Functional Programming beyond map/filter/reduce

JFokus 2018

Prof. Dierk König

canoo

mittie

Monads
are just
monoids in the category
of endofunctors.

Explanation Path

Monoid *it's simple*

Functor *you already know it*

Endo- *cool idea*

Monoid of endofunctors = monad

Integers (+)

```
2 + 3          == 5
2 + 0          == 2
0 + 3          == 3
( 1+2 )+3 == 1+( 2+3 )
```

Integers (*)

```
2 * 3       == 6
2 * 1       == 2
1 * 3       == 3
( 2*3 )*4 == 2*( 3*4 )
```

Boolean (&&)

```
a && b                  == c
a && true               == a
true && b               == b
( a && b ) && c == a &&( b && c )
```

Lists, Arrays (+)

```
[1] + [2]         == [1, 2]
[1] + [ ]         == [1]
[ ] + [2]         == [2]
([1]+[2])+[3] == [1]+([2]+[3])
```

Strings (+)

```
"1" + "2"     == "12"
"1" + ""      == "1"
""  + "2"     == "2"
(  "Hi"+","  )+"JFokus" ==
                "Hi"+(  ","+"JFokus"  )
```

Extract

type <> type => type

neutral element

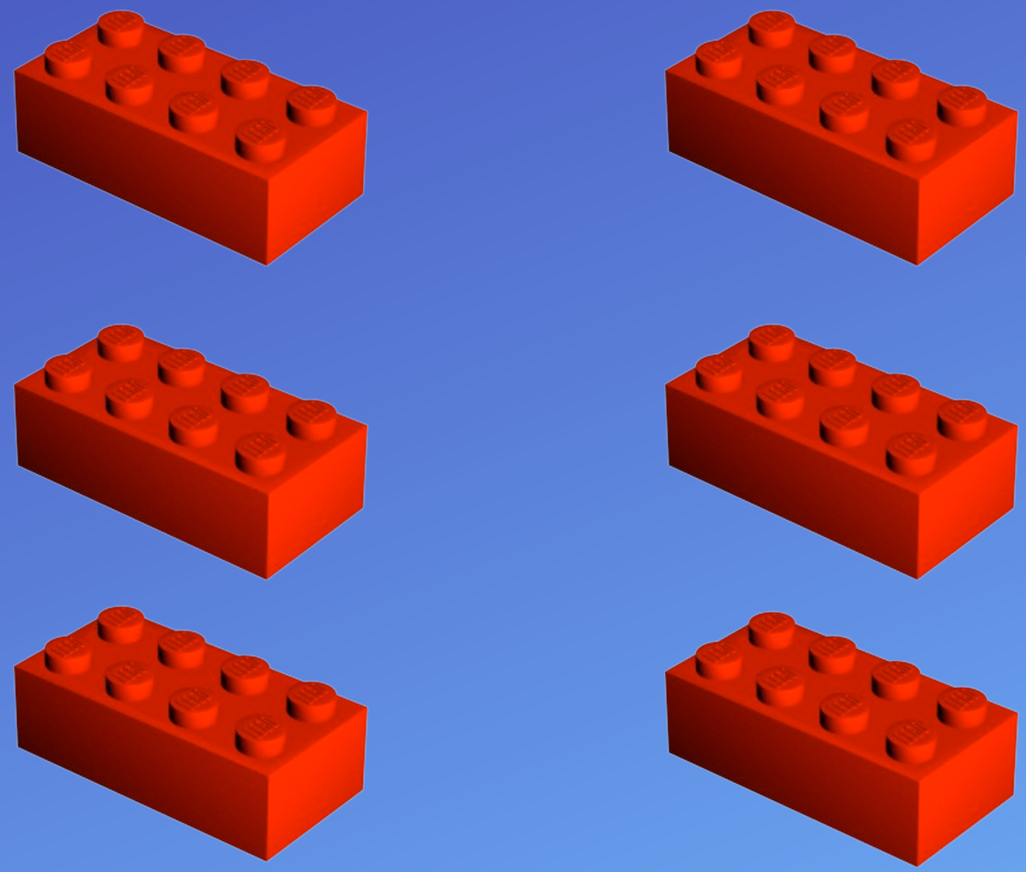
left & right identity

associative

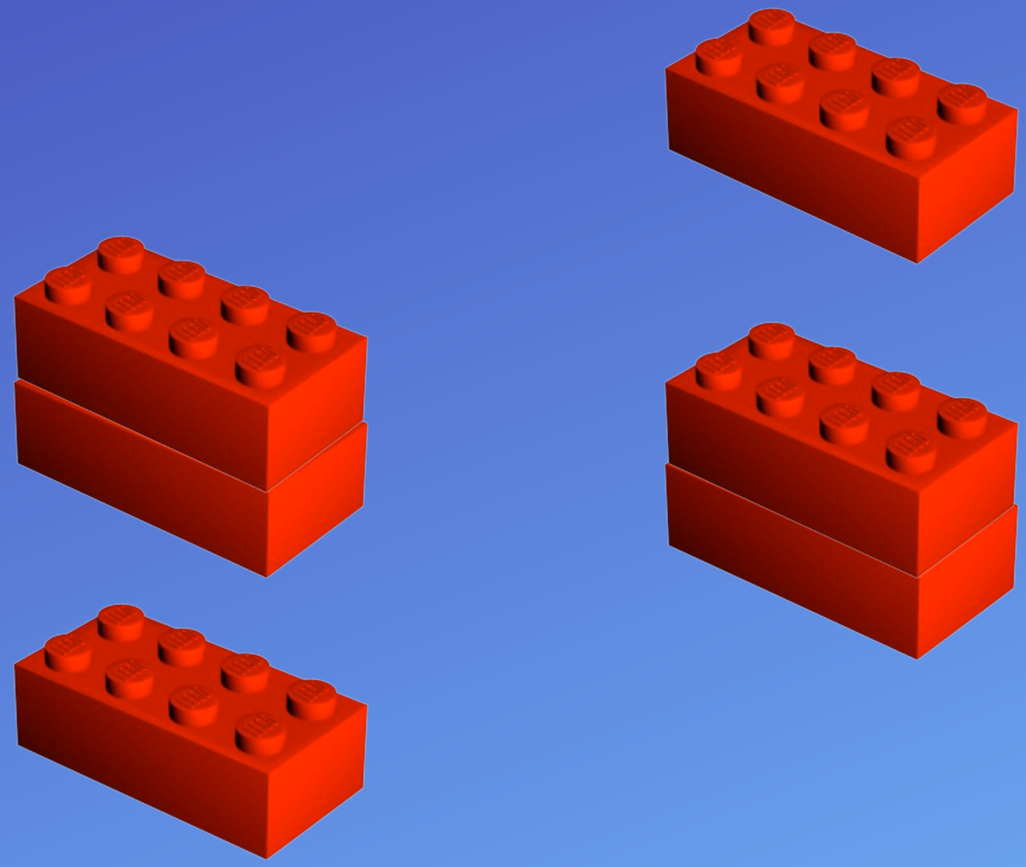
Abstract

We call this concept "Monoid"
short for "*that thing, you know, that you can combine*
with another thing and the combination must yield the same
result no matter which two you combine first."

