Software Project Specification

for

Reimplementation of Pygmalion

Pygmalion was a proof of concept visual programming system put forward in the year 1975. It involved programming using so-called icons, and creating functions by remembering user actions, instead of writing them in code. This project focuses on creating a simple programming system with the same key ideas as Pygmalion.

1.0.0

Adrián Habušta

May 14, 2023

Contents

1 Basic Information **2**

1.1 Description of Software Project 2

1.2 Used technologies 2

1.3 References 2

2 Detailed functionality **2**

2.1 Icon representation 2

2.2 Evaluation 3

3 Screens **4**

3.1 Single Tab 4

4 Usage example - Factorial **4**

5 Other requirements **10**

6 Project restrictions **10**

7 Timeline & Milestones **10**

Appendix A: Terminology **10**

1 Basic Information

1.1 Description of Software Project

This software project is inspired by an old programming system called Pygmalion. Pygmalion never moved past the prototype stage, and this project is a proof of concept that is meant to show how using a system with ideas borrowed from Pygmalion would be like. These ideas include representing data, methods and even objects with icons. Another key idea is creating methods by remembering user actions. This is what this project will focus on the most.

1.2 Used technologies

- F# - used programming language
- Fable - F# to JS compiler

1.3 References

- Pygmalion specification
  - `https://apps.dtic.mil/sti/pdfs/ADA016811.pdf`

2 Detailed functionality

2.1 Icon representation

Internally, custom icons will be stored in a Dictionary mapping from names to specific icons. The data will look something like this:

```
type CustomIcon =
    { InstructionTree : Instruction
      ParameterCount : int
      IconContext : DrawnIcon list }

and DrawnIcon =
    { Name : string
      xPosition : int
      yPosition : int
      Result : int Option
      Parameters : int Option list }
```

The CustomIcon type represents a sort of ”master” icon, which stores all data related to an icon. This means both the instructions of said icon, and also information necessary for drawing the icon to the screen. This is done by using the DrawnIcon, which only stores iformation used for drawing an icon.

2.2 Evaluation

Icons are evaluated by building simple trees of instructions. The instructions will look something like this:

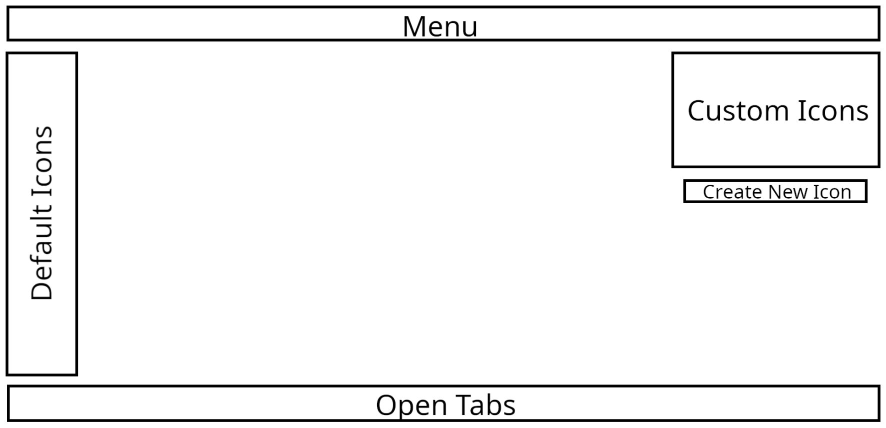
```
type Instruction =
    | Trap
    | Primitive of int
    | Unary of char * Instruction
    | Binary of char * Instruction * Instruction
    | If of Instruction * Instruction * Instruction
    | Icon of string * Instruction list
    | Parameter of int

type Context =
    { Parameters: int list }

let fetchIconInstruction name =
    Trap

let rec eval iconContext instruction =
    match instruction with
    | Primitive n -> n
    | Trap -> failwith "Trap sprung"
    | Unary(operator, operand) -> 0
    | Binary(operator, leftOperand, rightOperand) -> 0
    | If(cond, trueBranch, falseBranch) ->
        let res = eval iconContext cond
        if res = 0 then eval iconContext trueBranch
        else eval iconContext falseBranch
    | Icon(name, parameters) ->
        let evaluatedParameters = List.map (eval iconContext) parameters
        let newContext = {iconContext with Parameters = evaluatedParameters}
        let nextInstruction = fetchIconInstruction name
        eval newContext nextInstruction
    | Parameter index -> iconContext.Parameters[index]
```

Icons will, by default, only contain a Trap, which when hit during runtime, opens the icon for editing. During editing, operations made by the user will slowly build up an instruction tree that is stored and later used to evaluate instances of said icon. Using custom icons within an icon is done by saving the name of the icon we wish to evaluate.

