Introduction to Functional Programming and Haskell

Aden Seaman

Functional Programming

Functional Programming

First Class Functions

Expressions (No Assignment)

(Ideally) No Side Effects

Different Approach

Brief History

1930	Lambda Calculus
1958	LISP
1974	ML
1974	Scheme
1986	Erlang
1990	Haskell
1996	OCaml
2004	Scala
2005	F#
2007	Clojure

Who?

Academia

Language Researchers

Growing Interest

Why?

Safety

Modularity

Concurrency

Concise

Fun

Better ⋆

Structures

Lists

[ ]
[1, 2, 3] = 1:2:3:[ ]
['a', 'b', 'c'] = 'a':'b':'c':[ ]

Concatenation
Indexing
Head, Tail
head [1, 2, 3] = 1
tail [1, 2, 3] = [2, 3]

Structures

Tuples

(1, 2, “hello”)
(“world”, [3, 4, 5])

Bundling Data

Techniques

(Only) Recursion

```
fib 1 = 1
fib 2 = 1
fib n = fib (n-1) + fib (n-2)
```

Techniques

Map

map f [a, b, c, d]

↓

[f a, f b, f c, f d]

Techniques

Filter

filter (< 5) [1, 5, 9, 3, 4]

↓

[1, 3, 4]

Techniques

Fold

```
foldl (+) 0 [ 1, 5, 9, 3, 4 ]
```

↓

22

Language Features

First Class Functions

```
f1 x = 2 * x
f2 x = 4 * x
f3 v f = ( f v ) + 3*( f v )
```

```
f3 5 f1 = ( 2*5 ) + 3*( 2*5 ) = 40
f3 5 f2 = ( 4*5 ) + 3*( 4*5 ) = 80
```

Language Features

Pattern Matching

Variable Assignment

a:b = [1, 2, 3, 4, 5]

↓

a = 1

b = [2, 3, 4, 5]

Language Features

Pattern Matching
Variable Assignment

(a, b, c) = (1, “hi”, [2, 3, 4])

↓

a = 1

b = “hi”

c = [2, 3, 4]

Language Features

Pattern Matching

Function Definitions

```
f 0 = 1
f 2 = 5
f x = x + 9
```

Language Features

Pattern Matching

Combinations

```
f (a,b) = a + b
f (1,2) = 3

map f [] = []
map f (h:t) = (f h):(map f t)
```

Language Features

Lambda Functions

```
f (a,b) = a + b
f (2,3) = 5
(\(a,b) -> a + b) (2,3) = 5

map (\(a,b) -> a + b) [ (1,2), (3, 4), (5,6) ]
                ↓
          [ 3, 7, 11 ]
```

Language Features

Currying

```
f a b = a + b

g = f 2 = (2 +)

g 3 = (f 2) 3 = (2 +) 3 = 5
```

Language Features

Composition

```
(f . g) x = f (g x)

f x = 2*x
g x y = 3*x + y
h = f . (g 3)
h 4 = 2*(g 3 4) = 2*(3*3 + 4) = 26
```

Language Features

Combined Example

Lists
Tuples
Map
Filter
Fold

First Class Functions
Pattern Matching
Lambda Functions
Currying
Composition
Recursion

Sum numbers under specific headers

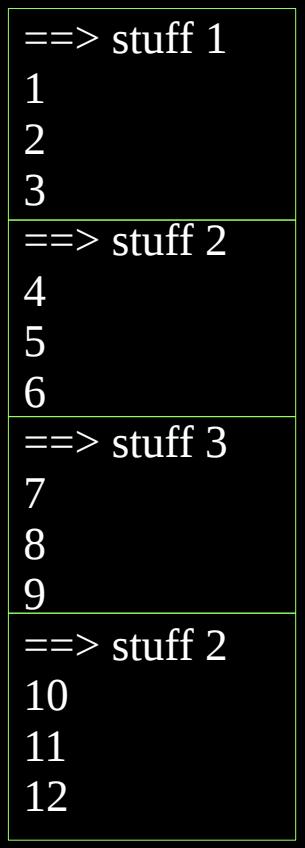
==> stuff 1
1
2
3
==> stuff 2
4
5
6
==> stuff 3
7
8
9
==> stuff 2
10
11
12