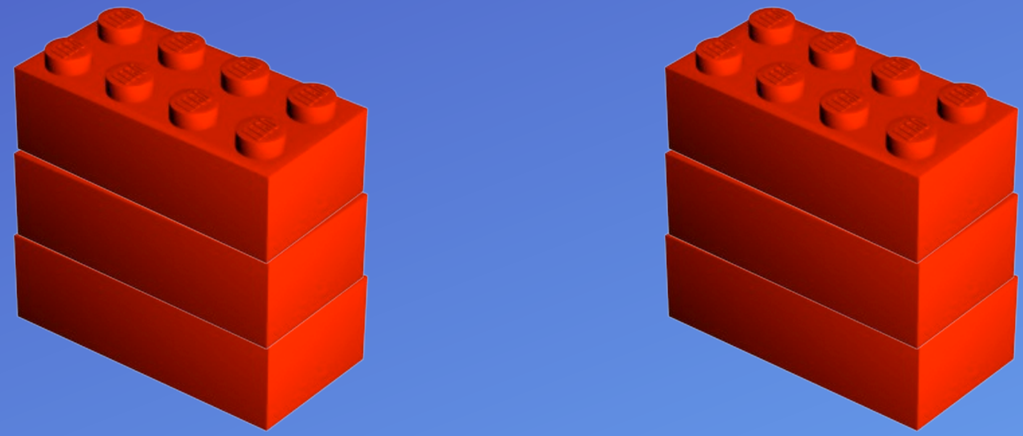
LEGO Monoid



LEGO Monoid

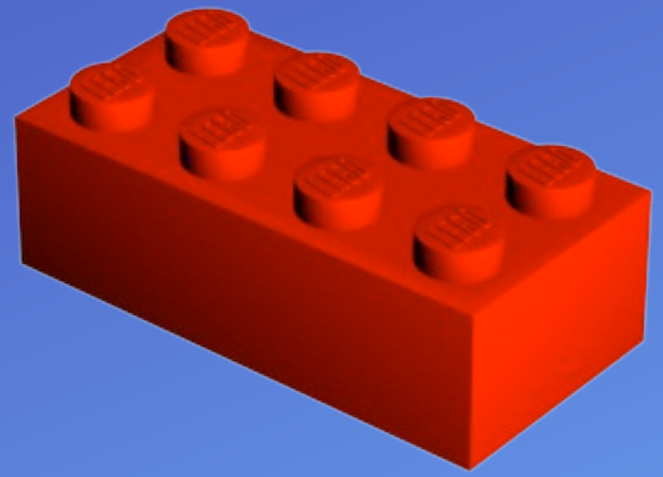


LEGO Monoid

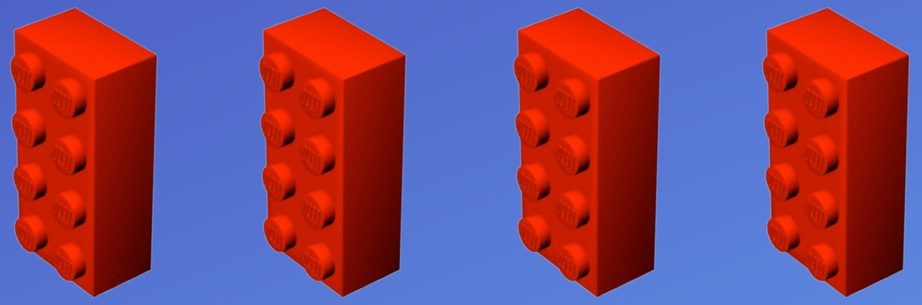


LEGO Neutral Brick

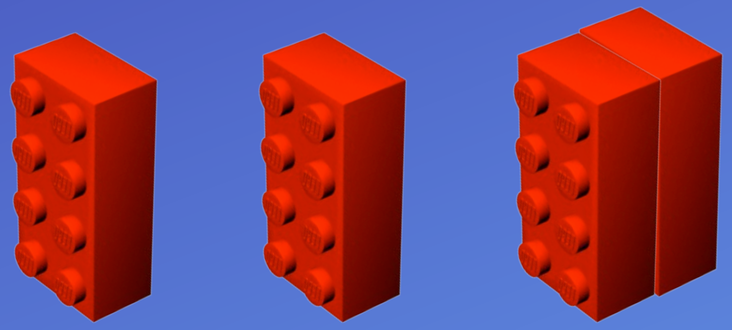
LEGO Left & Right Identity



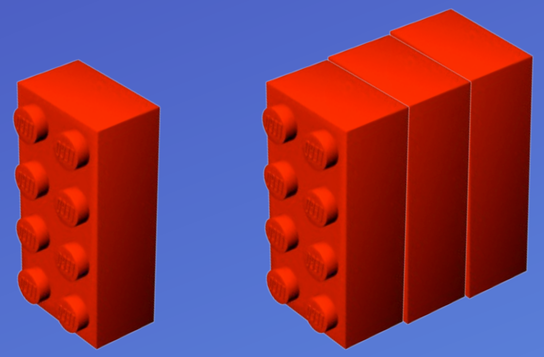
Generic Fold Right



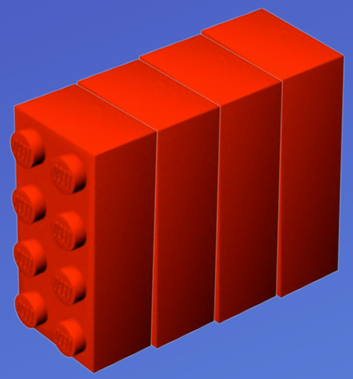
Generic Fold Right



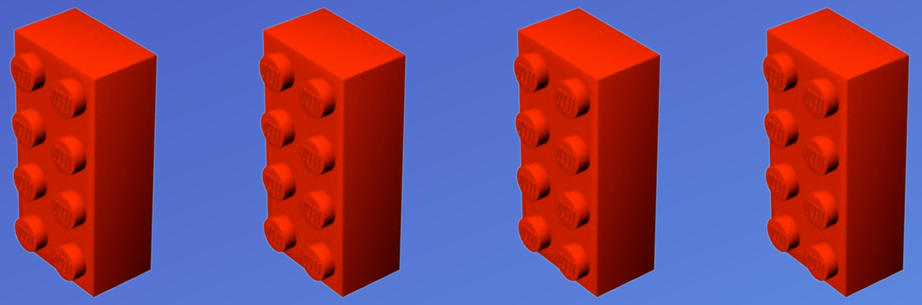
Generic Fold Right



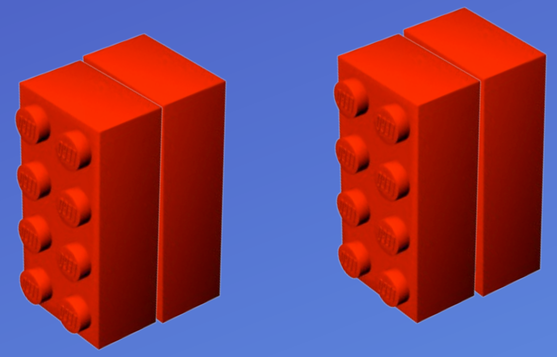
Generic Fold Right



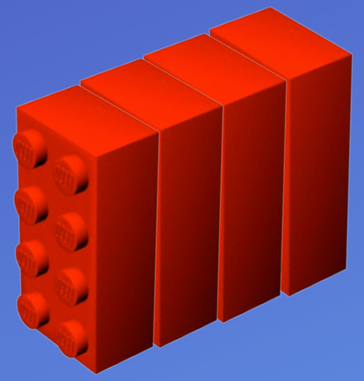
Generic Optimized Fold



Generic Optimized Fold



Generic Optimized Fold



Crazy Monoids

```
(a -> b) <> (b -> c) == (a -> c)
```

Can a function type `(a->b)` be a monoid?

What is the operation?

What is the neutral element?

Is it associative?

Function Composition

```
co(f,g) = x => f(g(x));

id(x) = x;

co(id,g) == x => id(g(x)) // definition co
         == x => g(x)     // apply id
         == g             // qed
```

Functors map

[1, 2, 3].map(x=> x+2) == [3, 4, 5]

(Just 1).map(x=> x+2) == Just 3

Nothing.map(whatever) == Nothing

List, Tree, Stream, Optional, Pipeline, Error, Validation, Observable, Future, Promise,..

Functors compose

```
[1, 2, 3].map( x => x + 2 )
          .map( x => 3 * x )
          ==
[1, 2, 3].map( x => 3 * (x+2) )
```

```
co( functor.map(f), functor.map(g) )
