

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 referred 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 symbolic 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.

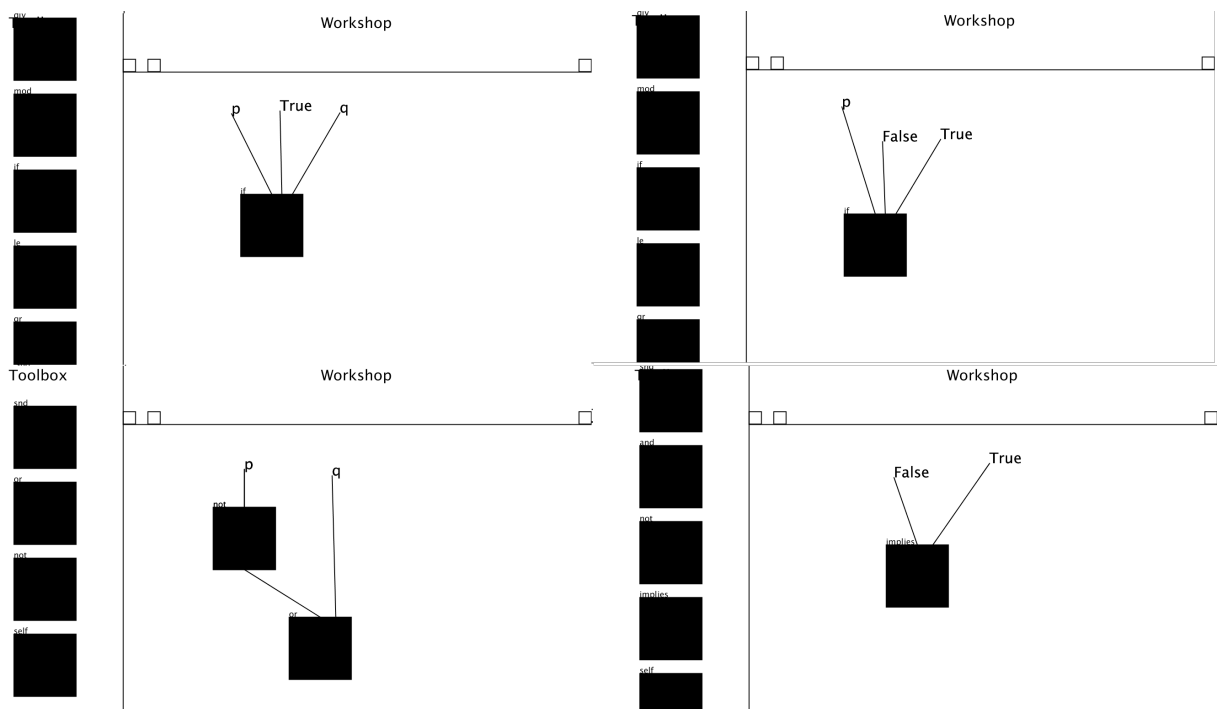


In the left picture above, a new function is defined, which finds the maximum of two numbers. And in the right one, it is called with the arguments 3 and 5.

The text of the program that the above generates is

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

2.

