A Primer on Functional Programming

Sarah Withee

@geekygirlsarah

#selfconf

Monads

(jk)

Introduction

- How many of you have heard of functional programming?
- How many of you have done functional programming?
- How many of you ARE functional programmers?

- How many wanted to learn but never had time/good resources?

Explaining Functional Programming

- It’s not new (some languages/principles from 1950s)
- Built on ideas of lambda calculus, designed in the 30s for mathematical principles

Explaining Functional Programming

- Based on idea of pure functions
- A function, given certain inputs, ALWAYS produces the same output
- Don’t have side effects
- Functions based on time, file access, database access, previous function calls, etc. are impure
- User input is never pure (duh)
- Call by reference is impure
- Nearly impossible to write 100% pure function programs

Explaining Functional Programming

Examples:

- sin(x) – always produces sine of value at x
- length(x) – always returns the same size of the string
- getAccountNumberFromDb(name) – kidding. Not pure because it relies on a database that may not may not produce the same result each time

Referential Transparency

- Any expression that can replace its value with no behavior changes
- Ex: $x = 3$

  $x + 5 = 8$

  $3 + 5 = 8$

  Both result in the same value with no behavior changes

(Note: Assignments in code are NOT transparent… more later)

Referential Transparency

- Referential opacity – the opposite
- In mathematics, all functions are transparent.
- In programming, this is not the case
- Pure functions always have referential transparency

Referential Transparency

- Assignments are NOT transparent

  ```
  x = x + 1

  def addOne(int x):
      return x + 1;
  ```

```
addOne(x) = addOne (y);
```

Lambda functions

- Anonymous functions (no name or identifier)
- Usually for higher level functions or to pass arguments to one
- Usually used once to a few times
- Can't be recursive*

* otherwise they need a name or some way of maintaining state**

** which is possible but outside of this scope

Lambda functions

```
f = lambda x:  x*x
print f(5)

def square(x):
    return lambda x: x*x
print square(5)
```

Lambda functions

- Functions ARE values
- Functions can be passed as values into functions

Lambda functions

```
def divide(x, y):
    return x/y

def divisor(d):
    return lambda r: divide (r, d)


half = divisor(2)


print half(32)
```

Explaining Functional Programming

- There’s more!
  - *Monads*
  - *Closures*
  - *Functors*
- Outside of the scope of today

Why Use Functional Languages?

- It’s simpler/faster to write
- If it’s a pure function, and you verified it’s right, it will always be right
- Stack traces are a pain, but in FP they simplify things

Why Use Functional Languages?

- How many have written unit tests that fail because of some state change?
- Pure functions will ALWAYS pass tests because they always return the same results with same input
- Global state of program isn’t affected by pure functions

Why Use Functional Languages?

- Concurency is WAY easier
- *Functions work well as independent units*
- *They don't have side effects*
- *Multiple functions can run simultaneously without affecting each other*

Why Use Functional Languages?

- Code ends up better as functions are designed better
- *Better small modules -> better large modules*

Activity 1

- Everyone stand up
- Count everyone in the room
- Sit down after you’re counted

Activity 2

- Everyone stand up
- Find a neighbor
- *Share your current room count*
- *One of you sit down*
- Repeat until one person remaining

Example of Functional Thinking

Activity 1 resembles a for or while loop

- x = x + 1 type thought
- Took a long time
- n steps

Activity 2 resembles concurrent recursive function

-
  ```
  def countPerson (val):
      return val + 1
  ```
- Multiple sets counted at the same time
- $\log_2 n$ steps

List of Functional Languages (Pure)

- Agda
- Charity
- Clean
- Coq
- Curry
- Elm
- Frege
- Haskell
- Hope
- Joy
- Mercury
- Miranda
- Idris
- SequenceL

List of Functional Languages (Impure)

- APL
- ATS
- CAL
- C++ (since C++11)
- C#
- Ceylon
- D
- Dart
- ECMAScript
- ActionScript
- ECMAScript for XML
- JavaScript
- Jscript
- Erlang
- Elixir
- LFE
- F#
- FPr
- Groovy
- Hop
- J
- Java (since Java 8)
- Julia
- Lisp
- Clojure
- Common Lisp
- Dylan
- Emacs Lisp
- LFE
- Little b
- Logo
- Scheme
- *Racket*
- Mathematica
- ML
- Standard ML
- *Alice*
- Ocaml
- Nemerle
- Opal
- OPS5
- Poplog
- Python
- Q
- R
- Ruby
- REFAL
- Rust
- Scala
- Spreadsheets

Languages - Elm

- Pure functional language
- Statically typed (primitive types, lists, tuples, records, unions)
- Immutable types (keeps data pure by making you create new variables)
- No runtime exceptions (compiler finds them first)
- Super friendly error messages
- Compiles to JavaScript for the browser

Languages - Haskell

- Pure functional language
- Statically typed, type inference
- Lazy evaluation and pattern matching

Languages - LISP

- “LISt Processor”
- (Known as the language with all the parentheses)
- NOT a pure functional language
- Dynamically typed (mostly lists of any type)
- If you can recursively solve your problem, then do functions on first item in list, recursively do on rest of list

Languages - Clojure

- Dialect of LISP
- Dynamically typed
- Runs on Java Virtual Machine (JVM)
- Used by Amazon, Capital One, Cerner, Groupon, Spotify, many others

Languages – F#

- Functional and Object Oriented (compiles into .Net)
- Based on Ocaml and C#
- Strongly typed, but inferred
- Every statement returns a type
- Parallelism is easily built into language
- Great for data analysis

Conclusion

- Functional Programming is getting popular, but been around for decades
- Adopting functional principles will make your code simpler, smaller, and more reliable
- Several different types of functional languages and how they’re built

Thank You!

Sarah Withee
@geekygirlsarah on Twitter
sarah@sarahwithee.com

I love to get feedback as well as hear how you use the new knowledge.

Reach out!