FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Adrián Habušta

Reconstructing the Pygmalion programming environment

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis:	Mgr. Tomáš Petříček, Ph.D.
Study programme:	Computer Science

Prague 2024

First of all, I would like to thank my supervisor for his patience and very helpful guidance throughout the process of writing this thesis. I would also like to thank my family and friends for their support and encouragement. Finally, I would like to thank my girlfriend for her love and understanding, and her help keeping me sane during all of this.

Title: Reconstructing the Pygmalion programming environment

Author: Adrián Habušta

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Tomáš Petříček, Ph.D., Department of Distributed and Dependable Systems

Abstract: This thesis focuses on reconstructing a subset of Pygmalion, a programming system that was revolutionary for its time, due to its use of programming by demonstration and iconic (visual) programming. With no current runnable version of the software available, reconstructing Pygmalion is crucial to preserve these unique ideas and allow users to interact with them. Our reconstruction focuses on one example from the original Pygmalion thesis, which loosely shows how to create a new icon capable of computing the factorial of an integer. We used an iterative methodology to learn more about the original system while creating two consecutive designs for our reconstruction of the system. Through this process, we gained deeper knowledge of Pygmalions features, and created software that fulfills our goal of recreating the factorial example.

Keywords: programming by demonstration, history of programming, pygmalion, programming environments

Název práce: Rekonstrukce programovacího prostředí Pygmalion

Autor: Adrián Habušta

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Tomáš Petříček, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Tato práce se zaměřuje na rekonstrukci podmnožiny Pygmalionu, programovacího systému, který byl ve své době revoluční díky použití programování pomocí demonstrací a ikonického (vizuálního) programování. Vzhledem k tomu, že v současnosti není k dispozici žádná spustitelná verze tohoto softwaru, je rekonstrukce Pygmalionu klíčová pro zachování těchto jedinečných myšlenek a umožnění interakce uživatelů s nimi. Naše rekonstrukce se zaměřuje na jeden příklad z původní práce, který přibližně ukazuje, jak vytvořit novou ikonu schopnou vypočítat faktoriál celého čísla. Použili jsme iterativní metodiku, abychom se dozvěděli více o původním systému a zároveň vytvořili dva po sobě jdoucí návrhy pro naši rekonstrukci systému. Tímto postupem jsme získali hlubší znalosti o vlastnostech Pygmalionu a vytvořili jsme software, který splňuje náš cíl - obnovit příklad faktoriálu.

Klíčová slova: programování demonstrací, historie programování, pygmalion, programovací prostředí

Contents

Introduction

Reconstructing programming systems from the past helps us see the history of modern programming paradigms, but it also serves as a way to explore ideas that were abandoned. One such system is Pygmalion, which was a programming system described in the 1970s that was "a computational extension of the brain's short term memory"[1]. The primary idea to facilitate this design was *Programming by Demonstration*. If a user of Pygmalion wanted to create a function capable of running some algorithm, they would have to do every step themselves on some concrete values. The system would record all of these steps and would be capable of playing them back later to run said algorithm. This would work even on different data than the ones the user used to build the algorithm initially. Programming by demonstration, according to the author, results in programmers creating bug-free programs [1]. The original paper describes use cases such as a data structure visualizer, a function computing the factorial of a number, a circuit simulator and a Smalltalk language interpreter.

The goal of this thesis is to reconstruct a small subset of Pygmalion, primarily focusing on the Programming by Demonstration aspect of the orignal system. Specifically, we will reconstruct the above mentioned example from the original thesis [1], that describes how to create a reusable function capable of computing the factorial of an integer. The reconstruction will not be a fully featured system, rather, we are trying to create a program that will help us understand how the original system might have worked, and allow users to experience the style of interaction presented in the original paper.

The main difficulty of the reconstruction is the limited amount of information available about Pygmalion. There are two major sources available. First is the original thesis [1] and second, a book that describes Programming by Demonstration systems called *Watch what I do: programming by demonstration*[2]. The book contains a chapter dedicated to Pygmalion that reiterates information from the original thesis and shows a slightly more detailed factorial example, but not much more than that. The full Smalltalk-72 source code of Pygmalion is available at the end of the original thesis. However, because Smalltalk-72 uses several non-ascii symbols for regular actions[3], the code cannot just be copied straight from the thesis and it would be necessary to fill in these symbols manually. Then there is the problem of actually running a Smalltalk-72 program. Because of these issues, we did not consider running the program as a viable option to understanding it.

