1: IDENTIFYING THE PROBLEM:

The problem is scaling. We will narrow this scope to the scaling of the logic of a program. Furthermore we can narrow that scope to keeping a program in comprehensible state, since that is the real issue people have when scaling logic and they just notice it by their inability to scale larger.

Reasons for incomprehensible state:

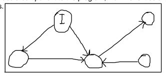
- Too large amount of context -> which context gets in the way?
- Unclear structure -> solutions: Literate Programming or Module-Structural Programming (MSP)

Literate programming tries to reduce the complexity in the program used for a complex algorithm.

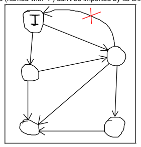
The modules form components in a structure representing the structure of the logic of the porgram. A downside? of this is that the complexity of the program is inherently tied to the complexity of the alogrithm. That said, that complexity is not what worries us, since our scope is finding a solution to problems arising from scaling, not problems arising from a certain given algorithm. We will break down the logic in components (the term "component" will be used here in its most abstract and generic meaning, under components may fall functions and/or data). Furthermore we'll only worry ourselves with immutable modules for now. The reason for that being that we start off simple. Imagine a house for instance, made on top of moving bricks, it doesn't really work or it's at least very complex. We will get back to mutual modules later.

2: USING MODULE-STRUCTURAL PROGRAMMING TO FIX OUR NARROWED SCOPE OF SCALABILITY ISSUES

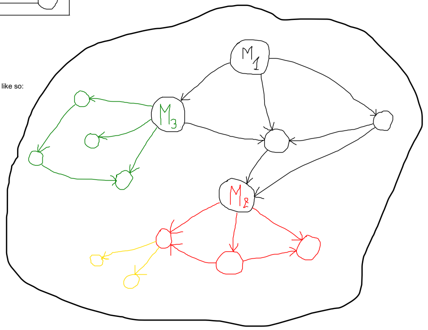
(2.1) A module provides a component to the program, we'll make it so any acyclic graph of the program can be made from modules.



(2.2) A module can be built from components. They can form any such acyclic graph. It becomes trivial to see that that module (named with "I") can't be imported by its children whilst maintaining its acyclic attribute.



=> All programs are formed like so:



3: CODE IS MODULAR AND UNDERSTANDABLE NOW, BUT WHAT ABOUT CODE REUSABILITY?

* MULTIPLE COMPONENTS DOING THE SAME THING ONLY ONE CLUSTER AWAY FROM EACHOTHER FOR INSTANCE CAN BECOME A HUMAN SOURCE OF ERROR BECAUSE HUMAN LOGIC IS RIDDLED WITH ASSUMPTIONS.

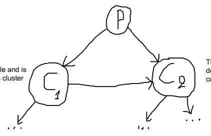
* WE COULD SPEED UP THE PROCESS OF REIMPLEMENTING SOMETHING 5 TIMES TO ONLY ONCE. BESIDES WE MIGHT NOT WANT TO RUN THE RISK OF FORGETTING AN EQUIVALENT/ISOMORPH MODULE.

THAT ALL BEGS THE QUESTION: "WHERE DO WE WANT CODE TO BE MODULAR AND WHERE TO BE REWORKABLE IN BULK?"

First of all it is easy to see that a module has to be a "leaf" cluster in order to even be considered something different than a component in a program. We'll accept here that components should always be written on an individual level and never in bulk. When we do have a "leaf" module, it might be used only in one specific cluster or in many. If it is used in only one specific cluster it's still regarded as a component in that cluster.

When the module spans across many different clusters, we have 2 options:
1) Remake the graph of the project, often by merging some clusters into one
2) Define the module as a language extension. A language that can define identifiers for immutable (keep in mind that we still find ourselves in the realm of immutable modules) data and functions really just defines extra vocabulary for an already given language. So if such vocabulary is provided in multiple clusters of the graph, then we say it is a language extension (of the language's vocabulary). Another term we use for it is a "library".

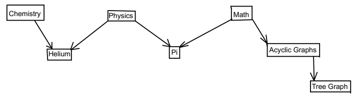
This is not a "leaf" module and is thus a component in this cluster



This is a "leaf" module and thus it is yet to be decided on whether or not this should be a component or a language extension/library

STRUCTURING COMPONENT MODULES DEPENDS ON THE LOGIC YOU WANT TO IMPLEMENT, BUT HOW DO WE STRUCTURE LIBRARIES/LANGUAGE EXTENSION MODULES?

Solution: Putting the library folder completely separate from the program structure, and forming another graph which facilitates the process of finding a required language extension:



This way the programmer can easily find the right libraries he needs. For instance: You want Pi, You know it has to do something with to with physics, so you look there. You find Pi. You could have also associated Pi with math and found it from there.

NOT DISCUSSED: mutable modules and spawnable modules (which only makes sense for mutable modules)