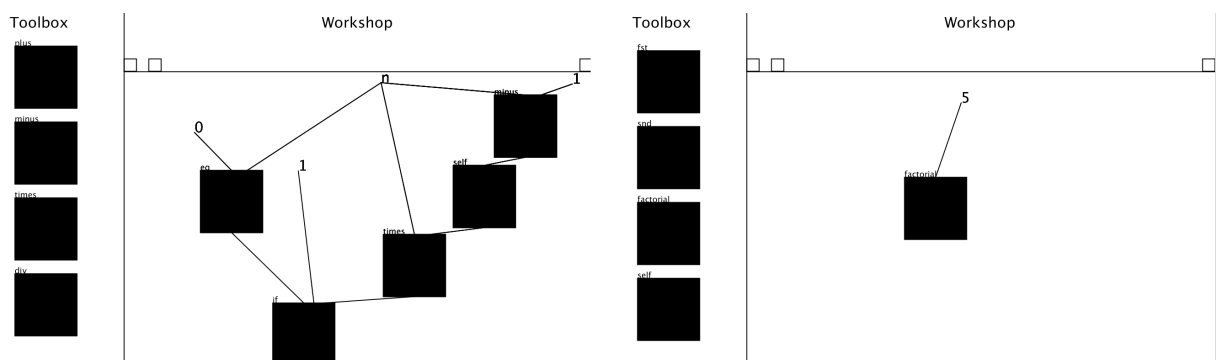


In the top two picture above, some new functions is defined: the boolean operators *or*, *not*. In the lower-left one, they are combined to define the function *implies* which calculates the matching logic operator for two values. Then *implies* is called with the arguments False and True.

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 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

The current grammar for the language is the following:

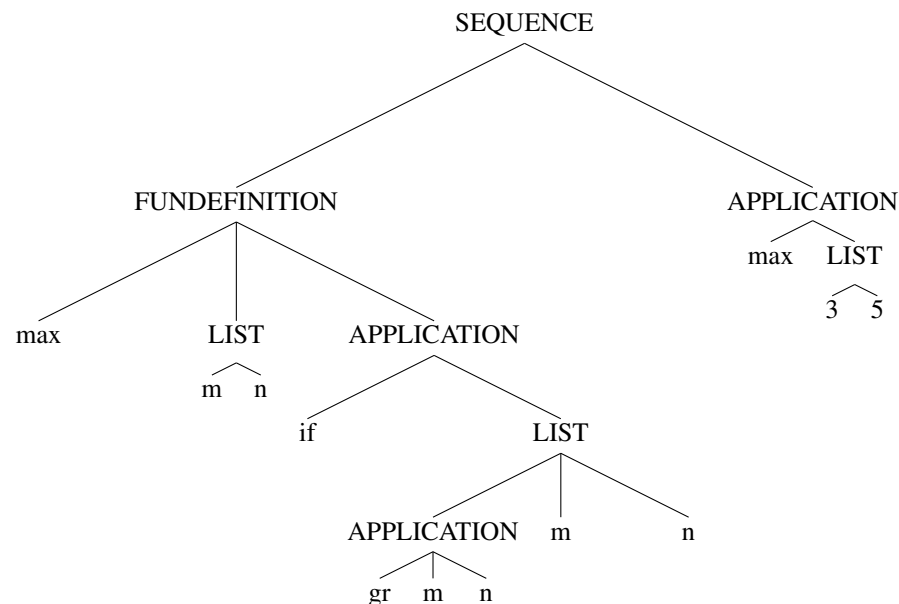
```

<expression> ::=
    | <sequence>
    | <value>
    | <application>
    | <fundefinition>
<sequence> ::=
    | <value>
    | <application>
    | (<fundefinition>, <sequence>)
<value> ::= <primitive>
<application> ::= <word> <expression>+
<fundefinition> ::= <word> <word>+ <expression>
<primitive> ::=
    | <bool>
    | Nil
    | <number>
    | <pair>
    | <arg>
<bool> ::= True | False
<number> ::= <d><number> | <d>
<d> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<pair> ::= [<primitive>, <primitive>]
<arg> ::= <word>
<word> ::= <l><word> | <l>
<l> ::= a | b | c | d | e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t | u
        | v | w | x | y | z
        | A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U
        | V | W | X | Y | Z

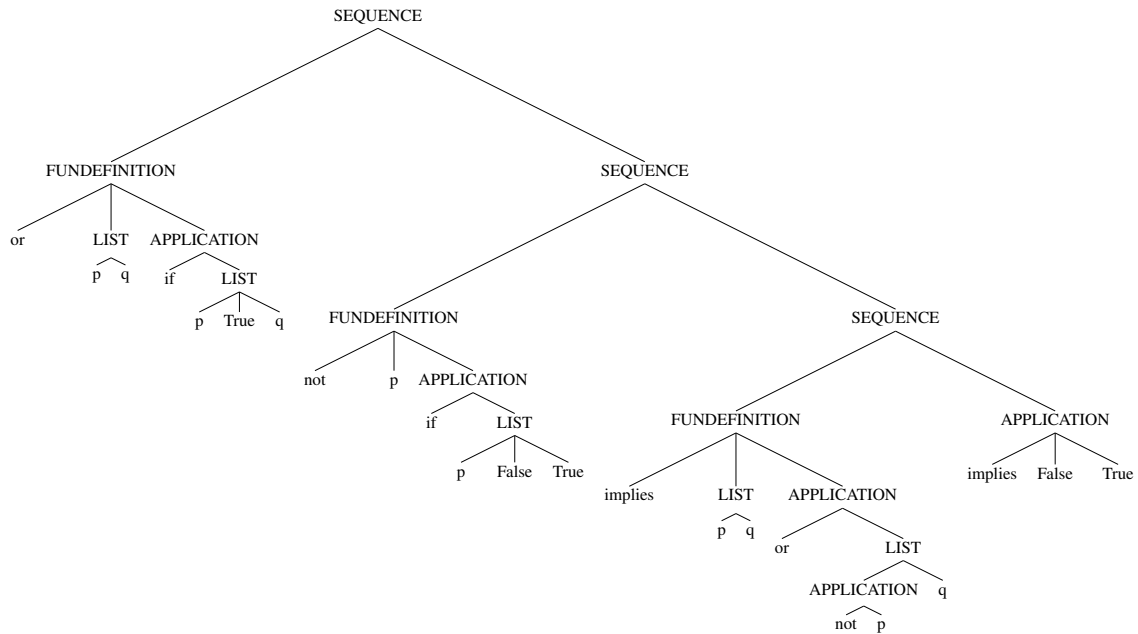
```