3 Screens

3.1 Single Tab

This is the only screen the software will feature. There will, however, exist multiple instances of this screen called 'tabs' that can be switched between.

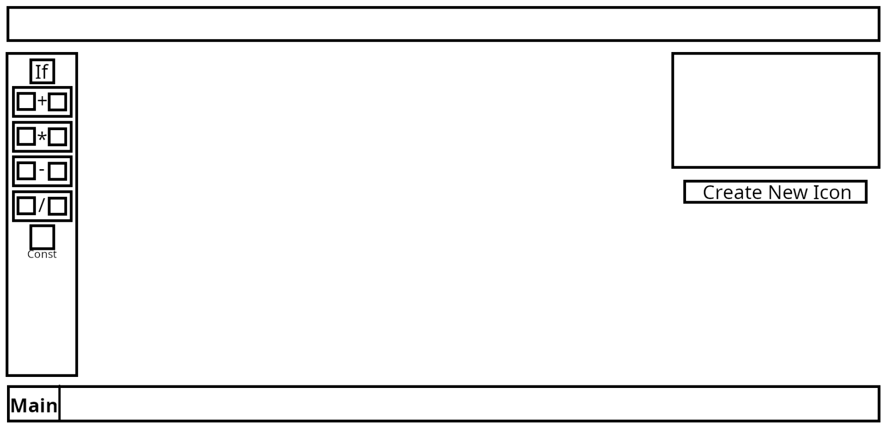


The default and custom icon menus will both be scrollable and collapsible, and the user will be able to drag icons from these menus to the screen to create duplicates of said icons.

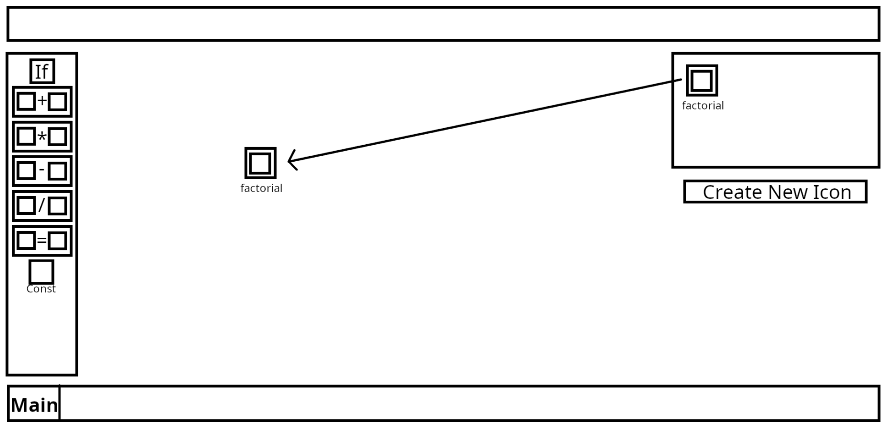
4 Usage example - Factorial

This section will demonstrate how an icon which computes the factorial of a number can be created.

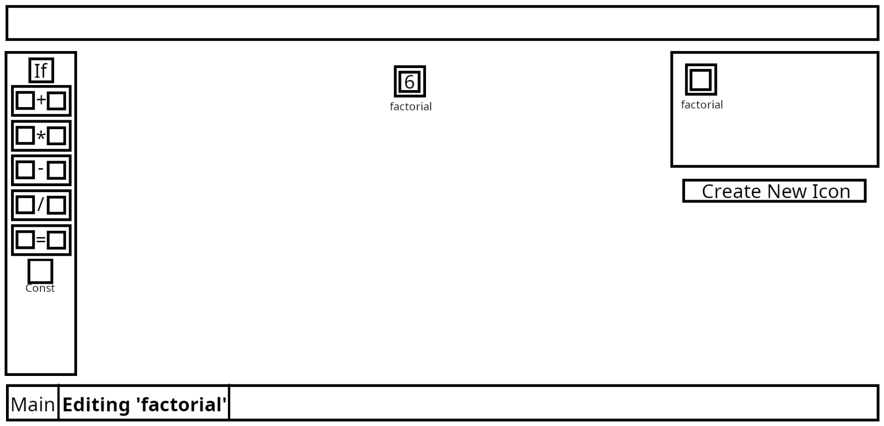
- We open the program and see the main screen



