AN OVERVIEW OF FUNCTIONAL PROGRAMMING

Presented By
Andrew Butt

Overview

- A Brief History of Functional Programming
- Comparison Between OOP and Functional programming
- Paradigms and Concepts
- Functional Programming in Other Languages

An Overview of Functional Programming

A Brief History of Functional Languages

What is Functional Programming?

- Programming Paradigm with roots in Lambda Calculus and Combinatory logic
- Lambda calculus provides a formal system for definition, function application and recursion
- Treats computation as evaluation of mathematical functions without states
- Realizes a computation by composing functions

History

- On of the first languages LISP was developed in 1950s at MIT for IBM scientific computers
- Languages developed throughout 60s and 70s
- Haskell was released in 1980s in an attempt to unify many functional languages
- Referred to as the “Algebra of Programming”

An Overview of Functional Programming

Comparison Between Object Oriented Programming and Functional programming

Object Oriented Programming

- OOP has Objects
- Objects hide data, encapsulate
- Objects perform a set of related operations through methods
- Objects are capable of storing data on the current state of itself or other objects
- Objects are highly reusable

Functional Programming

- Takes a set of instructions to perform a task
- Selectively executing instructions can perform a task
- Pure Functions contain no mutable data
- Pure Functions are calculated solely on the data passed into them
- Calling foo(a,b) will always produce the same result

Definition of Function

- Functional Languages use “Function” in the mathematical use of the word
- Map input values to the output values

- Imperative “Functions” can be considered subroutines which subroutines with states and mutable data

Pure Functions

- Have no memory
- Will always result in the same answer when called with equivalent parameters
- If all calls are Pure functions then very efficient optimization is capable through the complier, as functions can be reordered or combined as needed.

Higher-order functions

- Can only take other functions as arguments
- Can return functions
- Analogous to returning *d/dx* when returning a derivative function
- Can enable currying: a function takes multiple arguments in such a way that it can be called as a chain of functions each with a single argument

Currying Example

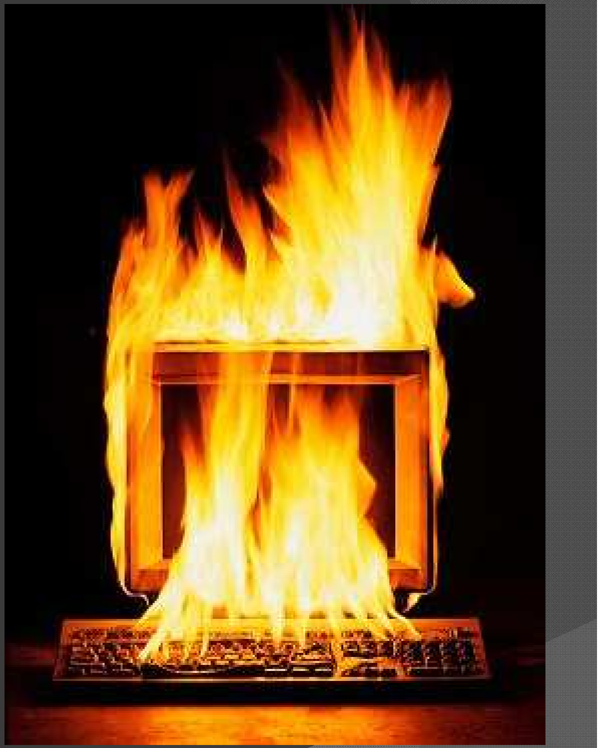
- Using $F(x,y) = x^2 / y^3$
- Evaluate F(5,5)
- Replacing x with 5 results in a new function in y: $g(y) = 5^2 / y^3$
- Replacing y with 5 results in: $g(3) = 25 / 5^3$
- $g(3) = 1/5$
- Each step results in a more simplified expression

Functions

- Functions don’t “DO” anything!
- That is they only return a value, no “side effects” will occur after the execution of a function
- For example no files can be written using pure functions and no variables will be changed in memory

A Little White Lie

- No file or I/O would do little more than warm up your computer
- Functional languages can actually write data and I/O using Non-Pure functions
- Purely functional languages only allow this inside language constructs
- Greatly limits "side-effects"