To understand the syntax better let's look at the ASTs generated from the example programs:

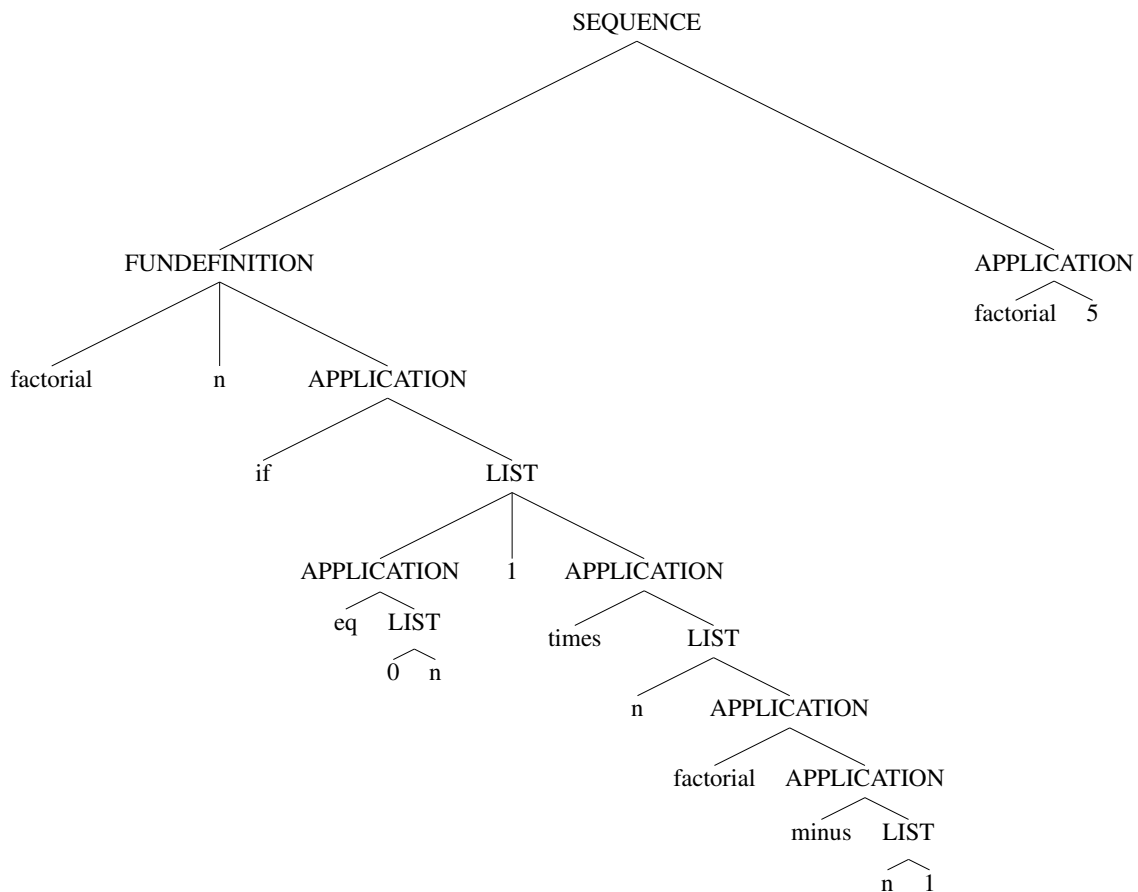
1.