This thesis describes two approaches to recreating Pygmalion. The first one was based on our initial understanding of the factorial example. The core of the design was interpreting user actions as operations that build a representation of the underlying program. This means that when a user performs an action, an internal representation of the program is modified. The internal representation is then used when the program is run. Although this appears as a natural implementation methodology, it suffers from a number of conceptual limitations that we will discuss. The second version was developed to address these limitations. It represents the underlying program as a sequence of user actions. Instead of modifying an internal program representation, whenever a user performs an action it is applied to immediate state and saved to a list of actions. When the program is then run, these actions are applied sequentially to a base state, which in practice recreates the computation of the program.

The thesis follows the following structure. In the first chapter, we will describe what software recreations are and their purpose. We will also describe Programming by Demonstration, as it is the underlying principle behind Pygmalion. In the second chapter, we will describe Pygmalion based on our understanding of it, and highlight some aspects that are lacking in explanation. The third and fourth chapters will describe the first and second designs explained above, respectively. Both chapters contain explanations of the underlying data structures, and a clarification of the designs respective core principles. The fifth and final chapter contain a comparison of our final design and the original system, highlighting the main similarities and differences between the two.

Chapter 1

Background

In this chapter we will start by touching upon the ideas behind reconstructing software and why software reconstruction is important. We will then describe the concept of programming by demonstration (PbD), and explore examples of this programming paradigm. Finally, we will discuss the libraries and tools that we have used to implement our reconstruction, along with the iterative methology we have followed.

1.1 Software reconstruction

Interlisp is a programming environment built around a version of the *Lisp* programming language. This system had many features for its time, but the one we will talk about is *Do-What-I-Mean*, or *DWIM* for short[4]. DWIM was a part of the Interlisp programming environment that automatically corrected common and easily fixable errors. This included errors such as a misspelling of a function or variable name, a missing paranthesis. These fixes were done straight to the source code, not just during runtime, and the user might not even know that DWIM fixed something.

In this example we can see that historical programming systems can contain interesting concepts and ideas that are rarely seen elsewhere. However, this leads us to a logical question. How many of these ideas have been lost to time? DWIM from Interlisp is a well documented example that still sees some use today. But there most likely exist countless systems and ideas that were never further explored. Some of these ideas might even be beneficial in modern programming systems. This is why we think that preserving programming systems (and software in general) is important. It prevents interesting ideas that may even prove beneficial from being forgotten, and allows us to play with these ideas ourselves.

Software reconstruction is a method of preserving software. The main idea behind reconstructing software is creating new software that closely mimics the *user and environemnt interactions* and potentially the UI design of the original software. However, we do not necessarily have to replicate implementation details such as the programming language used, the environment in which the software ran or the architecture of the original software. Doing these things would complicate running the software on modern hardware, which is one of the goals of software preservation.

We have found two primary methods of reconstructing software. The first is creating software that runs nearly identically. An example of such a reconstruction is Smalltalk Zoo [5]. Smalltalk Zoo is a website hosting a reconstructed Smalltalk environment along with many examples of software written in the early versions of Smalltalk. Most of the software on the site runs the same as it would when it was created.

Another way to achieve the goals of a reconstruction, atleast partially, is to create software that is similar to the orignal, but simpler. This can be done to highlight an interesting part of the software without having to deal with details that are not important to the demonstration. Petříček [6] created a reconstruction of the BASIC programming environment to showcase its unique ability to program, test, debug and run code at the same time. The author decided to not fully implement the feature set of BASIC, as that would not add anything to the feature the author was trying to showcase. This method is the one that we chose for our reconstruction of Pygmalion.

1.2 Programming by demonstration

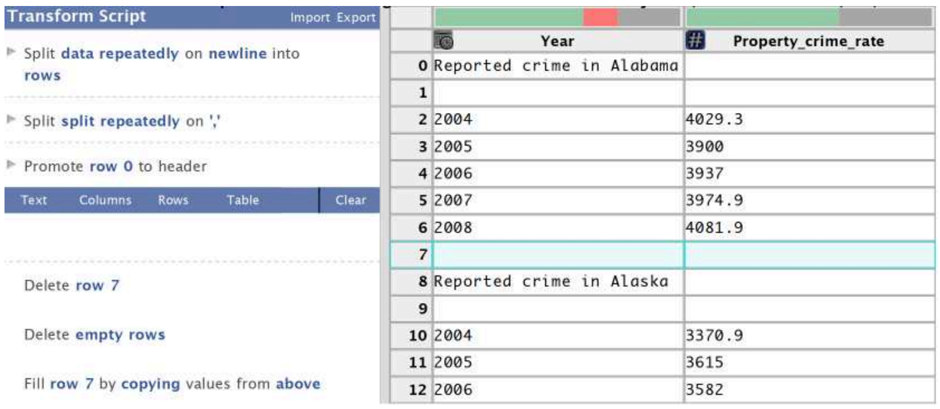
To explain the concept of *Programming by Demonstration*, it is useful to first review the widely accepted understanding of what program and programming are. The definition of a computer program is "[a] detailed plan or procedure for solving a problem with a computer; more specifically, an unambiguous, ordered sequence of computational instructions necessary to achieve such a solution." [7] If we look at programs as a "sequence of computational instructions", we can understand coding as someone writing down this sequence of instructions, while potentially using some abtractions. Abstractions are mechanisms that allow programmers to keep these sequences relatively short. The two most important aspects of abstractions are "removing detail to simplify and focus attention" and "the process of generalization to identify the common core or essence". [8]

