) |>
DelayList.head |> println
```

Prints `Some(4)` and terminates immediately.

The two map operations are pure, consequently they are applied lazily.

Example III

The Flix program fragment:

```
let count = ref 0;
List.range(1, 1_000_000_000) |>
List.map(x -> x + 1) |>
List.take(1_000) |>
List.map(x -> { count := deref count + 1; x * 2});
println(deref count)
```

Prints 1000 and terminates very quickly.

The first map operation is applied lazily and the subsequent take operation is applied lazily.

The final map operation is applied eagerly, but only to the first 1000 elements.

Example: Selective Parallelism

We can also write a `map` function that uses selective parallelism:

```
def mapWithKey(f: (k, v1) -> v2 & ef, t: RBTree[k, v1]): ... =
    reifyEff(f) {
        case Pure(g) => parMapWithKey(g, t)
        case _       => seqMapWithKey(f, t)
    }
```

We use this function in the implementation of the `Set` and `Map` data structures.

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 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.

Related Work:

- First-class effect reflection for effect-guided programming [**Long et al., '16**]

5

Design Principles

The Flix Design Principles

A set of **design choices** or **design principles** collected over time.

Based on:

- Discussions on GitHub.
- Discussions on other programming language forums.
- Perceived mistakes of other programming languages.
- Feedback from users.

Listed on the Flix website for reference and to keep us honest.

- We have now collected more than **59** such principles.

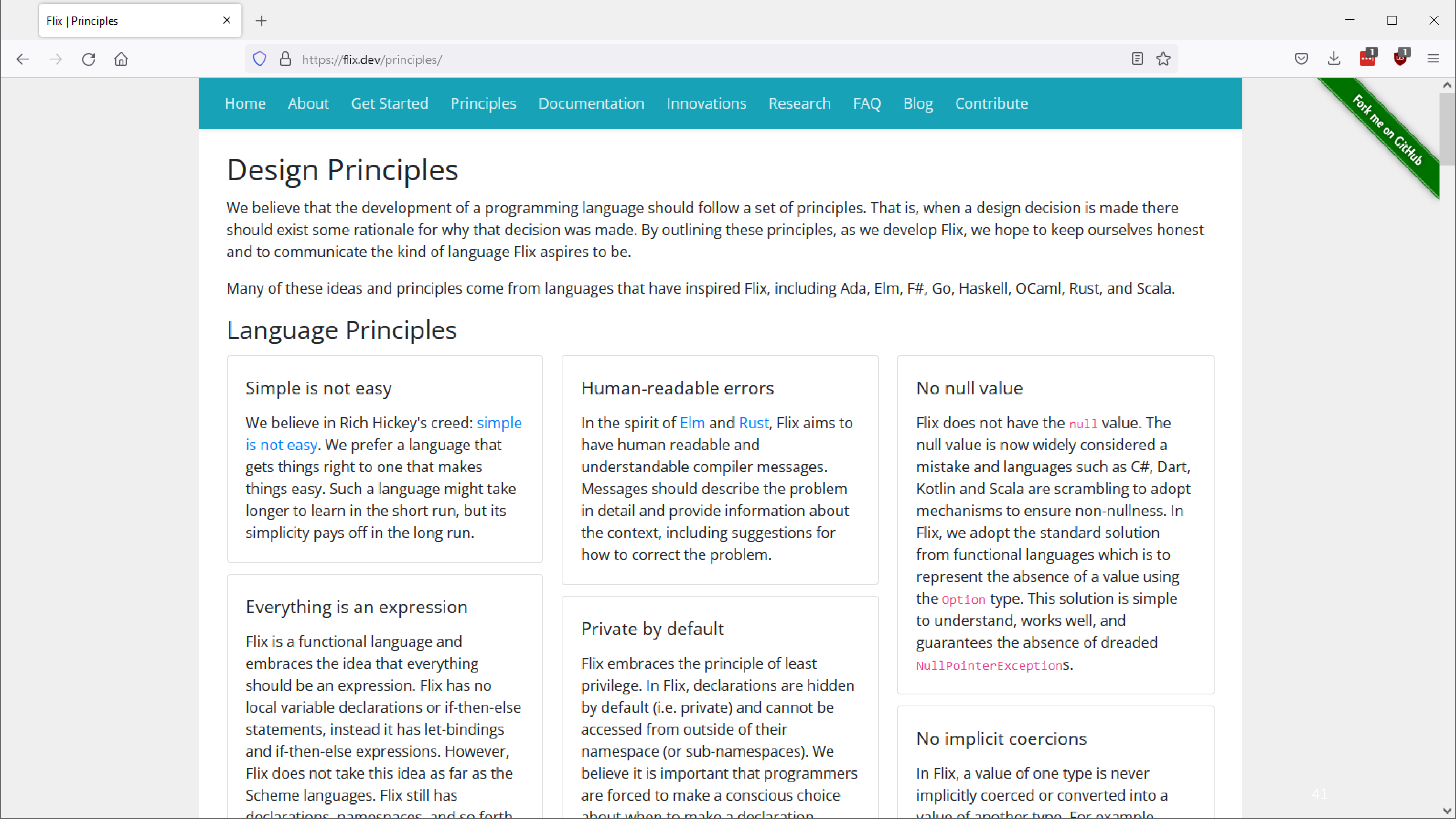
Principle: No warnings, only errors.

Principle: No unused declarations.

Principle: Declaration monotonicity.