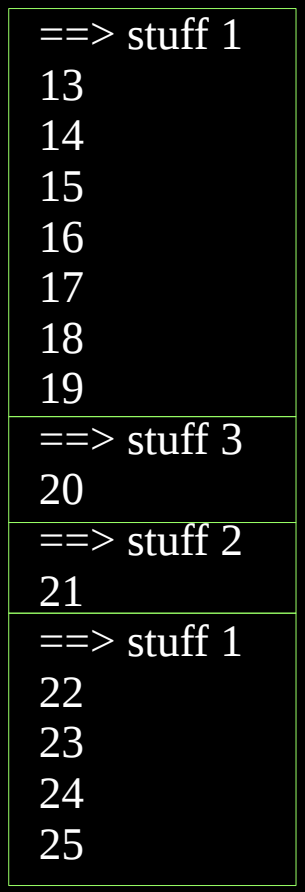
==> stuff 1
13
14
15
16
17
18
19
==> stuff 3
20
==> stuff 2
21
==> stuff 1
22
23
24
25

```
==> stuff 1
1
2
3
==> stuff 2
4
5
6
==> stuff 3
7
8
9
==> stuff 2
10
11
12
```

```
==> stuff 1
13
14
15
16
17
18
19
==> stuff 3
20
==> stuff 2
21
==> stuff 1
22
23
24
25
```

```
segmentByPrefix :: String -> [String] -> [[String]]
segmentByPrefix prefix lines = tail $ helper [] [] lines
  where helper totalAcc listAcc [] = reverse ((reverse listAcc):totalAcc)
        helper totalAcc listAcc (x:xs)
          | isPrefixOf prefix x = helper ((reverse listAcc):totalAcc) [x] xs
          | otherwise           = helper totalAcc (x:listAcc) xs
```






```
segmentByPrefix :: String -> [String] -> [[String]]
segmentByPrefix prefix lines = tail $ helper [] [] lines
  where helper totalAcc listAcc [] = reverse ((reverse listAcc):totalAcc)
        helper totalAcc listAcc (x:xs)
          | isPrefixOf prefix x = helper ((reverse listAcc):totalAcc) [x] xs
          | otherwise           = helper totalAcc (x:listAcc) xs
```

- ✓Lists
- Tuples
- Map
- Filter
- Fold
- First Class Functions
- ✓Pattern Matching
- Lambda Functions
- Currying
- Composition
- ✓Recursion

```
segmentByPrefix :: String -> [String] -> [[String]]
segmentByPrefix prefix lines = tail $ helper [] [] lines
  where helper totalAcc listAcc [] = reverse ((reverse listAcc):totalAcc)
        helper totalAcc listAcc (x:xs)
          | isPrefixOf prefix x = helper ((reverse listAcc):totalAcc) [x] xs
          | otherwise           = helper totalAcc (x:listAcc) xs
```

==> stuff 1
1
2
3
==> stuff 2
4
5
6
==> stuff 3
7
8
9
==> stuff 2
10
11
12

==> stuff 1
13
14
15
16
17
18
19
==> stuff 3
20
==> stuff 2
21
==> stuff 1
22
23
24
25

```
process1 :: String -> String -> String -> [Double]
process1 prefix header inputString = summedSegments
  where inputLines = lines inputString
        segments = segmentByPrefix prefix inputLines
        headeredSegments = map (\(h:t) -> (h,t)) segments
        filteredSegments = filter (\(h,_) -> h == header) headeredSegments
        numberSegments = map (\(h,t) -> map read t :: [Double]) filteredSegments
        summedSegments = map (foldl (+) 0) numberSegments
```

