PWL-BSB # 13:
"**Why Functional Programming Matters**"
John Hughes

Papers We Love Brasília

Presenter: Alessandro Leite

January 17, 2019

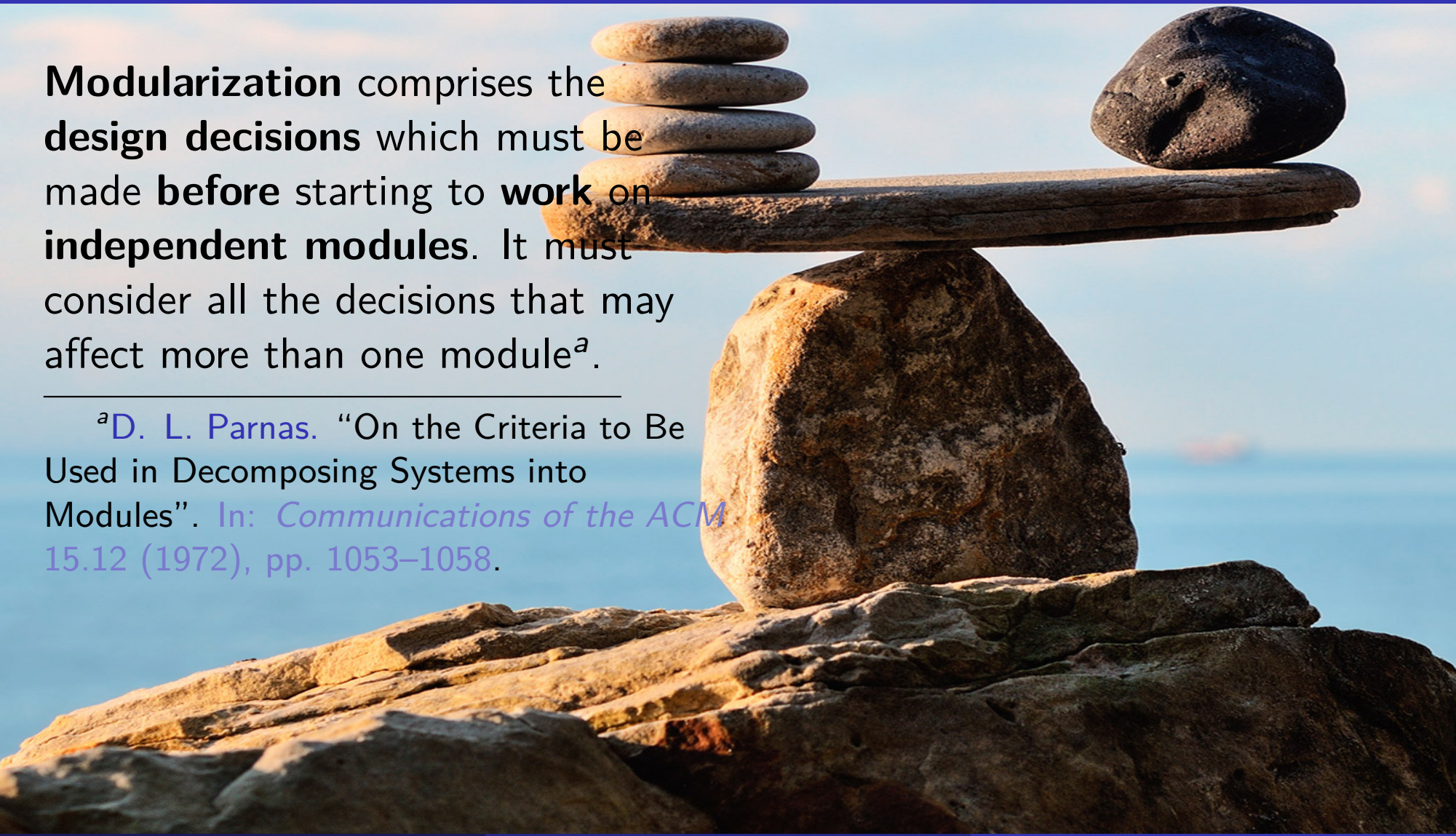
It's all about modularity

"A well-defined segmentation of the project effort ensures system **modularity**. **Each task** forms a **separate**, distinct program **module**. At implementation time each module and its **inputs** and **outputs** are **well-defined**, there is no confusion in the intended interface with other system modules"[a]

[a] Richard L Gauthier and Stephen D Ponto. *Designing systems programs*. Prentice-Hall, Englewood Cliffs, 1970.