Computers understand and operate on simple instructions. Programming languages can be viewed as abstractions over said instructions. A natural next step is to create abstractions over programming languages. This means creating methods that allow development using little to no code. Today, we call these methods low-code or no-code development, and they are slowly but surely rising in popularity [9, 10].

The no-code approach to programming that we will focus on appeared from a simple question: What if we could **perform** the actions a program does, instead of writing them down beforehand using code? This is the main idea behind programming by demonstration, or *PbD* for short.

1.2.1 Contemporary example

Programming by demonstration is a concept still in use today, as showcased by Wrangler. Wrangler is software used for data analysis that implements a programming by demonstration component capable of creating/reusing data transformations [11].


Transform Script   Import Export

- Split data repeatedly on newline into rows
- Split split repeatedly on ','
- Promote row 0 to header

Text   Columns   Rows   Table   Clear

Delete row 7

Delete empty rows

Fill row 7 by copying values from above

	Year	Property_crime_rate
0	Reported crime in Alabama	
1		
2	2004	4029.3
3	2005	3900
4	2006	3937
5	2007	3974.9
6	2008	4081.9
7		
8	Reported crime in Alaska	
9		
10	2004	3370.9
11	2005	3615
12	2006	3582

Figure 1.1 Wrangler interface. On the left is a menu showing previously applied transforms and suggestions on what transforms to apply to the selection [11].

The main method of interaction with wrangler is applying the suggested simple transforms, or performing some ourselves, which builds a reusable script for applying the same compound transform later. The user can use the suggested operations to transform the data into various formats. The transforms can include things like removing a row, removing all empty rows, extracting common text into a new column, etc.

An example in "Wrangler: Interactive visual specification of data transformation scripts" [11] is used to clean up data of housing crime in the U.S. First, the user imports a CSV table with the data into the program. Wrangler then recognises the format, and automatically splits it into rows and columns. The user then selects an empty row and uses a suggestion to delete all empty rows in the document. Then, the user uses another suggestion to extract names of states from text and creates a new column for them. The rows with the original text are then deleted. All of these actions combined then create a script that can be run on any CSV file and does these same steps.

The main benefit of this approach to editing data is the fact that all changes show a preview of what will happen. This allows the user to check whether the action they are about the perform is actually what they wanted, and helps speed up the editing task.

1.2.2 Historical example

The book *Watch what I do: programming by demonstration* [2] compiles the most prominent PbD systems that were created up until 1993. The first of these examples is *Pygmalion*, which is the focus on this thesis. We will discuss Pygmalion later on. Other examples include systems such as *Metamouse* or *TELS*. We will discuss Metamouse next.

Metamouse

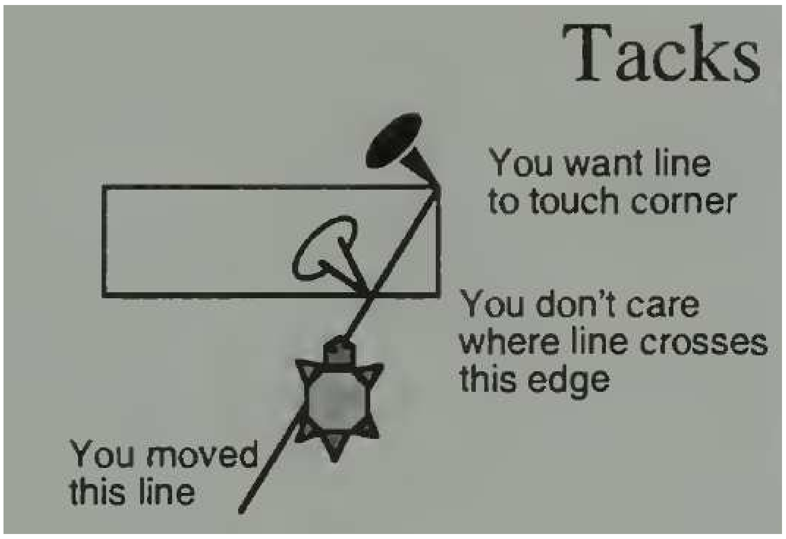

Tacks
You want line to touch corner
You don't care where line crosses this edge
You moved this line

Figure 1.2 Excerpt from the Metamouse manual [2].