```
namespace A {
    def inc(x: Int32): Int32 = x + 1
}

namespace B {
    // ...
}
```

```
def main(): Int32 =
    use A._;
    use B._;
    inc(1)
```



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.

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.

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

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.

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

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 `NullPointerException`s.

No implicit coercions

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

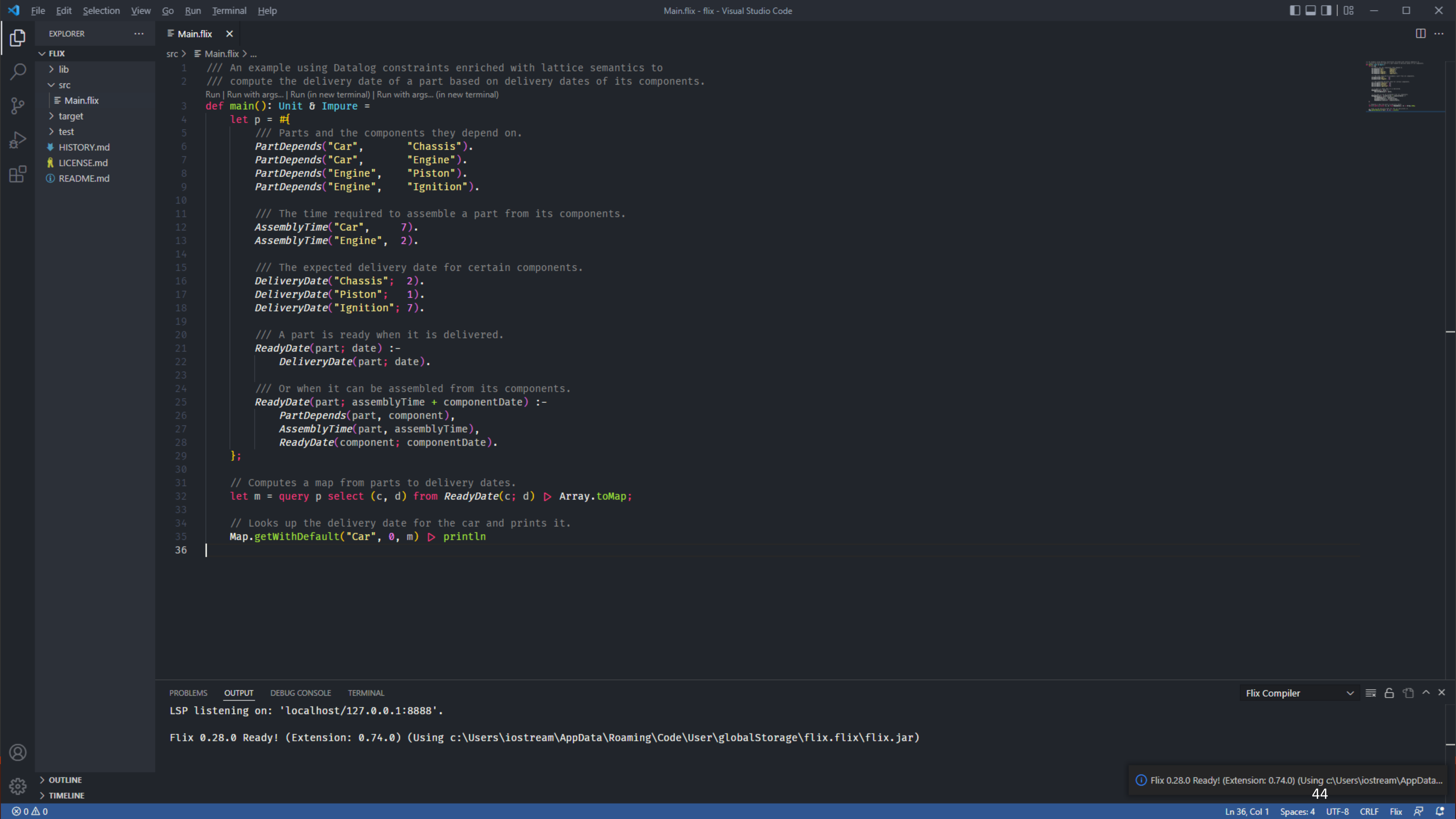
⑥

Ecosystem & Tooling

Visual Studio Code Support

We support most Visual Studio Code features, including:

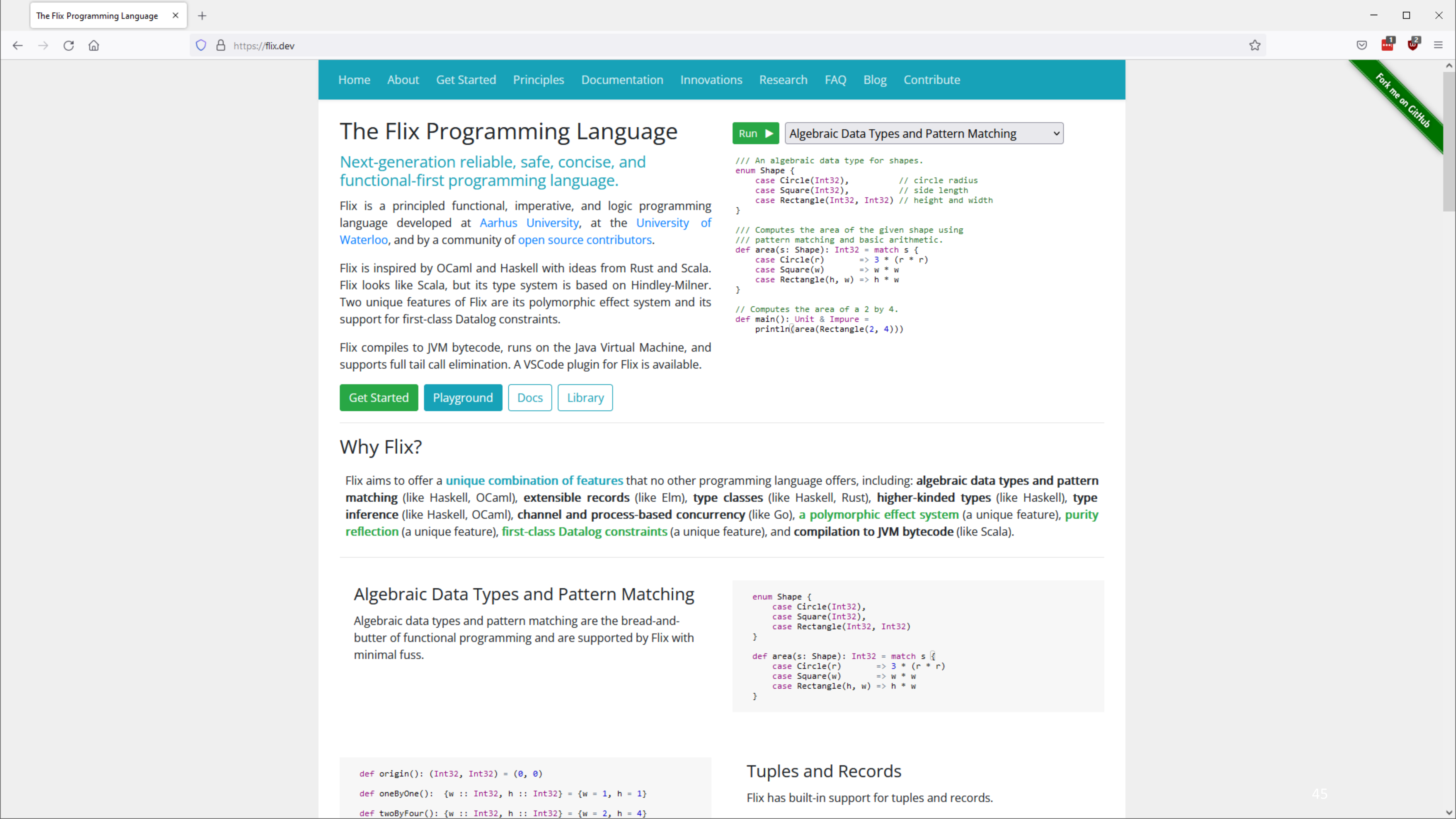
- ✓ syntax highlighting
- ✓ inline diagnostics
- ✓ auto-complete
- ✓ type and effect hover
- ✓ find references
- ✓ jump to definition
- ✓ rename
- ✓ code hints
- ✓ code lenses (e.g. "click to run")
- ✓ document symbols
- ✓ workspace symbols
- ✓ highlight related symbols
- ✓ incremental compilation



