1 Introduction

In this course we have spent a decent chunk of our time looking at the advantages of functional programming. We first discussed LISP and the innovations it brought, then we looked at the reading "Why Functional Programming Matters", which gave a detailed account of some benefits of functional programming and now we are learning F#. It is safe to say, then, that functional programming is the unofficial theme of this course. But, if functional programming is so useful, why was it unfamiliar to most of us before this class? Why is it regarded as something complex and scary when it is neither, when, in fact, at its core it is very simple and elegant? In my view it is because it is because it is not introduced early in a programmer's life and so, when the time comes for one to learn it, they are already familiar with other common programming paradigms and they have a hard time adapting to the new way of thinking and coding.

This is the problem that my project, tentatively named **FUNny** , is aiming to solve. **FUNny** will consist of a simple functional programming language, and a visual programming environment for that language. The visual environment will work based on the simple view of functions as machines that transform the input given to them into output, just like we have seen multiple times in class. The programmer will have some pre-built machines at their disposal and the ability to compose them by piping their outputs to other machines in order to create more complex machines. **FUNny** is aimed mainly to children as an introduction to functional programming. It's goal is to demystify functional programming, illustrate the simplicity of its core concepts in a fun, enjoyable way and facilitate its learning, just like visual programming environments like Scratch are helpful as introductions to the imperative paradigm.

2 Design Principles

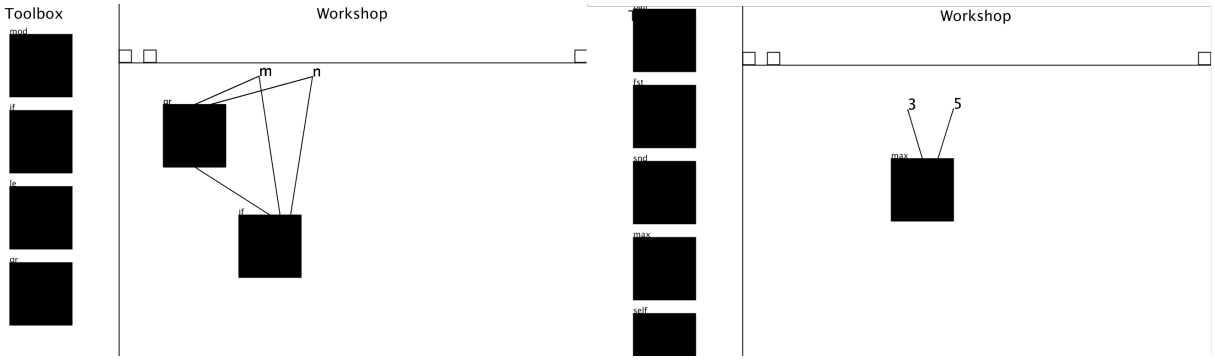
From now on, **FUNny** will be used to refer to the project as a whole. The language will be refered to as **FUNny Language** and the interface as **FUNny Visual Interface** .

Naturally, **FUNny Language** will be a simple functional programming language. It will be based on the idea of pure functions and declarative statements. My aim is for the code to be highly readable and descriptive, so as to facilitate the understanding of the code and its correlation with the **FUNny Visual Interface** by younger children who may have a hard time reading sybmolic expressions. Since the **FUNny Language** is not meant to be written, but rather generated using the **FUNny Visual Interface** , it can be very strict in its form (the use of whitespace for example), since automatically generating consistent code is fairly simple.

FUNny Visual Interface will be a visual interface utilizing the analogy between functions and machines that convert input into output. What will guide the implementation of the **FUNny Visual Interface** will be the design of the **FUNny Language** , as I would like them to feel strongly correlated, and also its design highly depends on the features that end up being implemented in the language. Aesthetically speaking, the **FUNny Visual Interface** will be minimalistic, not only because I find that style to be pleasing, but also because I lack the ability to create any more intricate design.