Metamouse is a prototype drawing program with a ”helper” called Basil. The original version of the program only supports drawing lines and boxes. The main purpose of Basil is the user teaching (showing) it how to perform a certain task. Basil would then ask the user if it could take over once it detects that an action it knows is being performed. Showing Basil how do do something is done by performing actions on the screen, such as moving a box or drawing a line. This creates ”tacks” on the intersections between objects, and the user can show Basil which of these intersections are important to preserve and which are concidental. The user can draw new shapes as helpers to make relationships more explicit. For example, a line can be drawn to signify how two points on different object must be placed in relation to eachother. The original article calls shapes like these ”tools” [12].

In an example from *Watch what I do: programming by demonstration* [2], the program was used to teach Basil how to sort boxes of differing heights, provided their base is on the same line. The user could do that by first drawing a spacer and a sweep line. The sweep line would be moved to the top of the lowest block, and the intersections with said block would be marked as important. Then, Basil would be shown how to use the spacer to place a block next to the last sorted one. When the user moves the sweep line to the second lowest block, Basil asks to take over, because he understands what to do now. If the user lets Basil take over, the boxes are then sorted one by one until there are no boxes left. The created procedure would then sort boxes of any height and count.

In some cases, Basil can incorrectly understand what the users intentions are. In this case, the user can undo what Basil did, and show it what it was actually expected to do [12].

1.2.3 Conclusions

As we could see from the examples above, most programming by demonstration systems operate on one core design principle. They act as a tool to simplify interactions within a single problem domain. This is understandable, as implementing a useful programming by demonstration system is a challenge in and of itself. This, however, is why Pygmalion is so interesting. The author imagined it as a generic programming tool using programming by demonstration. While examples such as Wrangler or TELS operated on spreadsheets and text respectively, Pygmalion allows us to interact with and create programs. This is a powerful idea. A generic programming by demonstration system could be made to support operations on spreadsheets or text as well. Pygmalion is not advanced or user friendly enough to achieve this, but it’s an interesting starting point for such an idea.

1.3 Our implementation

We chose to implement our reconstruction as a web application. The main reason for this are easy access, and programming simplicity.

Putting the application on a website makes it more accessible to the user. When we upload the program on the internet, accessing it is possible from any device with a modern web browser. If we were to create a computer application, the user would either have to compile it themselves from source, or we would have to publish pre-compiled binaries for multiple systems. This way, we can just share the URL to the application and anyone can access it.

The application is hosted on GitHub Pages, and is configured to automatically recompile and update on every push to the underlying repository[1].

1.3.1 Programming language choice

Our program is coded using the F# language. We also used CSS to help build the UI.

The first language that we considered was JavaScript. JavaScript is a weakly typed object-oriented lanugage, that is natively implemented in modern web browsers. This makes it a great fit for web development. However, because of our personal preferences and experience, we decided that we would prefer to write the software using a strongly typed language.

Our second pick was Haskell, a purely functional programming language. Due to the nature of functional programming languages, creating our program in Haskell would enable us to easily create understandable and maintainable code. The fact that Haskell is a purely functional language is also a downside in some cases. We expected that our project would be easier if we allowed mutability or an object-oriented approach in some places. This would be impossible to do with Haskell. That, combined with our unfamiliarity with the web development scene of Haskell, we decided to search for another language.

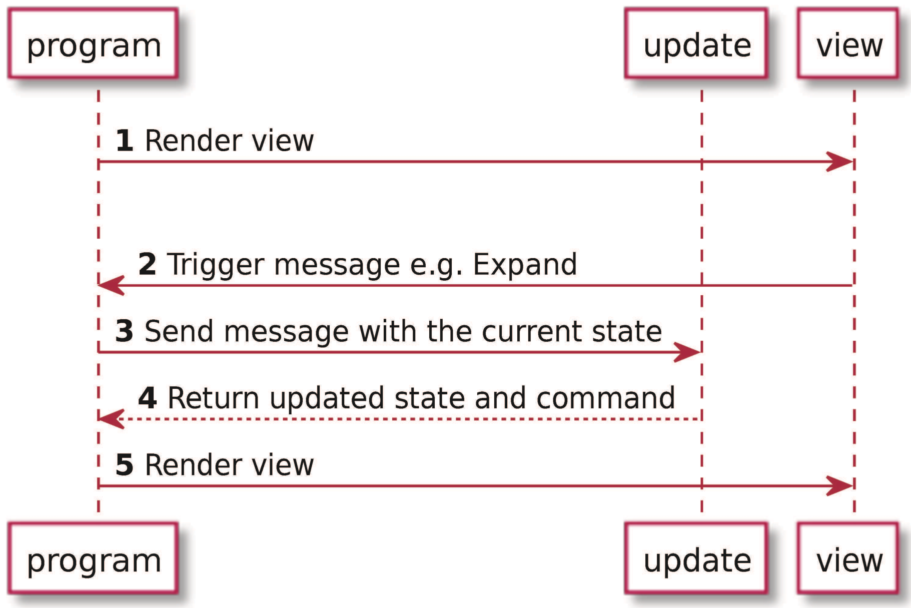
Our final pick was F#, a primarily functional programming language running on .NET, that also supports an object-oriented programming style. Similarly to Haskell, the functional programming aspects of the language allow us to easily create readable, maintainable code, while also allowing mutable variables and object-oriented design principles where necessary. The deciding factor for our decision was the fact that a transpiler from F# to JavaScript exists. The project is called Fable [13]. Since we can transpile our F# code into JavaScript, we can make use of the extensive suite of development tools that exist for JavaScript projects.