2.



3.



For the **FUNny Visual Interface**, the "syntax" is fairly intuitive. Taking a "machine" from the toolbox to the workspace allows the programmer to connect inputs to it by first clicking on the input and then on the machine, and connect its output to different machines by clicking on it and then clicking on the other machine.

6 Semantics

Syntax	Abstract Syntax	Meaning
n	Value (Number of int)	A primitive value holding an integer. It is stored as an F# integer.
True	Value (Boolean True)	A primitive value: holds the boolean value true.
False	Value (Boolean False)	A primitive value: holds the boolean value false.
Nil	Value Nil	A primitive value: holds the value nil (which can stand in for things like an empty list)
word	Value (Arg of string)	A primitive value: holds the name of the argument of a function. It is only meant to be used as part of function definitions.
(name a1 a2 ... an)	Application name [a1; a2; ...; an]	Application of arguments a1, a2, ..., an to the function 'name'
(fun name a1 ... an exp)	FunctionDefinition name [a1;...;an] exp	Define the word 'name' to correspond to the expression 'exp' substituting values for a1, ..., an.
exp1 exp2	Sequence (exp1, exp2)	Sequential evaluation of exp1, exp2. exp1 has to be a function definition.

Types in the **FUNny Language** are checked at runtime. All expressions are parenthesized so there is no need to define precedence and associativity.

As for the semantics of the GUI, I will try to describe them here:

A line connecting a primitive to function means that that primitive is being passed as an argument to that function. The arguments are passed to functions from left to right (meaning that the first argument is the one whose line connects farther to the right than all the other arguments, the second is the next one, etc).

For functions, lines coming in on the top are the arguments and a single line coming out of the bottom of the box is the output.

Any input that is not a number or "True", "False" or "Nil" is interpreted as an argument, and the function being built is interpreted as a function definition and not as a function application. Those arguments can be passed around as values and they will be substituted with concrete values upon applying the function.

Recursion is supported through the use of the self function. The self function is available in the toolbox and should only be used in function definitions, not applications. It stands in for the function being built as a whole and produces the appropriate code when it is run. It works like any other function, taking in inputs and producing an output.

7 Remaining Work

I want to work on adding real numbers in the **FUNny Language** and higher order functions, since those two would allow for much more interesting programs. Especially for higher order functions it will be quite the challenge, so I don't know whether these features will make it into the project.

Other than that I want to improve the appearance and usability of the **FUNny Visual Interface** because at its current form it is very hard to use and understand.