3 Example Programs

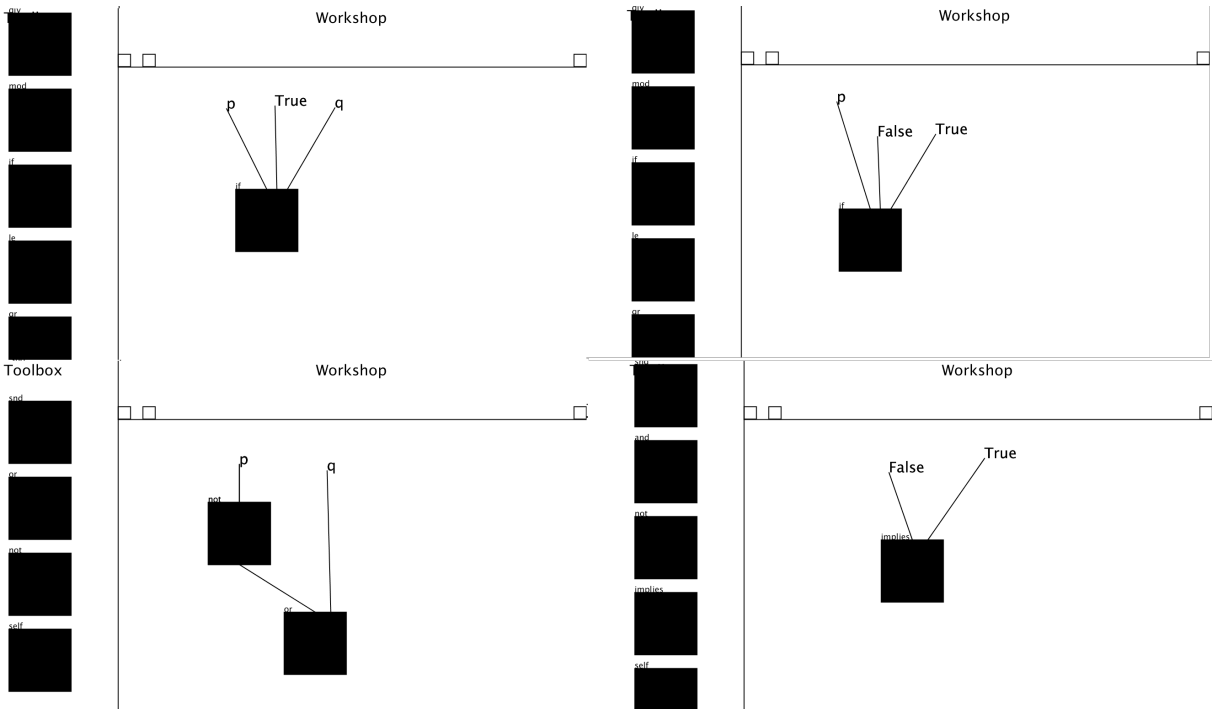
1.



In the left picture above, a new function is defined, which finds the maximum of two numbers. And in the right one, it is called with the arguments 3 and 5.
The text of the program that the above generates is

```
(fun max m n ( if ( gr m n ) m n ))
( max 3 5 )
```

2.

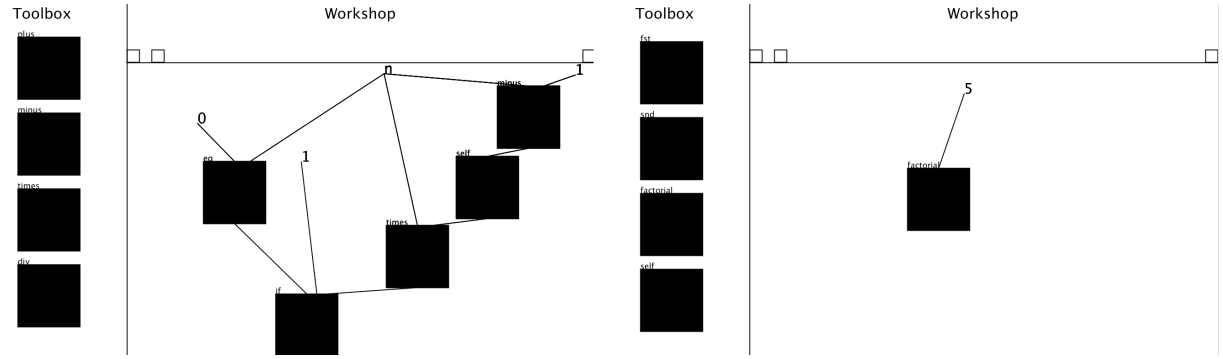


In the top two picture above, some new functions is defined: the boolean operators *or, not*. In the lower-left one, they are combined to define the function *implies* which calculates the matching logic operator for two values. Then *implies* is called with the arguments False and True.

The text of the program that the above generates is

```
(fun or p q ( if p True q ))
(fun not p ( if p False True ))
(fun implies p q ( or ( not p ) q ))
( implies False True )
```

3.



In the left picture above, a new function is defined, which finds the factorial of a number. And in the right one, it is called with the argument 5.
The text of the program that the above generates is

```
(fun factorial n ( if ( eq 0 n ) 1 ( times n ( factorial ( minus n 1 ) ) ) ))
( factorial 5 )
```

4 Language Concepts

The core concept of the **FUNny Language** is functions. Currently the **FUNny Language** supports of functions of integers, booleans and pairs. They are always pure functions, meaning that their only effect when given an input is to produce an output. There is no mutable data in **FUNny Language** and so variables don't play an important role, except for passing arguments to functions. Recursion is also a very important concept as it is in any functional programming language, since it is the sole way of implementing complex functionality and it is also supported in the current version of the language.

5 Syntax

6 Semantics

7 Remaining Work