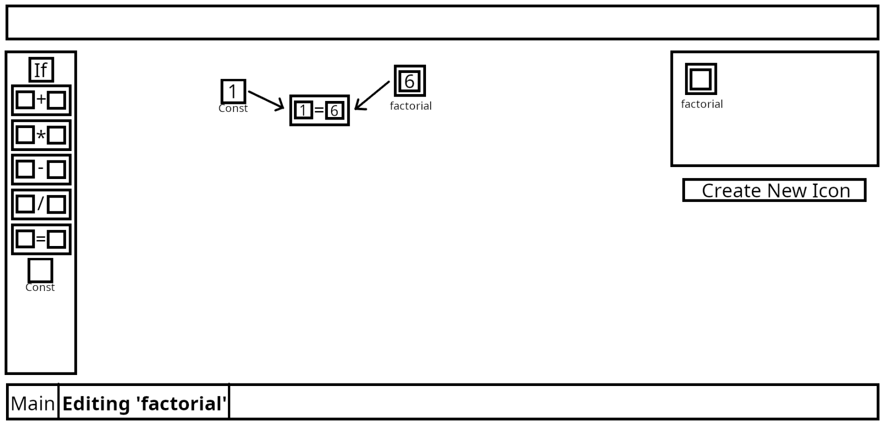
- We click the **Create New Icon** button below the custom icons menu.
- We name our icon 'factorial' and set the amount of its inputs to one. This creates a new empty icon, with said name, which is added to the custom icons menu. This icon can than be dragged onto the field.

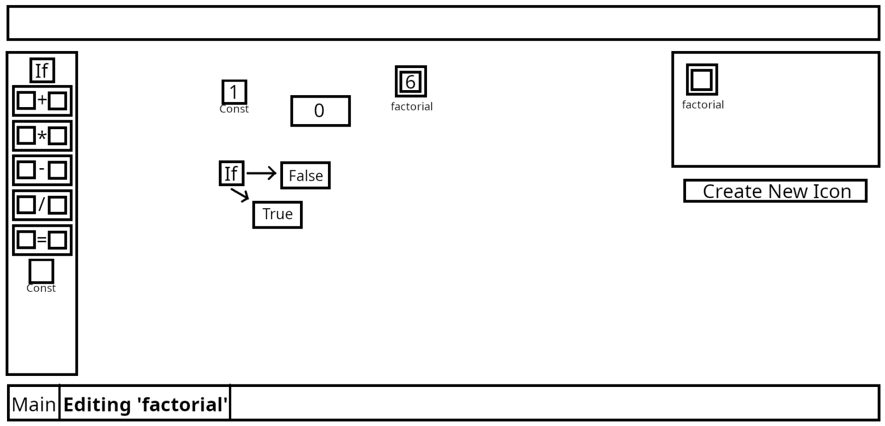


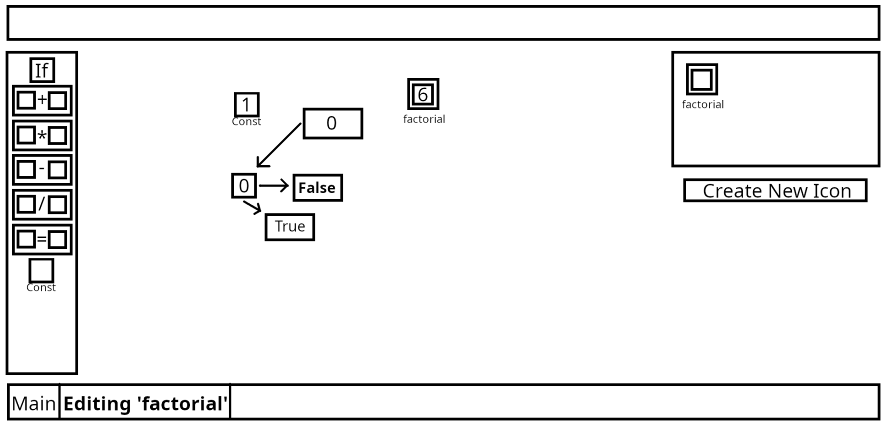
- With the icon on the field, we can fill its inputs with values, and then left-click on the icon to execute its code.
- Since the icon is new, we immediately hit a so called **trap**. This pauses execution and opens a new tab containing the **context** of the icon (which is mostly empty for new icons). This is where we program icons.