- ✓Lists
- ✓Tuples
- ✓Map
- ✓Filter
- ✓Fold
- ✓First Class Functions
- ✓Pattern Matching
- ✓Lambda Functions
- ✓Currying
- Composition
- ✓Recursion

```
process1 :: String -> String -> String -> [Double]
process1 prefix header inputString = summedSegments
  where inputLines = lines inputString
        segments = segmentByPrefix prefix inputLines
        headeredSegments = map (\(h:t) -> (h,t)) segments
        filteredSegments = filter (\(h,_) -> h == header) headeredSegments
        numberSegments = map (\(h,t) -> map read t :: [Double]) filteredSegments
        summedSegments = map (foldl (+) 0) numberSegments
```

- ✓Lists
- ✓Tuples
- ✓Map
- ✓Filter
- ✓Fold
- ✓First Class Functions
- ✓Pattern Matching
- ✓Lambda Functions
- ✓Currying
- ✓Composition
- ✓Recursion

```
process2 :: String -> String -> String -> [Double]
process2 prefix header = (map (foldl (+) 0)) .
         (map (\(h,t) -> map read t :: [Double])) .
         (filter (\(h,_) -> h == header)) .
         (map (\(h:t) -> (h,t))) .
         (segmentByPrefix prefix) .
         lines
```

```
import Data.List (isPrefixOf)

process2 :: String -> String -> String -> [Double]
process2 prefix header = (map (foldl (+) 0)) .
        (map (\(h,t) -> map read t :: [Double])) .
        (filter (\(h,_) -> h == header)) .
        (map (\(h:t) -> (h,t))) .
        (segmentByPrefix prefix) .
        lines

segmentByPrefix :: String -> [String] -> [[String]]
segmentByPrefix prefix lines = tail $ helper [] [] lines
  where helper totalAcc listAcc [] = reverse ((reverse listAcc):totalAcc)
        helper totalAcc listAcc (x:xs)
          | isPrefixOf prefix x = helper ((reverse listAcc):totalAcc) [x] xs
          | otherwise           = helper totalAcc (x:listAcc) xs

main :: IO ()
main = do
  inputFileContents <- readFile "input.file.txt"
  putStrLn $ show (process2 "==>" "==> stuff 1" inputFileContents)
```

[6.0, 112.0, 94.0]

Language Features

Others

Immutability

Garbage Collection

Closures

Closures

```
f x = (\y -> x + y)

makeClosure :: ContextData -> (ContextData -> InputData ->
OutputData) -> (InputData -> OutputData)

makeClosure contextData processFunction = \inputData ->
processFunction contextData inputData

makeClosure contextData processFunction = processFunction
contextData
```

Closures

```
f x = (\y -> x + y)

makeClosure :: ContextData -> (ContextData -> InputData ->
OutputData) -> (InputData -> OutputData)

makeClosure contextData processFunction = \inputData ->
processFunction contextData inputData

makeClosure contextData processFunction = processFunction
contextData
```

Uninterestingly common in Haskell!

Language Features

Absent

Loops

Multiple Assignment ⋆

Object Oriented ⋆

Global Variables

Contemporary Languages

Language	Pure	Typed	Lazy	OO
Haskell	✓	✓	✓	✗
OCAML	✗	✓	~	✓
Scala	✗	✓	~	✓
Clojure	✗	✗	✗	✗
Erlang	✗	✗	✗	✗

Haskell

Approach

Constrained

Only the Haskell way

Top-Down (Abstractions, Lambda Calc & Category Theory)

Safe

Fast

Powerful

Influences

Clean, FP, Gofer, Hope, Id, ISWIM, KRC, Lisp, Miranda, ML, Standard ML, Orwell, SASL, Scheme, SISAL

Influences

Clean, FP, Gofer, Hope, Id, ISWIM, KRC, Lisp, Miranda, ML, Standard ML, Orwell, SASL, Scheme, SISAL

Unification Language
Designed by committee

Language Features

Currying
Infix
Concise
Pattern Matching
Implicit Functions
Lazy Evaluation

Why?