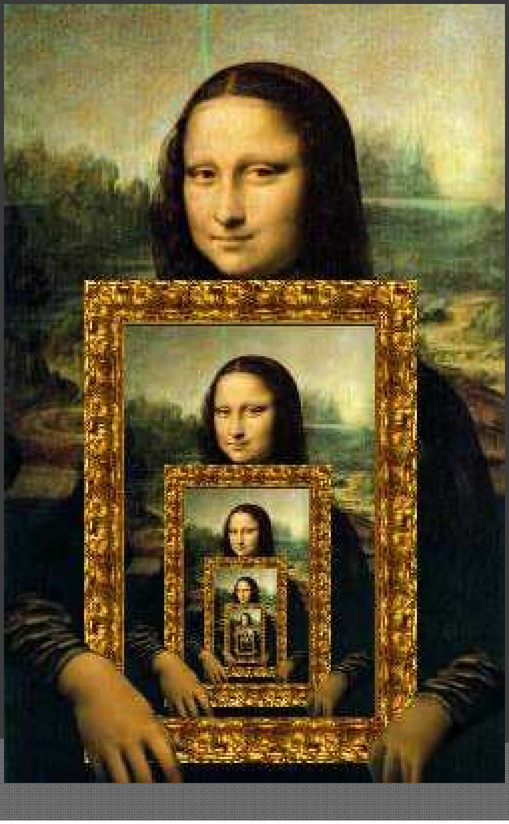


Faking States

- Many programs are closely tied to idea of states
- Functional languages can use monads to use I/O and mutable data
- Abstract this functionality away to maintain pureness

Functional Programming

- No loops! NONE! Nadda! Zip! Zilch!
- Loops are replaced with recursion



Efficiency

- Slower in many cases than imperative languages
- Very efficient at large matrix calculations
- Optimized for array functional languages

An Overview of Functional Programming

Paradigms and Concepts

Strict Evaluation

- In strict evaluation any function which contains a failing term will also fail
- Eg: print length([5+2], 3*3, 6/0]) will fail due to divide by zero error

Non-Strict (lazy) Evaluation

- Length will return 3, as its terms are not evaluated
- Lazy evaluation does not fully evaluate the expression before invoking a function.

Coding Techniques

- Steps are usually combined to emphasize composition and arrangement of functions, often without explicit steps defined.

```
Imperitive style:
    target = List[];
    for (item : source){
        x = G(item)
        Y = F(trans1)
        target.append(trans2)
    }

Functional Style:
    compose2 = lambda A, B: lambda x: A(B(x))
    target = map(compose2(F, G), source)
```

Recursion

- Widely used in functional languages
- Largely replaces iteration, as functions invoke themselves
- Most functional languages allow unrestricted recursion and are Turing Complete
- Halting Problem is undecidable in many Functional Languages

Problems with Functional Programming

- As systems grow they become a large collection of functions
- All of these functions are interconnected
- Changing one function breaks all others relying on it
- Very hard to manage on large scales

An Example: Functional Factorial

Haskell:

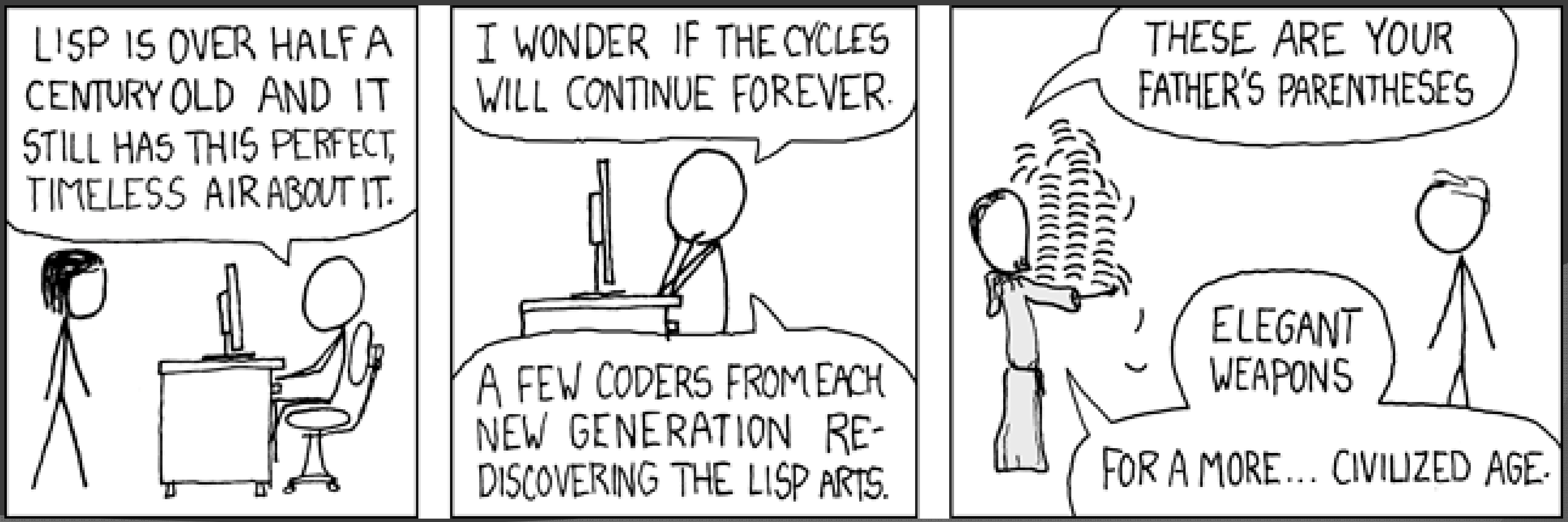
```
factorial :: Integer -> Integer
factorial 0 = 1
factorial n | n > 0 = n * factorial (n-1)
```

