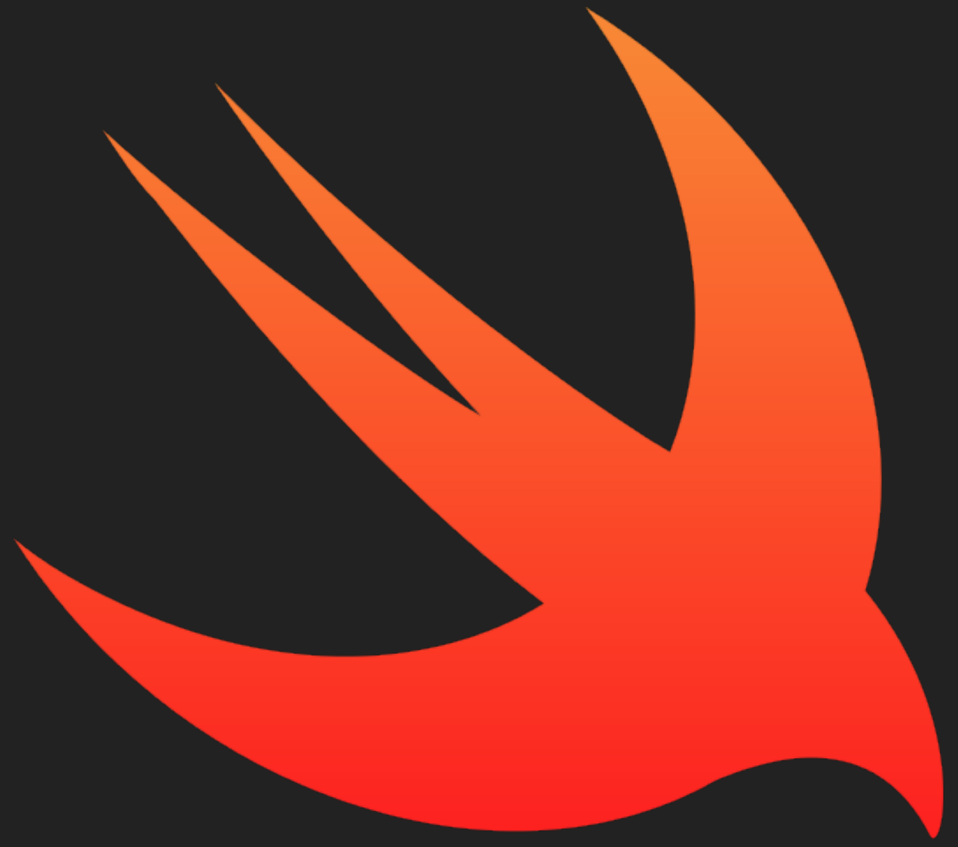
ROB NAPIER

λ: THERE AND BACK AGAIN



≠ λ

```haskell
slice :: [a] -> Int -> Int -> [a]
slice [] _ _ = []
slice xs i j = map snd . filter((>=i) . fst) $ zip [1..j] xs
```

λ ≠

```
>>=
<*>
<$>
%~
<+=
```

FUNCTIONAL PROGRAMMING IS A WAY OF THINKING

```
var persons: [Person] = []
for name in names {
    let person = Person(name: name)
    if person.isValid {
        persons.append(person)
    }
}
```

```swift
var persons: [Person] = []
for name in names {
    let person = Person(name: name)
    if person.isValid {
        persons.append(person)
    }
}
```

```
var persons: [Person] = []
for name in names {
    let person = Person(name: name)
    if person.isValid {
        persons.append(person)
    }
}
```

```swift
var persons: [Person] = []
for name in names {
    let person = Person(name: name)
    if person.isValid {
        persons.append(person)
    }
}
```

```
var persons: [Person] = []
for name in names {
    let person = Person(name: name)
    if person.isValid {
        persons.append(person)
    }
}
```

```
var possiblePersons: [Person] = []
for name in names {
    let person = Person(name: name)
    possiblePersons.append(person)
}


var persons: [Person] = []
for person in possiblePersons {
    if person.isValid {
        persons.append(person)
    }
}
```

```
var possiblePersons: [Person] = []
for name in names {
    let person = Person(name: name)
    possiblePersons.append(person)
}
```

↓

```
let possiblePersons = names.map(Person.init)
```

```swift
var persons: [Person] = []
for person in possiblePersons {
    if person.isValid {
        persons.append(person)
    }
}
```

```swift
let persons = possiblePersons.filter { $0.isValid }
```

```swift
var persons: [Person] = []
for name in names {
    let person = Person(name: name)
    if person.isValid {
        persons.append(person)
    }
}
```