1.3.2 Tools and libraries

We already mentioned Fable, a program that can transpile F# code into various languages, primarily JavaScript [13]. There is a number of different libraries that simplify web development that were used in this project.

[1]`https://github.com/A-Habusta/Pygmalion-Like-Reimplementation`

Elmish



program
update
view
1 Render view
2 Trigger message e.g. Expand
3 Send message with the current state
4 Return updated state and command
5 Render view
program
update
view

Figure 1.3 The flow of data in an Elmish application [14].

To go along with Fable, we used the Elmish library for our program structure. The library implements a *model, view, update* design pattern that is inspired by the Elm programming language. The basic idea of the library are three functions: *update*, *view* and *init*. The init function creates an initial state. The view function is responsible for drawing the current state to the screen. It receives a function called *dispatch* as a parameter. This function is supplied by the library. The function can then be called by UI elements to signal to the program that something is happening. When a message is dispatched, it is passed to the update function, along with the current state. This function takes care of updating the state based on the message that was dispatched [14].

We chose this library because the model described above is a good fit for simple interactive web applications that are built around an immutable representation of the application state.

Aether

Aether is an F# library that adds *optics* to the language [15]. While there are multiple concepts that can be categorized as optics, we will only talk about *lenses* and *prisms*, since those are the optics that we used the most. According to Steckermeier [16], “lenses in functional programming offer a way of focusing on

```
// Create simple data types
type Address = {
    Street: string
    City: string
    PostalCode: string
}
type Person = {
    Name: string
    Address: Address
}

// Create simple lenses
let personAddressLens =
    (fun p -> p.Address),
    (fun address p -> { p with Address = address })
let addressCityLens =
    (fun a -> a.City),
    (fun city a -> { a with City = city })

// Create a composed lens used to modify the nested City
// field within a Person instance
let personCityLens =
    Lens.compose personAddressLens addressCityLens

// Create a modified copy of person using the composed lens
let updatedPerson = Lens.set personCityLens "New City" person
```

Figure 1.4 Simple use case of lenses.

a particular part of a, possibly nested, data structure or container." This means that lenses allow us to work with a specific part of a data structre. Lenses are also composable, meaning that we can create a lens that looks at a member of a nested data structure. The usage of lenses is illustrated in figure 1.4.

In our system, we use the term Prism to refer to something akin to a lens but capable of handling the potential unavailability of the data its pointing to. An example of this is accessing the n-th index in a list. If the list has more than $n + 1$ members then the access works fine, but if it has less than or equal to n members, the access would fail. Prisms deal with this by returning the accessed type wrapped in the *Option* type. An `Option<T>` type has two states. It can either contain a value of type `T`, or `None`[2]. In the case of the above mentioned example, accessing a list index outside of the range of the list using a prism would return `None`, while a valid access would return the normal value wrapped in `Some`.

[2]This is similar to `std::optional` in C++ or nullable types in C#

Feliz

Feliz is another F# library. It provides a DSL for simple, type safe creation of React elements inside F# [17]. We used this library to build the basic structure of our application. Fable provides its own library for this, but the official repository of the Fable project mentions that users should prefer to use Feliz instead [18].

Vite

Vite is a web development build tool that bundles web applications for production [19]. It is also capable of running a dev-server and hot reloading the application on file changes. This was very useful during development, as we could see the changes we made to the code in real time. We also use Vite to bundle the transpiled F# code into a single JavaScript file before it is uploaded to the project website.

GitHub

We used GitHub for version control of both the software and this thesis. We also used GitHub Pages to host the application, since it is static and does not require a server to run. Lastly, we used GitHub Actions to automatically recompile and update the published application on every push to the repository. The same was done for the thesis.