- Line 1: defines the factorial function to take an integer and return an integer
- Line 2: Return 1 if input is 0
- Line 3: if n > 0 call factorial on itself

An Example: Functional Factorial

Common Lisp:

```
(defun factorial (n)
     (if (<= n 1)
            1
            (* n (factorial (- n 1)))))
```


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

An Overview of Functional Programming

Functional Programming in Other Languages

Functional Programming in C

- Function pointers can be used in similar fashion as “higher-order” functions
- In C# lambda functions can used to program in a functional style
- Lazy evaluation can be used for lists in C
- Closures are possible in C through the use of pointers

Functional Programming in Java

```
Runnable foo= new Runnable() {
  public void run() {
     bar();
  }
};
```

- Bar is enclosed within the Runnable foo, and can be passed between methods as if it were data and executed at anytime by foo.run()

Why Use Functional Programming?

- Advantages in Parallel and concurrent programming by eliminating race conditions and locking of mutable data
- Very common in research and academia (Mathematica is a functional language)
- Testing can be easier as every function can be seen as independent

Why You May Not Want To Use Functional Programming!

- It requires a lot of overhead learning (and unlearning!)
- Most computer hardware implement optimization for imperative techniques
- Many problems are simply better suited for OOP or similar techniques

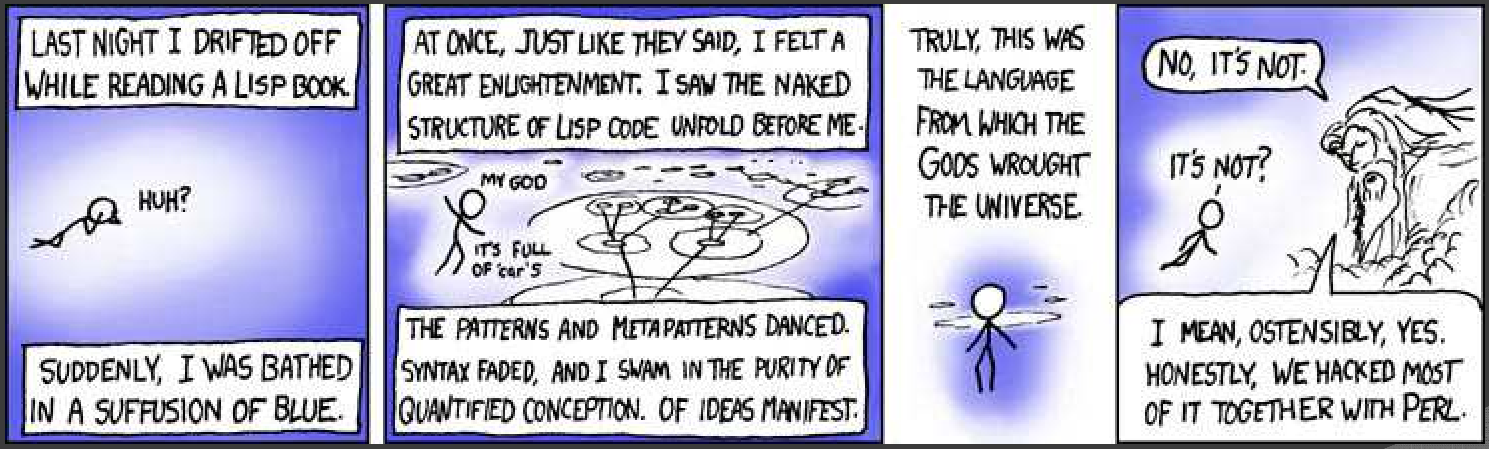
References

or…

How I learned to Stop Asking Questions and RTM

- Konrad Hinsen, "**The Promises of Functional Programming**," Computing in Science and Engineering, vol. 11, no. 4, pp. 86-90, July/Aug. 2009, doi:10.1109/MCSE.2009.129
- Turner, D.A, “**Total Functional Programming**,” Journal of Universal Computer Science, vol. 10, no. 7, Jun 04
- Hughes, John, “**Why Functional Programming Matters**”, 1984

Thank You!


LAST NIGHT I DRIFTED OFF WHILE READING A LISP BOOK.
HUH?
SUDDENLY, I WAS BATHED IN A SUFFUSION OF BLUE.
AT ONCE, JUST LIKE THEY SAID, I FELT A GREAT ENLIGHTENMENT. I SAW THE NAKED STRUCTURE OF LISP CODE UNFOLD BEFORE ME.
MY GOD
IT'S FULL OF 'car'S
THE PATTERNS AND METAPATTERNS DANCED. SYNTAX FADED, AND I SWAM IN THE PURITY OF QUANTIFIED CONCEPTION. OF IDEAS MANIFEST.
TRULY, THIS WAS THE LANGUAGE FROM WHICH THE GODS WROUGHT THE UNIVERSE.
NO, IT'S NOT.
IT'S NOT?
I MEAN, OSTENSIBLY, YES. HONESTLY, WE HACKED MOST OF IT TOGETHER WITH PERL.