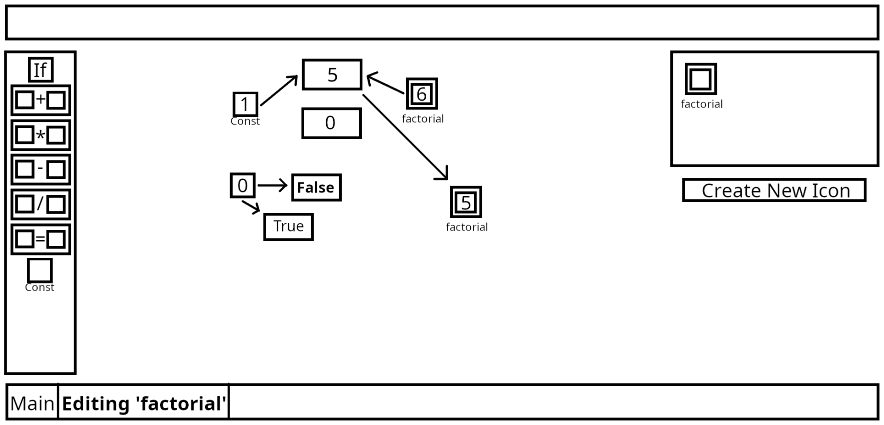
- The context of an icon reprents the internals of an icon. We can program an icon by dragging icons onto the screen, and dragging values between them. This builds up a tree of operations which will be executed anytime our new icon is ran.
- In the factorial, we need to use an **If** instruction to continue. We create an **IsEqual** icon and use it to check whether the input parameter of the factorial is equal to one. In our case, it is not, so the **False** branch is highlighted, which means that all following operations we do will only execute if the input is not equal to one.



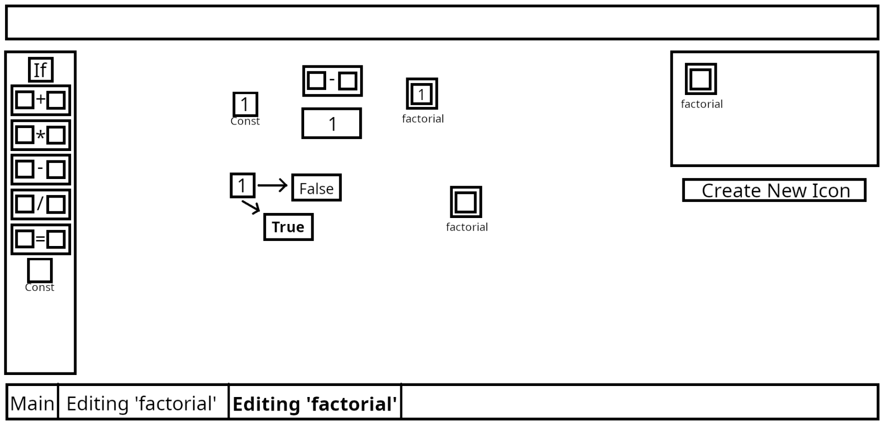


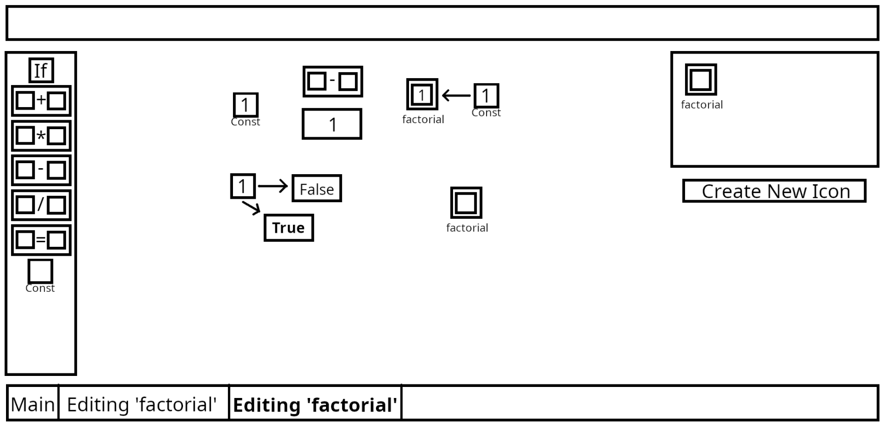


- We are writing factorial recursively, so we need to add a new factorial icon to our screen. We then subtract one from the input, and drag the result of this operation into the factorial icon. We then left click this icon to run it.