          == functor.map( co(f, g) )
```

```
functor.map(id) == id  // only for completeness
```

Functors are not Monoids

```
[1, 2, 3]       .map( x => x.toString() ) == ["1","2","3"]

[Int]           .map( Int -> String )       -> [String]

functor a       .map( a -> b)               -> functor b
```



Clever Idea:

instead of (a->b)
provide a special mapping with (a -> functor b),
which is essentially a constructor for functors.

```
functor a   <>   functor b                 =>   functor  b
functor a   .endo(a->functor b)   =>   functor  b
```

Clever Idea in Action

```
[1, 2, 3].endo( x => replicate(x, x.toString()) ) ==
        [ ["1"], ["2","2"], ["3","3","3"] ]

then flatten => ["1", "2", "2", "3", "3", "3"]  //aka "join"

funcA.flatMap(f) = funcB.flatten(funcA.endo(f))

// aka "bind" or ">>="
```

Finalising the Monoid

functor a .flatMap(a->functor b) => functor b

(a->functor a) <> (a->functor b) => (a->functor b)

We need an (a-> functor a) ctor and flatMap (we already have map, so we only need flatten). If the functor is monoidal with flatMap as <> and ctor as neutal element, then we call it a Monad.

Wrapping up

Let (m a) be a given monad over type a.
(m a) has a ctor (a -> m a). // aka "return" or "pure"
(m a) is a functor over a.
(m (m a)) can be flattened to (m a).
(a-> m a).flatMap(a -> m b) is a monoidal operation.