But, what is modularity

Modularity involve decisions that impact systems' independences



Modularization comprises the **design decisions** which must be made **before** starting to **work** on **independent modules**. It must consider all the decisions that may affect more than one module[a].

[a] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Communications of the ACM* 15.12 (1972), pp. 1053–1058.

The Next 700 Programming Languages

P. J. Landin

Univac Division of Sperry Rand Corp., New York, New York

"... today ... 1,700 special programming languages used to 'communicate' in over 700 application areas."—*Computer Software* Issues, an American Mathematical Association Prospectus, July 1965.

- Proposed an abstract programming language, named ISWIM (**I**f you **S**ee **W**hat **I** **M**ean)[a]
- It has influenced the development of subsequent programming languages, especially functional programming languages such as Miranda, ML, and Haskell
- **"Expressive power should be by design, rather than by accident"**

---

[a] P. J. Landin. "The Next 700 Programming Languages". In: *Communication of the ACM* 9.3 (1966), pp. 157–166.

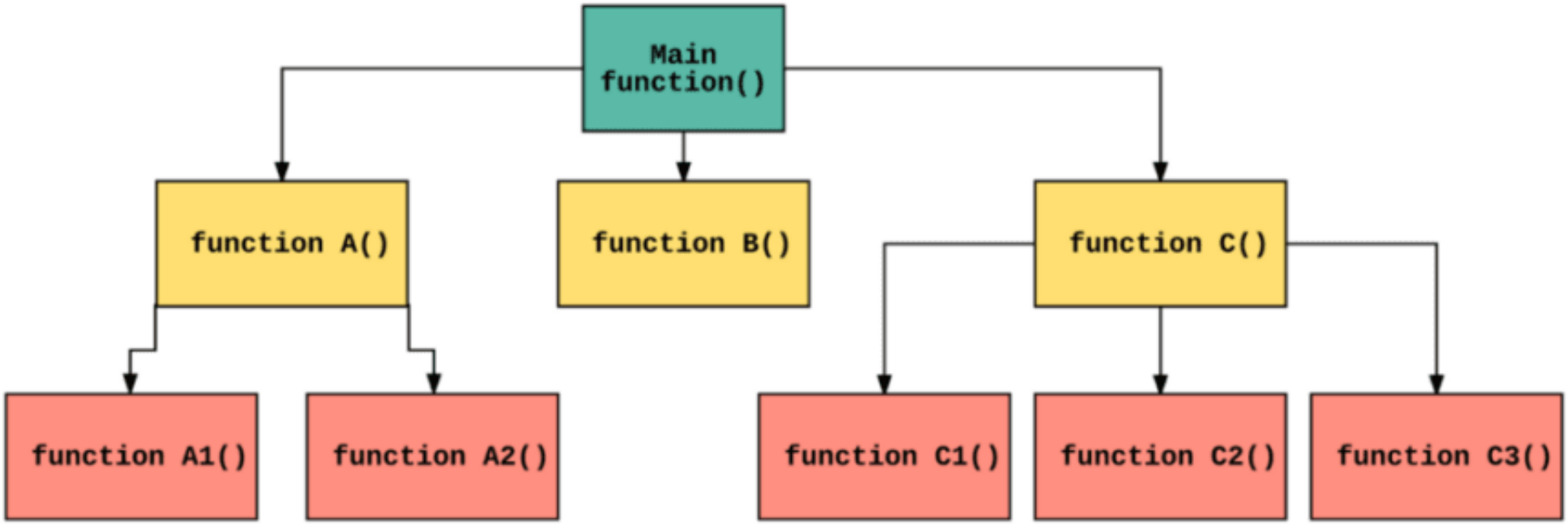
Modular design is the key to successful programming

"When writing a modular program to solve a problem, one first divides the problem into subproblems, then solves the subproblems, and finally combines the solutions. ***The ways in which one can divide up the original problem depend directly on the ways in which one can glue solutions together. Therefore, to increase ones ability to modularize a problem conceptually, one must provide new kinds of glue in the programming language****."*

This paper attempts to show how functional programming helps on modularization

"This paper is an attempt to demonstrate to the "real world" that ***functional programming*** *is* ***vitally important****, and also to* ***help functional programmers*** *exploit its advantages to the full by making it clear what those advantages are."*

Functional codes consist entirely of other functions



People usually highlights what functional code doesn't have

- contain no assignment statements
- no side-effects at all
- Expressions can therefore be evaluated at any time and replaced by their values, and vice-versa; i.e., programs are "referentially transparent"

Such a catalogue of "advantages" *is all* ***very well****, but one must* ***not be surprised if outsiders don't take it too seriously****.* ***It says a lot about what functional programming is not*** *(it has no assignment, no side effects, no flow of control)* ***but not much about what it is****.*