```
/// An example using Datalog constraints enriched with lattice semantics to
/// compute the delivery date of a part based on delivery dates of its components.
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

```

```
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)
```

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


The Flix Programming Language
https://flix.dev
Home About Get Started Principles Documentation Innovations Research FAQ Blog Contribute
Fork me on GitHub

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.

```
/// 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 Circle(r)       => 3 * (r * r)
    case Square(w)       => w * w
    case Rectangle(h, w) => h * w
}

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

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), **type inference** (like Haskell, OCaml), **channel and process-based concurrency** (like Go), **a polymorphic effect system** (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

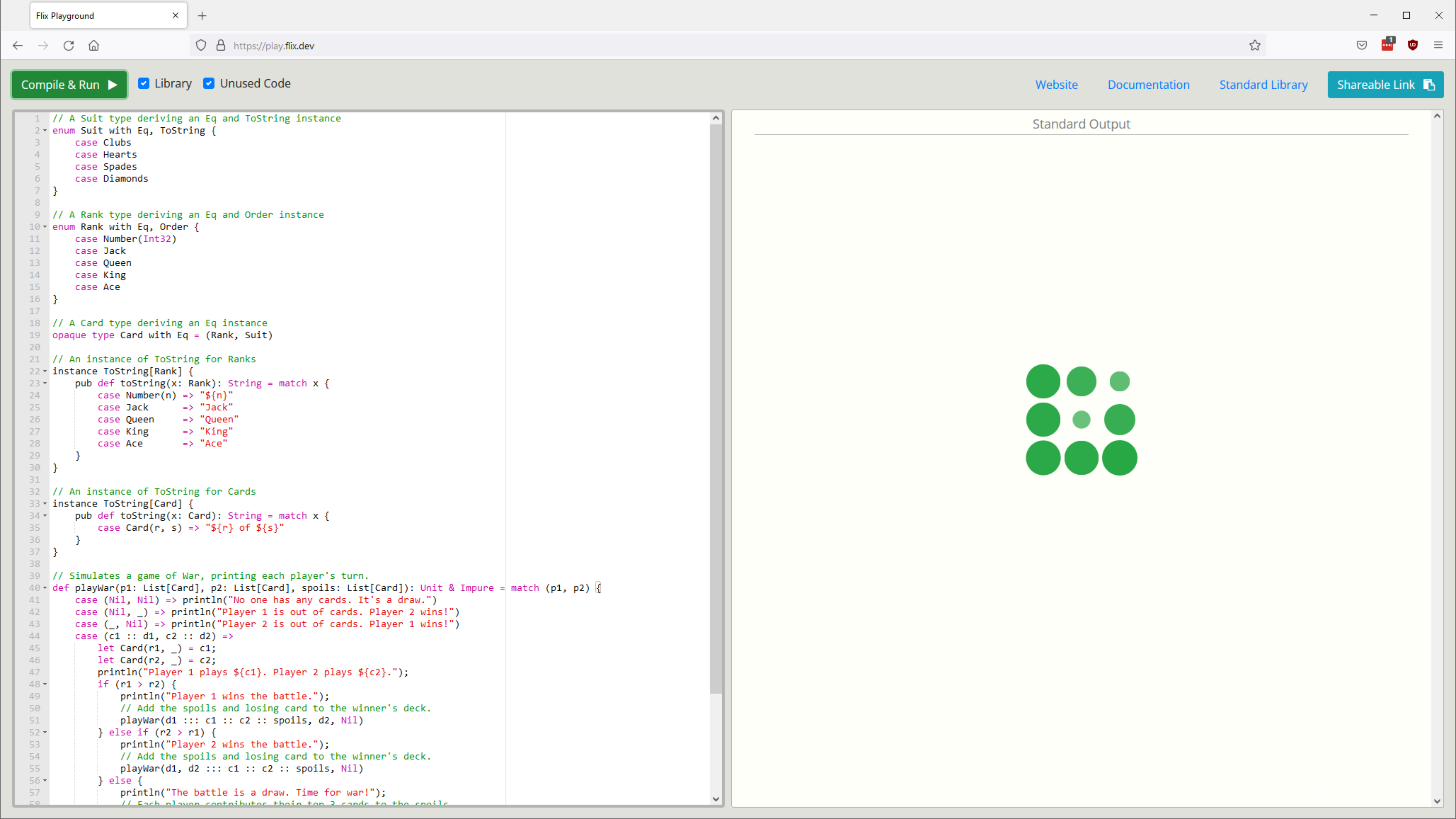
Algebraic data types and pattern matching are the bread-and-butter of functional programming and are supported by Flix with minimal fuss.

```
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
}
```