Alternative way of writing in pure FP style
(>>=) :: Monad m => m a -> (a -> m b) -> m b

Takeaways

Associativity (monoid) requires pure functions.

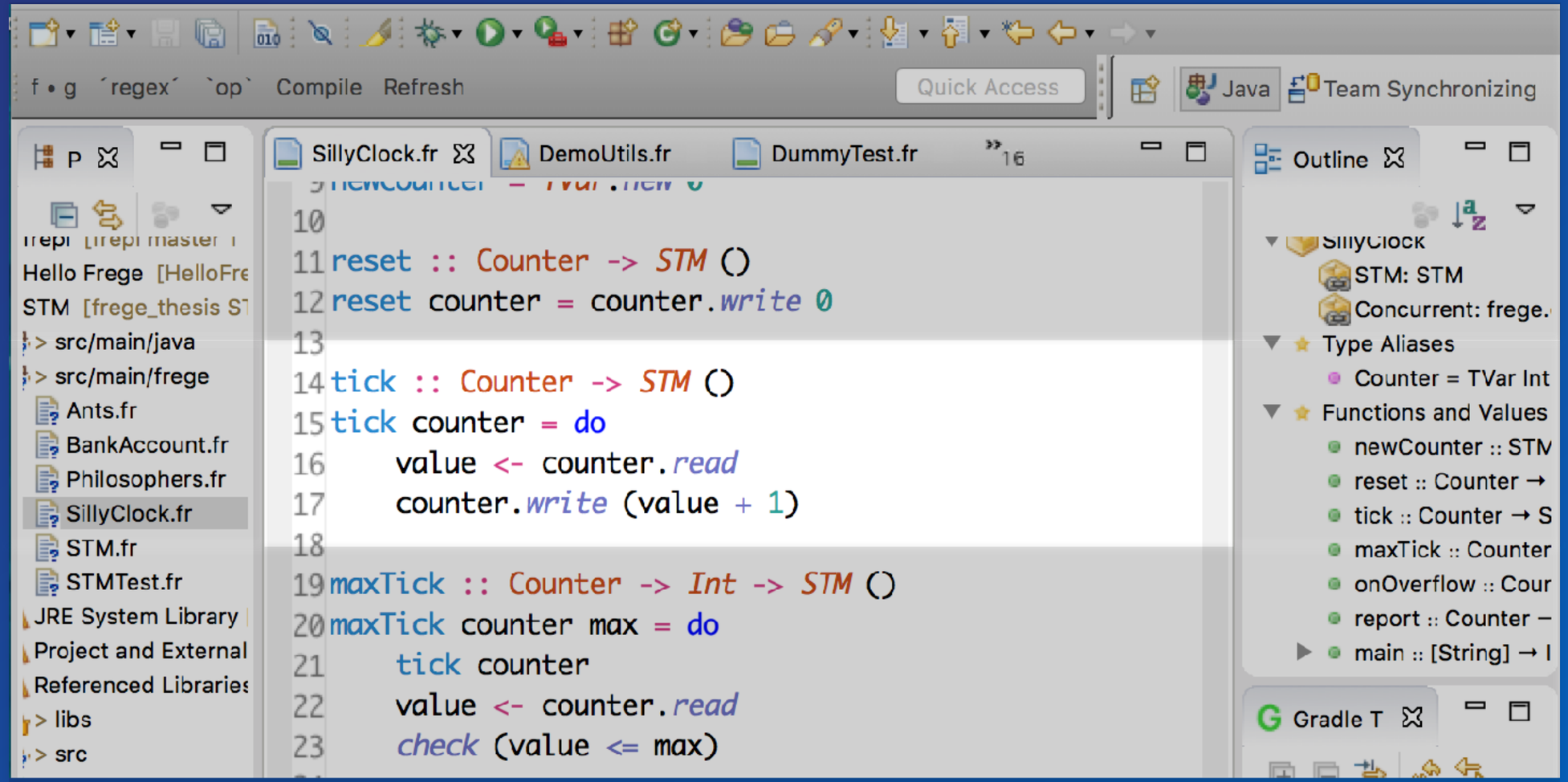
Monoids can fold but they cannot escape.

Monads are the most versatile functors (map, filter, expand, reduce) that composes and folds without escaping.