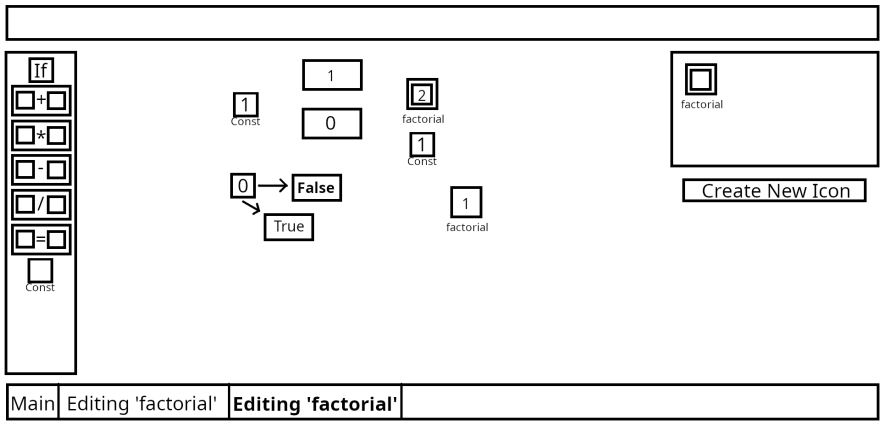


- Since we never defined what happens if we execute the **True** branch, our program hits a **Trap**, and it creates a new tab that contains our current status, which means the state where the input is equal to 1. We create a new constant block with the value 1, and drag it to the master icon (the first icon that was on the screen when we started programming the factorial). This signifies that this value is the output, so running the factorial icon with an input of 1 will now actually return a valid value.

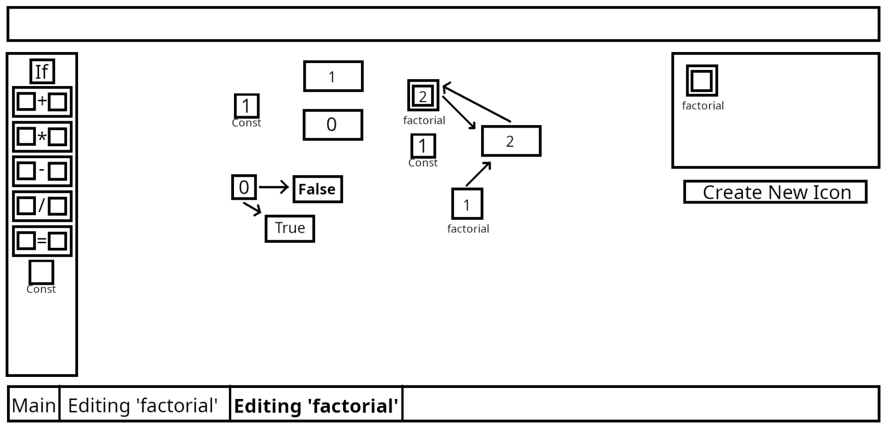




- Now we can close this tab and return to our first editing tab. We can see that the changes we made in the other tab carried over. We now can try evaluating the factorial again, which will hit another trap, but this time it does so because we don't know what to do after evaluating the factorial. We will be put into a new tab where the input is 2.



- In this tab, we can finally finish the method, by taking the output of the inner factorial, multiplying it with our input, and dragging the result into the output. We can now close all new tabs, and our factorial block can used.



5 Other requirements

Because of the fact that this project is a proof of concept, there are not many requirements placed on aspects of the project such as performance. The only requirement is that the software works on modern browsers, and is responsive.

6 Project restrictions

- The only supported type that can be used within the icons are integers. The project may be extended to support arbitrary types later on, but for now, this is a non-goal.
- The icons can only represent pure functions. There is no internal state that can be referenced. This means that icons representing data can only exist as functions returning a constant, and that creating icons which represent objects is impossible as of now.

7 Timeline & Milestones

Date	Milestone	Presentation method
14.05.2023	First version of documentation	Meeting with supervisor

Appendix A: Terminology