1.3.3 Methodology

One of the goals of our application was to understand how Pygmalion could have been implemented. To achieve this, we followed an iterative development process. This meant that we would implement a design, see what we learned from it, and then transfer our knowledge to the next design. We only created two designs from this process, as we felt that we had learned enough for our purposes.

Chapter 2

Pygmalion

Pygmalion is a visual programming system created in the year 1975 by David Canfield Smith. The main goal of Pygmalion was to “develop a system whose representational and processing facilities correspond to and assist the mental processes that occur during creative thought.” [1] The author wanted Pygmalion to serve as an intuitive translation medium between a users mind and the computer.

This chapter serves as the main introduction of the Pygmalion programming system. We will discuss a high level overview of the system and then move on to aspects of the design that are unclear from the original document [1].

2.1 Introduction

According to Smith [1], Pygmalion is ”a two-dimensional, visual programming system implemented on an interactive computer with graphics display. Communication between human being and computer is by means of visual entities called ’icons’, subsuming the notions of ’variable’, ’reference’, ’data structure’, ’function’ and ’picture’. The heart of the system is an interactive ’remembering’ editor for icons, which executes and (optionally) saves operations for later re-execution.”

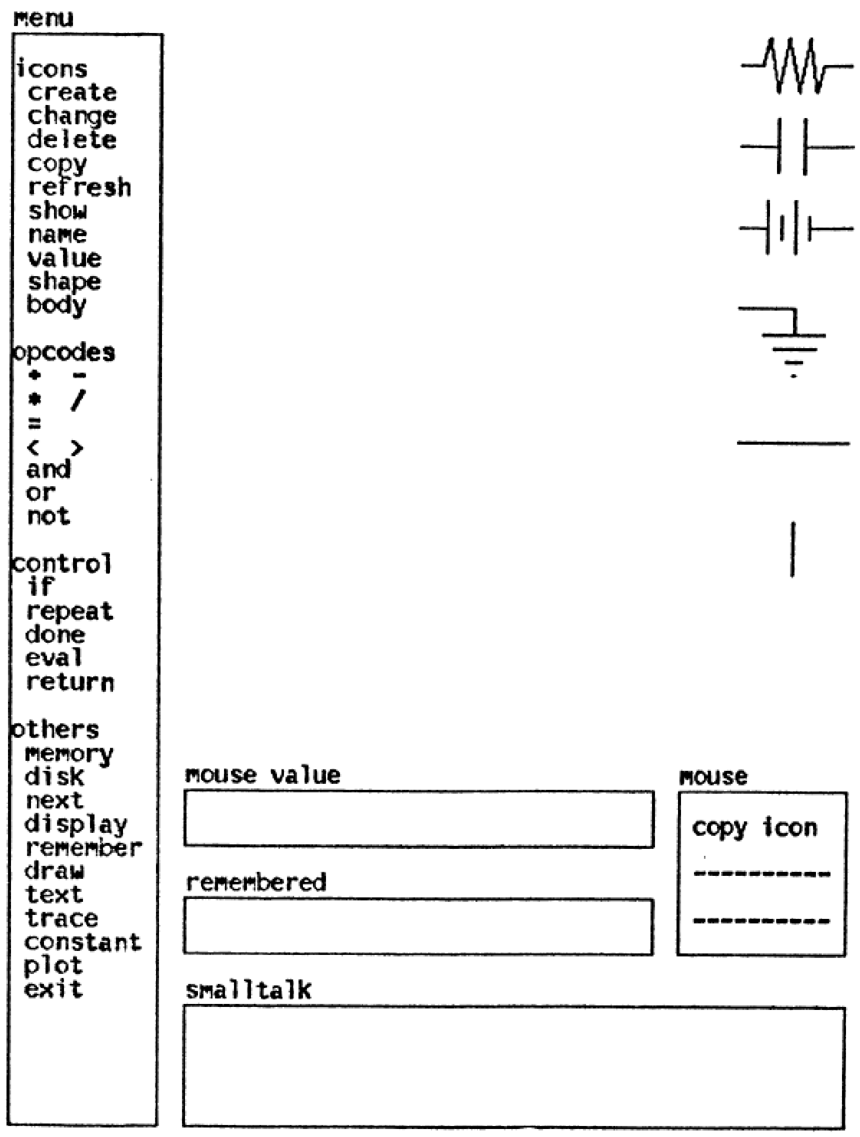


Figure 2.1 The Pygmalion programming environment [1].