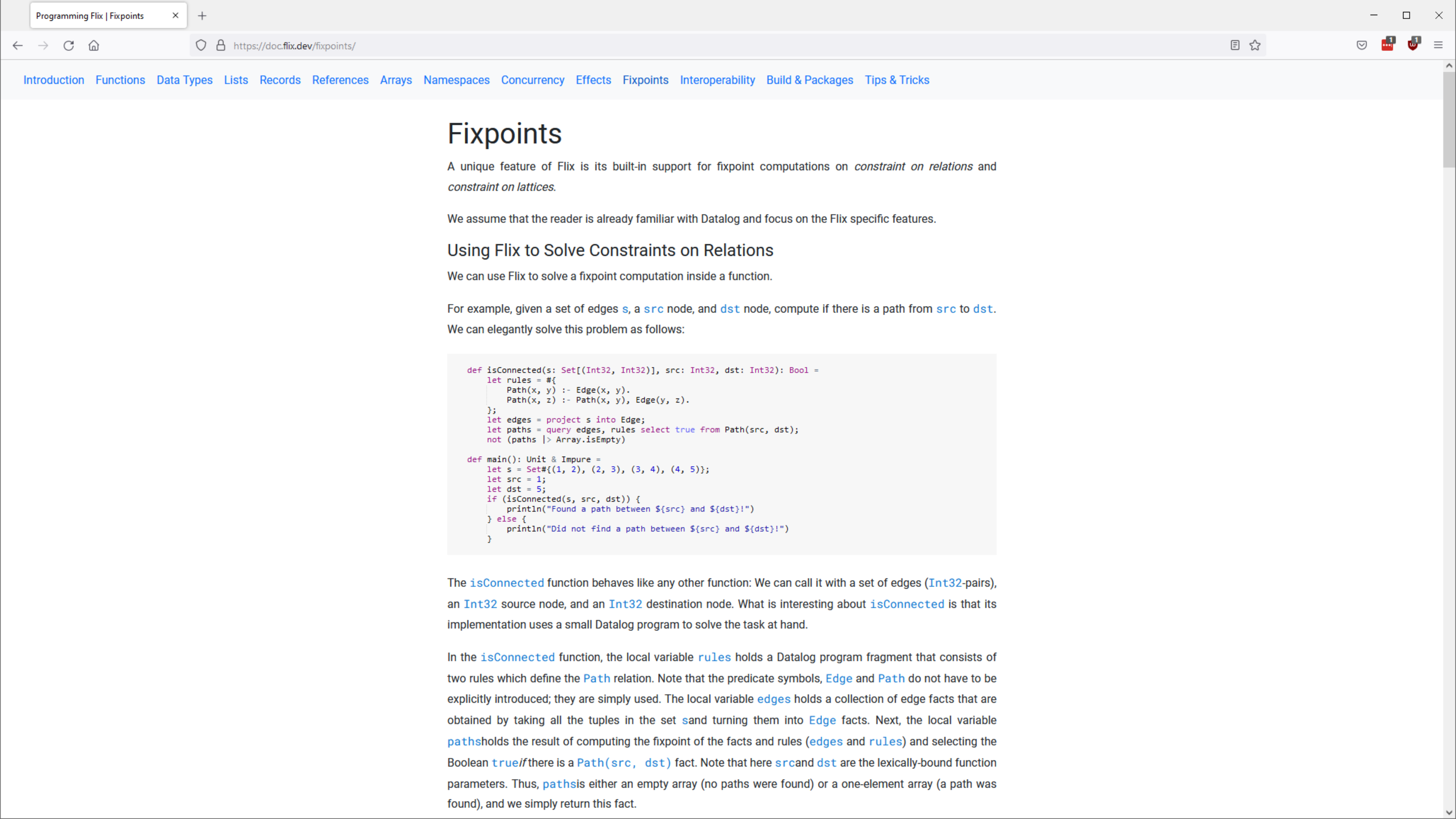
Tuples and Records

Flix has built-in support for tuples and records.

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

def oneByOne():  {w :: Int32, h :: Int32} = {w = 1, h = 1}

def twoByFour(): {w :: Int32, h :: Int32} = {w = 2, h = 4}
```



```
// A Suit type deriving an Eq and ToString instance
enum Suit with Eq, ToString {
    case Clubs
    case Hearts
    case Spades
    case Diamonds
}

// A Rank type deriving an Eq and Order instance
enum Rank with Eq, Order {
    case Number(Int32)
    case Jack
    case Queen
    case King
    case Ace
}

// A Card type deriving an Eq instance
opaque type Card with Eq = (Rank, Suit)

// An instance of ToString for Ranks
instance ToString[Rank] {
    pub def toString(x: Rank): String = match x {
        case Number(n) => "${n}"
        case Jack      => "Jack"
        case Queen     => "Queen"
        case King      => "King"
        case Ace       => "Ace"
    }
}

// An instance of ToString for Cards
instance ToString[Card] {
    pub def toString(x: Card): String = match x {
        case Card(r, s) => "${r} of ${s}"
    }
}