↓

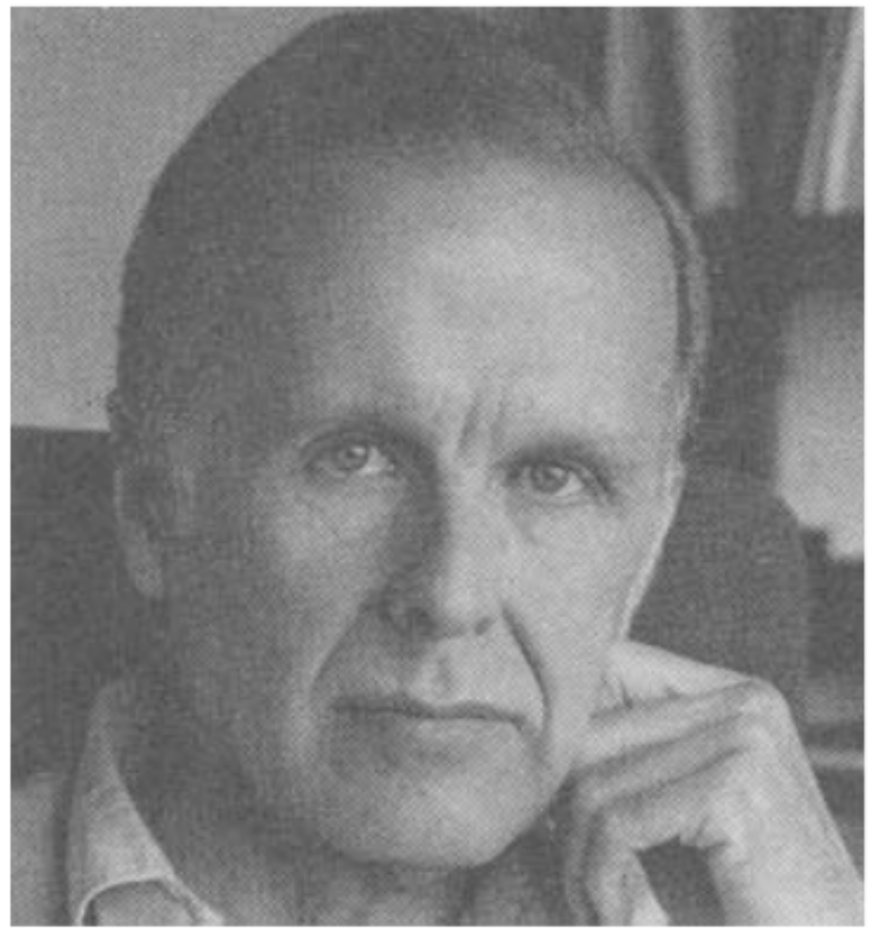
```swift
let possiblePersons = names.map(Person.init)
let persons = possiblePersons.filter { $0.isValid }
let persons = names
    .map(Person.init)
    .filter { $0.isValid }
```

“FUNCTIONAL” TOOLS

- dropFirst
- dropLast
- forEach
- flatMap
- prefix
- split
- suffix
- first(where:)
- contains
- elementsEqual
- enumerated
- flatten
- joined
- max
- min
- reduce
- reversed
- sorted
- starts(with:)
- isEmpty
- count
- index(of:)
- index(where:)
- popFirst
- removeFirst
- …

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Author's address: 91 Saint Germain Ave., San Francisco, CA 94114.

Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

An alternative functional style of programming is founded on the use of combining forms for creating programs. Functional programs deal with structured data, are often nonrepetitive and nonrecursive, are hierarchically constructed, do not name their arguments, and do not require the complex machinery of procedure declarations to become generally applicable. Combining forms can use high level programs to build still higher level ones in a style not possible in conventional languages.