Pygmalion was programmed using Smalltalk 72 [1]. Smalltalk 72 is an object-oriented language, that uses ”messages” to communicate between instances of classes. These messages are passed around using strings of characters. The messages that a class can respond to define the semantics of said class [3]. At the time of writing this thesis, Smalltalk-72 is only usable through emulators. An example of such an emulator is Smalltalk Zoo [5]. Even though we have the full source code of Pygmalion available through a scan, it cannot be easily run in such an emulator. This is because there is no way to easily copy the source code from the original thesis, due to special Smalltalk symbols.

2.1.1 Icons

The main focal point of Pygmalion are icons. Like mentioned in the earlier citation, icons can represent variables, data structures and even functions. The user can manipulate icons by selecting various menu entries and then clicking on the canvas or on icons. This menu can be seen in figure 2.1. The entries allow the user to move icons, create new ones, define their shape, etc. Icons can also be nested within one another. This can be used to represent nested data structures, or function parameters, function results, etc.

Icons can be thought of as objects that have multiple attributes. The interesting ones are `SHAPE` and `BODY`. Both of these hold a list of actions that the icon can perfrom. This list is called the *code list*. While `SHAPE` and `BODY` are functionally identical[1], they should be used for different purposes. As their names suggest, the `SHAPE` attribute should be used for drawing the visual representation of the icon. The `BODY` attribute should be used for the actual functionality of the icon, if there is any. When an icon is evaluated, the body is executed by replaying all the actions in the body list. Similarly, when an icon is drawn, the shape is drawn by replaying all the actions in the shape list.

Other attributes are expected things such as `X` and `Y` for the position of the icon on the canvas, `NAME` for the name of the icon, etc.

2.1.2 Modes of operations

Pygmalion has two modes of operation. The first is the *display* mode, and the second is the *remembering* mode. These modes seem to be mutually exclusive, but it also seems that we don't have to have either of them turned on.

Display mode

This mode is not very well explained. According to Smith, "[d]isplay mode provides a means for communicating the semantics of operations visually." [1] and that "[i]t is also turned on when an iconic trace of a program is desired." From these descriptions, we can assume that display mode can be used for a visual replay of an operation. The mode can also be turned off to gain execution speed [1].

[1]The author specifies that the `SHAPE` attribute may be used as another function body for an icon, but does not explain how.

Remember mode

This mode is the used for defining the `BODY` or `SHAPE` attributes of an icon. When remember mode is turned on, most, if not all, actions that the users perform are saved in the correct code list of a specific icon. Remember mode can then be turned off using the `done` operation, or by pressin a "Stop remembering" button.

To enter remember mode, the user either has to explicitly turn it on, or the system will automatically turn it on when a trap is hit. Traps are special actions that are saved at the end of unfinished code lists. They represent a branch of the code list that is not yet defined fully. When they are executed, remember mode is turned on and the screen is cleared of all icons that are not part of the icon that hit the trap. This allows us to continue defining the unfinished branch.

2.1.3 Pre-defined operations

Some of the entries in the menu, specifically the ones under the fields `opcodes` and `control` allow the user to put pre-defined icons on the screen.

Opcodes

The opcodes are simple operations, such as basic arithmetic. Internally within Pygmalion, evaluating one of these pre-defined icons sends a Smalltalk message to the first parameter, containing the textual representation of the opcode and the second parameter. The operation that is then executed depends on the response to the message defined in the first parameter. For example, this allows the + operation function as addition for integers and as concatenation for strings. The author however, does not specify how custom Smalltalk objects could be created for reusing these default opcodes with different underlying operations.

Control

The control tab of the menu contains operations that affect the flow of the program. Only two of the entries in this tab are icons, specifically the `if` and the `repeat` entries. Both icons can be seen in figure 2.2 and figure 2.3, respectively. We will discuss these icon in more detail in later sections.

Other operation within this menu are not icons, but rather special operations. The `eval` operation executes the `BODY` of an icon. The `done` operation finishes the definition of an icons `BODY` while in remembering mode. The other operations are not explained in the original document.

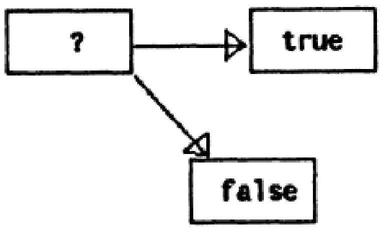


Figure 2.2 ’If’ icon in Pygmalion [1].

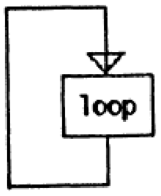


Figure 2.3 ’Repeat’ icon in Pygmalion [1].

2.2 Factorial example

The example that is best explained in the original thesis is the factorial example. We used this example as the basis for our reconstruction.