Functional programmers also argue about the benefits of functional programming

- functional programmers are an order of magnitude more productive than their conventional counterpart
- functional programs are an order of magnitude shorter
- *The only plausible reason one can suggest to explain these "advantages" is that conventional programs consist of 90% assignment statements*, and in functional programs they can be omitted!
- Clearly, this characterization of functional programming is inadequate
- We must **find something to put in its place — something** which **not only explains the power of functional programming**, but also **gives a clear indication of what the functional programmer should strive towards**

Structured programming is not only about the absence of goto expression

- The most important difference between **structured** and **unstructured** programs is that **structured programs are designed in a modular way**
- Modular design brings with it great productivity improvements:
  - Small modules can be coded quickly and easily
  - General purpose modules can be reused, leading to faster development of subsequent programs
  - The modules of a program can be tested independently, helping to reduce the time spent debugging
- The **absence** of **gotos** helps with programming *"in the small"*, whereas **modular design** helps with programming *"in the large"*

Focusing on what really matter

- *"We shall argue in the remainder of this paper that **functional languages provide two new**, very important **kinds of glue**"*:
  - High-order functions
  - Lazy evaluation
- This is the key to functional programming's power — **it allows improved modularization**. It is also the **goal** for which **functional programmers must strive** — smaller and simpler and more general modules, glued together with the new glues."

What is a higher-order function?

Higher-order function

- It is a function that:
  - takes one or more functions as argument
  - returns a function as a result.
- **Higher-order functions** enable simple functions to be glued together to make more complex ones.
- For example:

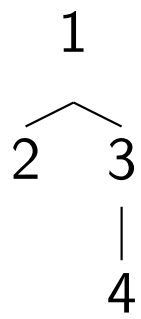
```
add x y = x + y
sum = reduce add 0
```

- **add** is a function which takes two arguments, *x* and *y* and returns their sum
- **reduce** is the higher-order function which has **add** as its argument and 0 is the first argument of *add*

Higher-order function

```
treeof X :: = node X (listof (treeof X))
```

- a tree of *Xs* is a **node** with a label in *X*
- a list of subtrees are also trees of *Xs*
- For example, the tree



- can be represented as

```
node 1
     (cons (node 2 nil)
           (cons (node 3
                       (cons (node 4 nil) nil))
                 nil))
```

Higher-order function

*"All **this can be achieved** because functional languages **allow functions that are indivisible in conventional programming languages to be expressed as a combinations of parts** — a general higher-order function and some particular specializing functions".*

Lazy evaluation

Lazy evaluation

- Also known as *call-by-need evaluation strategy* which delays the evaluation of an expression until it is needed, avoiding repeated evaluation
- Lazy evaluation makes it practical to modularize a program as a **generator** that constructs a large number of possible answers, and a *selector* that chooses the appropriate one.
- "**Can lazy evaluation and side-effects coexist?** *Unfortunately they cannot*: **adding lazy evaluation to an imperative notation is not actually impossible**, but the *combination would make the programmer's life harder rather than easier*."


REASON

*"**Lazy evaluation's power depends on the programmer giving up any direct control over the order in which the parts of a program are executed**, it would make programming with side effects rather difficult, because predicting in what order – or even whether – they might take place **would require knowing a lot about the context in which they are embedded**. Such global interdependence would defeat the very modularity that – in functional languages – lazy evaluation is designed to enhance."*

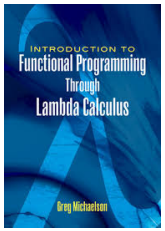
We can conclude that

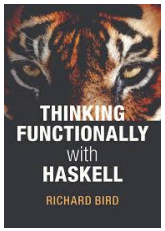
- Modularity is the key to successful programming
- **Languages** which aim to improve productivity **must support modular programming**
- **Modularity** goes **beyond modules**
- The ability to decompose a problem into parts depends on the ability to **glue solutions together**
- A **language** must provide **good glue** to support modularity
- Functional programming languages provides two new kinds of glue — higher-order functions and lazy evaluation

Going further

video: **John Hughes's keynote at Functional Conf 2016**, *youtu.be/XrNdvWqxBvA*

podcast: **Functional Programming Languages and the Pursuit of Laziness with Simon Peyton Jones**, *bit.ly/2FzuyFX*

- 

Greg Michaelson. *An introduction to functional programming through lambda calculus*. Courier Corporation, 2011
- 

Richard Bird. *Thinking functionally with Haskell*. Cambridge University Press, 2014


That's all Folks!