Enforced Safety

Strongly Typed
Single Assignment
Pure/Contained IO
Smart Shared State
Expressions (everything is one)

Why?

Separation

Fast (reordering, fusion)
Parallelism (sparks)
Concurrency (Haskell threads)
Modularity & Reuse

Why?

Lazy Evaluation

Infinite Data Structures
Efficiency
Composition & Modularity
Hard to reason about

Why?

Others

Concise
Foreign Function Interface
Inline code (Python, Java, C, R)
Abstract Power
Advanced Features

Purity

Safe

Referential Transparency

Easy to reason about

Composable

Type System

Strict (hard to get used to)
Polymorphism
Type Inference
Mental Gymnastics
Algebraic Data Types
Type Classes

Type System

Algebraic Data Types

```
data Maybe a     = Just a | Nothing

data Either a b = Left a | Right b

data List a      = Item a (List a) | Null

data Center      = Center
{
    centerX :: Double
  , centerY :: Double
}

data Shape       = Square Center Double |
                   Rectangle Center Double Double |
                   Circle Center Double
```

Type System

Algebraic Data Types

```
data Maybe a     = Just a | Nothing

data Either a b = Left a | Right b

data List a      = Item a (List a) | Null

data Center      = Center
{
    centerX :: Double
  , centerY :: Double
}

data Shape       = Square Center Double |
                   Rectangle Center Double Double |
                   Circle Center Double
```

Can be emulated in C and others!!

Type System

Types say a lot

```
func1 :: String -> Int

func2 :: Num a => a -> a -> a

func3 :: Double

func4 :: IO Double

func5 :: FilePath -> IO String

func6 :: String -> IO ()
```

ADTs + Type Classes

Contexts

Monoids
Functors
Applicative Functors
Monads
etc...

Monads

Isolation

Containment

Context Sensitive

Imperative-Like Sugar

“Haskell in the finest imperative programming language” - SPJ

Monads

Definition

```
class Applicative m => Monad where
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  return :: a -> m a
  fail :: String -> m a


class Functor f => Applicative where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b


class Functor where
  fmap :: (a -> b) -> f a -> f b
```

Monads

Kinds

IO
STM
ST
List, Maybe, Either
State, Reader, Writer
etc...

Monads

Example (Maybe)

```
class Applicative m => Monad where
  (>>=)  :: m a -> (a -> m b) -> m b
  (>>)   :: m a -> m b -> m b
  return :: a -> m a
  fail   :: String -> m a


instance Monad Maybe where
    return x =   Just x
    Nothing  >>= f  = Nothing
    Just x   >>= f  = f x
    fail _   =   Nothing
    (>>)     =   (*>)
```

Monads

Example (Maybe)

```
checkString :: String -> Maybe Double
checkString input = do
   val1 <- readMaybe input :: Double
   val2 <- inBounds val1
   return $ 4.0 + 5.0*val2

inBounds :: Double -> Maybe Double
inBounds val
   | val < 10  = Nothing
   | otherwise = Just val


checkString “foo” = Nothing
checkString “5”   = Nothing
checkString “20”  = Just 104.0
```

Side Effects

Happen in IO monad

Can’t escape from IO monad

Push side-effects to “edges”

Pure functional core

More Advanced Stuff

Applicative Functors
Category Theory
Concurrency (MVAR, STM)
Parallelism
Laziness
Monad Transformers
Pipe/Conduit
Arrows
Lens
Parsers
Persistent
Free Monads
Crazy type extensions
Template Haskell (DSLs)
Metaprogramming
Way more ...

FP + Haskell = Cool

New Ideas

Better Programmer

Safer Programmer

Amazing Concurrency

Very Fast

Brain Bending

Horizon Expanding

Challenging & Fun!