// Simulates a game of War, printing each player's turn.
def playWar(p1: List[Card], p2: List[Card], spoils: List[Card]): Unit & Impure = match (p1, p2) {
    case (Nil, Nil) => println("No one has any cards. It's a draw.")
    case (Nil, _) => println("Player 1 is out of cards. Player 2 wins!")
    case (_, Nil) => println("Player 2 is out of cards. Player 1 wins!")
    case (c1 :: d1, c2 :: d2) =>
        let Card(r1, _) = c1;
        let Card(r2, _) = c2;
        println("Player 1 plays ${c1}. Player 2 plays ${c2}.");
        if (r1 > r2) {
            println("Player 1 wins the battle.");
            // Add the spoils and losing card to the winner's deck.
            playWar(d1 ::: c1 :: c2 :: spoils, d2, Nil)
        } else if (r2 > r1) {
            println("Player 2 wins the battle.");
            // Add the spoils and losing card to the winner's deck.
            playWar(d1, d2 ::: c1 :: c2 :: spoils, Nil)
        } else {
            println("The battle is a draw. Time for war!");
            // Each player contributes their top 3 cards to the spoils
```



Programming Flix | Fixpoints

https://doc.flix.dev/fixpoints/

Fixpoints

A unique feature of Flix is its built-in support for fixpoint computations on *constraint on relations* and *constraint on lattices*.

We assume that the reader is already familiar with Datalog and focus on the Flix specific features.

Using Flix to Solve Constraints on Relations

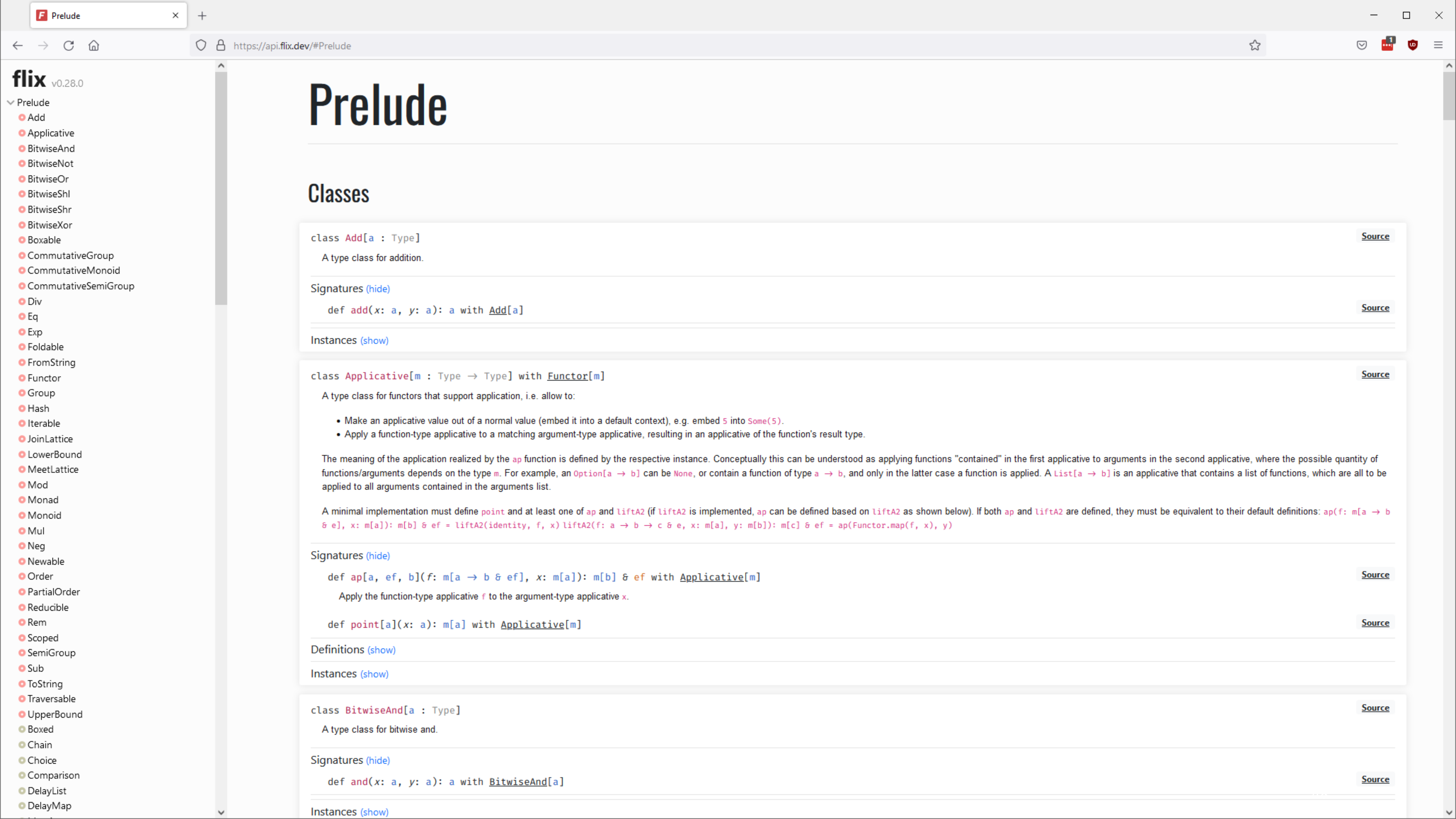
We can use Flix to solve a fixpoint computation inside a function.

For example, given a set of edges `s`, a `src` node, and `dst` node, compute if there is a path from `src` to `dst`. We can elegantly solve this problem as follows:

```
def isConnected(s: Set[(Int32, Int32)], src: Int32, dst: Int32): Bool =
    let rules = #{
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
    };
    let edges = project s into Edge;
    let paths = query edges, rules select true from Path(src, dst);
    not (paths |> Array.isEmpty)

def main(): Unit & Impure =
    let s = Set#{(1, 2), (2, 3), (3, 4), (4, 5)};
    let src = 1;
    let dst = 5;
    if (isConnected(s, src, dst)) {
        println("Found a path between ${src} and ${dst}!")
    } else {
        println("Did not find a path between ${src} and ${dst}!")
    }
```

The `isConnected` function behaves like any other function: We can call it with a set of edges (`Int32`-pairs), an `Int32` source node, and an `Int32` destination node. What is interesting about `isConnected` is that its implementation uses a small Datalog program to solve the task at hand.

In the `isConnected` function, the local variable `rules` holds a Datalog program fragment that consists of two rules which define the `Path` relation. Note that the predicate symbols, `Edge` and `Path` do not have to be explicitly introduced; they are simply used. The local variable `edges` holds a collection of edge facts that are obtained by taking all the tuples in the set `s`and turning them into `Edge` facts. Next, the local variable `paths`holds the result of computing the fixpoint of the facts and rules (`edges` and `rules`) and selecting the Boolean `true`*if* there is a `Path(src, dst)` fact. Note that here `src`and `dst` are the lexically-bound function parameters. Thus, `paths`is either an empty array (no paths were found) or a one-element array (a path was found), and we simply return this fact.



Prelude

https://api.flix.dev/#Prelude

flix v0.28.0

- Prelude
  - Add
  - Applicative
  - BitwiseAnd
  - BitwiseNot
  - BitwiseOr
  - BitwiseShl
  - BitwiseShr
  - BitwiseXor
  - Boxable
  - CommutativeGroup
  - CommutativeMonoid
  - CommutativeSemiGroup
  - Div
  - Eq
  - Exp
  - Foldable
  - FromString
  - Functor
  - Group
  - Hash
  - Iterable
  - JoinLattice
  - LowerBound
  - MeetLattice
  - Mod
  - Monad
  - Monoid
  - Mul
  - Neg
  - Newable
  - Order
  - PartialOrder
  - Reducible
  - Rem
  - Scoped
  - SemiGroup
  - Sub
  - ToString
  - Traversable
  - UpperBound
  - Boxed
  - Chain
  - Choice
  - Comparison
  - DelayList
  - DelayMap

Prelude

Classes

`class Add[a : Type]` Source

A type class for addition.

Signatures (hide)

`def add(x: a, y: a): a with Add[a]` Source

Instances (show)

`class Applicative[m : Type → Type] with Functor[m]` Source

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 `liftA2` (if `liftA2` is implemented, `ap` can be defined based on `liftA2` as shown below). If both `ap` and `liftA2` are defined, they must be equivalent to their default definitions: `ap(f: m[a → b & e], x: m[a]): m[b] & ef = liftA2(identity, f, x) liftA2(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]` Source