Use Cases

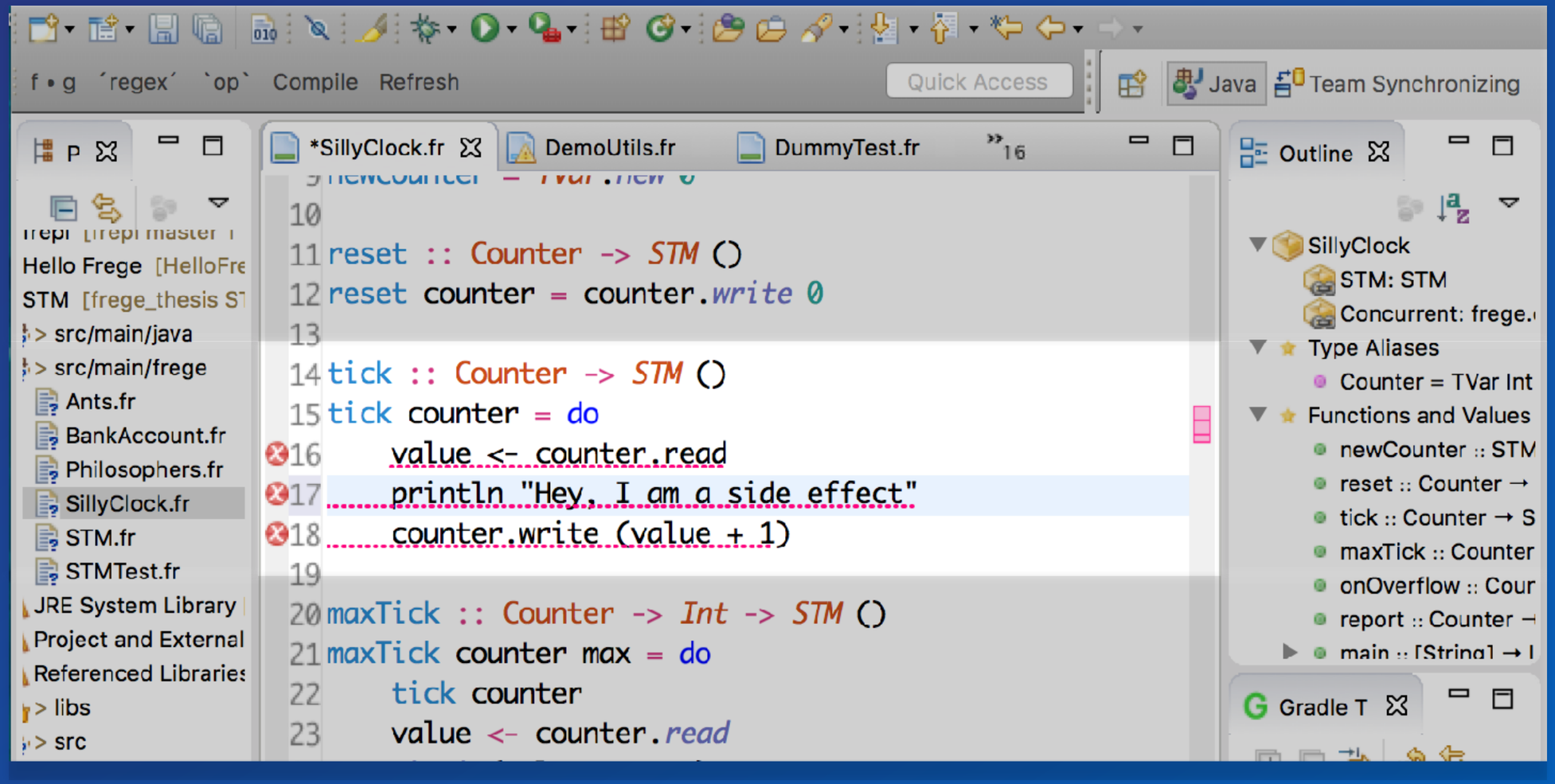
Purely functional state threads
List comprehensions, Streams (possibly reactive)
CompletableFuture, Promises, Continuations
LINQ-style database access
Either, Validation, Exception handling
Optionality, Indeterminism, Parsers, Search trees
Sequencing IO actions, isolating UI actions
STM transactions, …

No IO in Transactions!



Type inference FTW

Less tricky errors



There is a world…

… where logical reasoning rules and structure arises from consistency and a rich set of relations.

Exploring this world is like programming without implementation.

Purely functional programming opens the door. Consider Haskell, Frege, Purescript, Idris.

Please give feedback!

Prof. Dierk König

canoo

mittie