HASKELL

Create new function sum by combining existing functions foldr and +

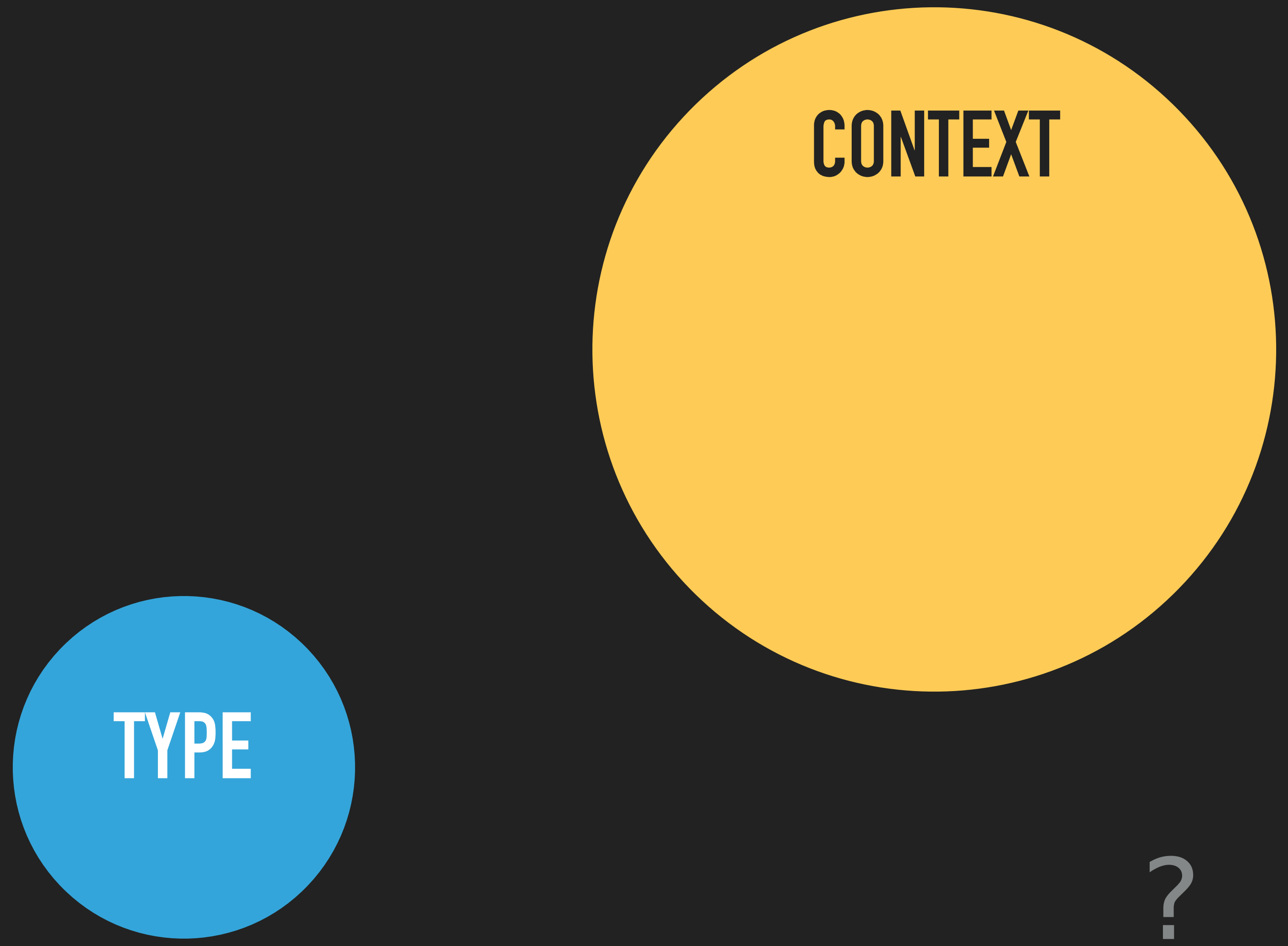
```
let sum = foldr (+) 0
sum [1..10]
```

SWIFT

Attach Sequence methods to MyStruct

```
extension MyStruct<T>: Sequence {
    func makeIterator() -> AnyIterator<T> {
        return ...
    }
}
```

LIFTING A TYPE



CONTEXT

TYPE

?

String

NO VALUE: MAGIC VALUE

```
let noValue = −1

let n = 0

if n != noValue { … }
```

NO VALUE: CONTEXT

```
let n: Int? = 0

if let n = n { … }
```

```
func login(username: String, password: String,
           completion: (String?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

```
func login(username: String, password: String,
           completion: (String?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

```swift
func login(username: String, password: String,
           completion: (_ token: String?, Error?) -> Void)
```

```swift
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

```
struct Token {
    let string: String
}
```

```
func login(username: String, password: String,
           completion: (Token?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

```
func login(username: String, password: String,
           completion: (Token?, Error?) -> Void)
```

```
login(username: "rob", password: "s3cret") {
    (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

“AND” TYPE
(PRODUCT)

```swift
struct Credential {
    var username: String
    var password: String
}
```

```
func login(credential: Credential,
           completion: (Token?, Error?) -> Void)
```

```
let credential = Credential(username: "rob",
                            password: "s3cret")
login(credential: credential) { (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

```
func login(credential: Credential,
           completion: (Token?, Error?) -> Void)
```

```
let credential = Credential(username: "rob",
                            password: "s3cret")
login(credential: credential) { (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

```
let credential = Credential(username: "rob",
                            password: "s3cret")
login(credential: credential) { (token, error) in
    if let token = token {
        // success
    } else if let error = error {
        // failure
    }
}
```

		token	
		set	nil
error	set	??	✅
	nil	✅	??

“OR” TYPE

(SUM)

```
enum Result<Value> {
    case success(Value)
    case failure(Error)
}
```

```
func login(credential: Credential,
           completion: (Result<Token>) -> Void)
```

```
login(credential: credential) { result in
    switch result {
    case .success(let token): // success
    case .failure(let error): // failure
    }
}
```

λ →

THE LESSONS

- Break apart complicated things into simpler things
- Look for generic patterns in the simple things
- Lift and compose simple things to make complex ones

LAMBDA: THERE AND BACK AGAIN

BREAK IT DOWN. BUILD IT UP.