Apply the function-type applicative `f` to the argument-type applicative `x`.

`def point[a](x: a): m[a] with Applicative[m]` Source

Definitions (show)

Instances (show)

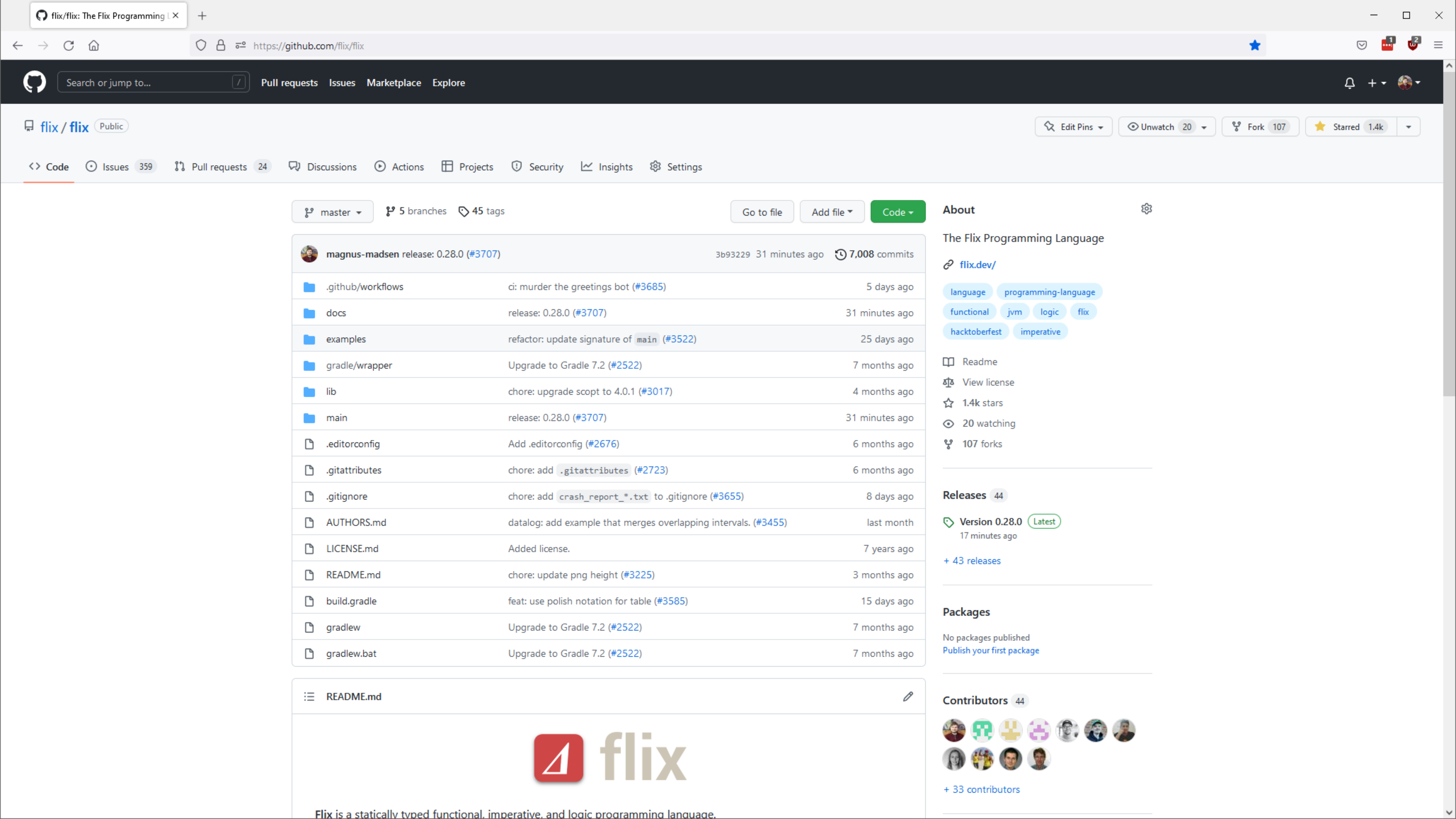
`class BitwiseAnd[a : Type]` Source

A type class for bitwise and.

Signatures (hide)

`def and(x: a, y: a): a with BitwiseAnd[a]` Source

Instances (show)


flix/flix: The Flix Programming L
https://github.com/flix/flix
Search or jump to...
Pull requests
Issues
Marketplace
Explore

flix / flix Public

5 branches 45 tags

magnus-madsen release: 0.28.0 (#3707) 3b93229 31 minutes ago 7,008 commits

.github/workflows	ci: murder the greetings bot (#3685)	5 days ago
docs	release: 0.28.0 (#3707)	31 minutes ago
examples	refactor: update signature of `main` (#3522)	25 days ago
gradle/wrapper	Upgrade to Gradle 7.2 (#2522)	7 months ago
lib	chore: upgrade scopt to 4.0.1 (#3017)	4 months ago
main	release: 0.28.0 (#3707)	31 minutes ago
.editorconfig	Add .editorconfig (#2676)	6 months ago
.gitattributes	chore: add `.gitattributes` (#2723)	6 months ago
.gitignore	chore: add `crash_report_*.txt` to .gitignore (#3655)	8 days ago
AUTHORS.md	datalog: add example that merges overlapping intervals. (#3455)	last month
LICENSE.md	Added license.	7 years ago
README.md	chore: update png height (#3225)	3 months ago
build.gradle	feat: use polish notation for table (#3585)	15 days ago
gradlew	Upgrade to Gradle 7.2 (#2522)	7 months ago
gradlew.bat	Upgrade to Gradle 7.2 (#2522)	7 months ago

README.md

Flix is a statically typed functional, imperative, and logic programming language.

About

The Flix Programming Language

flix.dev/

Readme

View license

1.4k stars

20 watching

107 forks

Releases 44

Version 0.28.0 Latest

17 minutes ago

+ 43 releases

Packages

No packages published

Publish your first package

Contributors 44

+ 33 contributors

7

Wrapping Up

Selection of Research Papers

From Datalog to Flix: A Declarative Language for Fixed Points on Lattices [**PLDI '16**]

Magnus Madsen, Ming-Ho Yee, Ondřej Lhoták

Fixpoints for the Masses: Programming with First-Class Datalog Constraints [**OOPSLA '20**]

Magnus Madsen, Ondřej Lhoták

Polymorphic Types and Effects with Boolean Unification [**OOPSLA '20**]

Magnus Madsen, Jaco van de Pol

Safe and Sound Program Analysis with Flix [**ISSTA '18**]

Magnus Madsen, Ondřej Lhoták

Implicit Parameters for Logic Programming [**PPDP '18**]

Magnus Madsen, Ondřej Lhoták

Relational Nullable Types with Boolean Unification [**OOPSLA '21**]

Magnus Madsen, Jaco van de Pol

You can find all papers at:
https://flix.dev/research/

Additional Resources

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

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
- local region-based mutation

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/