The example starts off by creating a new icon, which will represent the factorial function, with one subicon that serves as the parameter of the function. A 6 is placed within the parameter subicon and the icon is evaluated. This hits a trap within the icons body, which causes the program to automatically turn on remembering mode, and hide everything that is not part of the icon. The author then starts defining the factorial by creating the necessary icons and using them for the calculation. One of these icons is the `If` icon, which is used to check if we are evaluating the factorial of 1. This icon is not very well explained, so we will discuss it later. A part of this process can be seen in figure 2.4. An important step in this process is the usage of a recursive factorial icon. The author creates another factorial icon, places the original parameter minus 1 into its parameter box and evaluates it. The definition if this recursve factorial is skipped, and we only see the result of the evaluation being placed into a multiplication icon. This can be seen in figure 2.5. Our understanding of this operation will be explained later. Afterwards, the result of the multiplication is placed into the original icon and the calculation is finished.

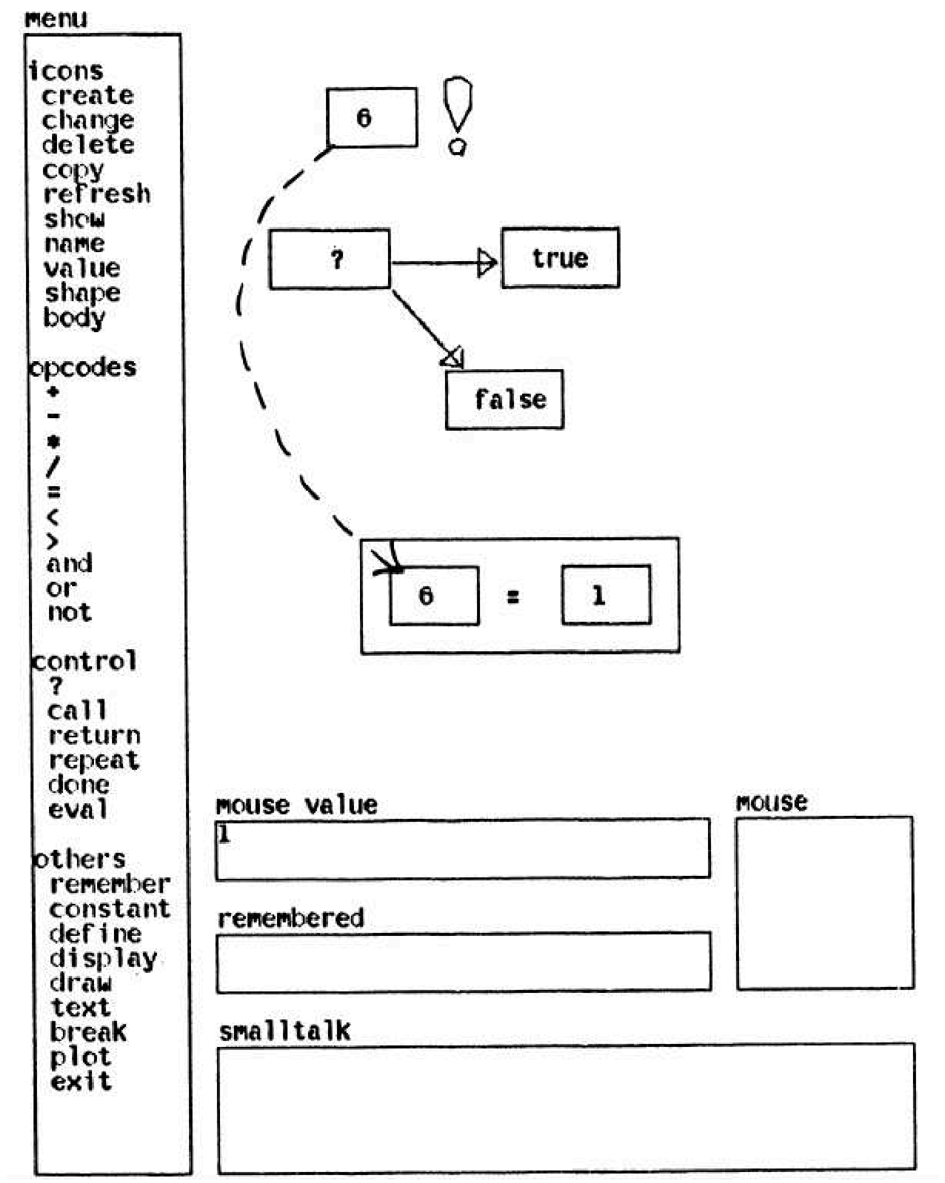


Figure 2.4 One step of the definition of factorial, placing the parameter of the factorial into another icon. Notice the dashed arrow that represents picking up a value and placing it elsewhere.

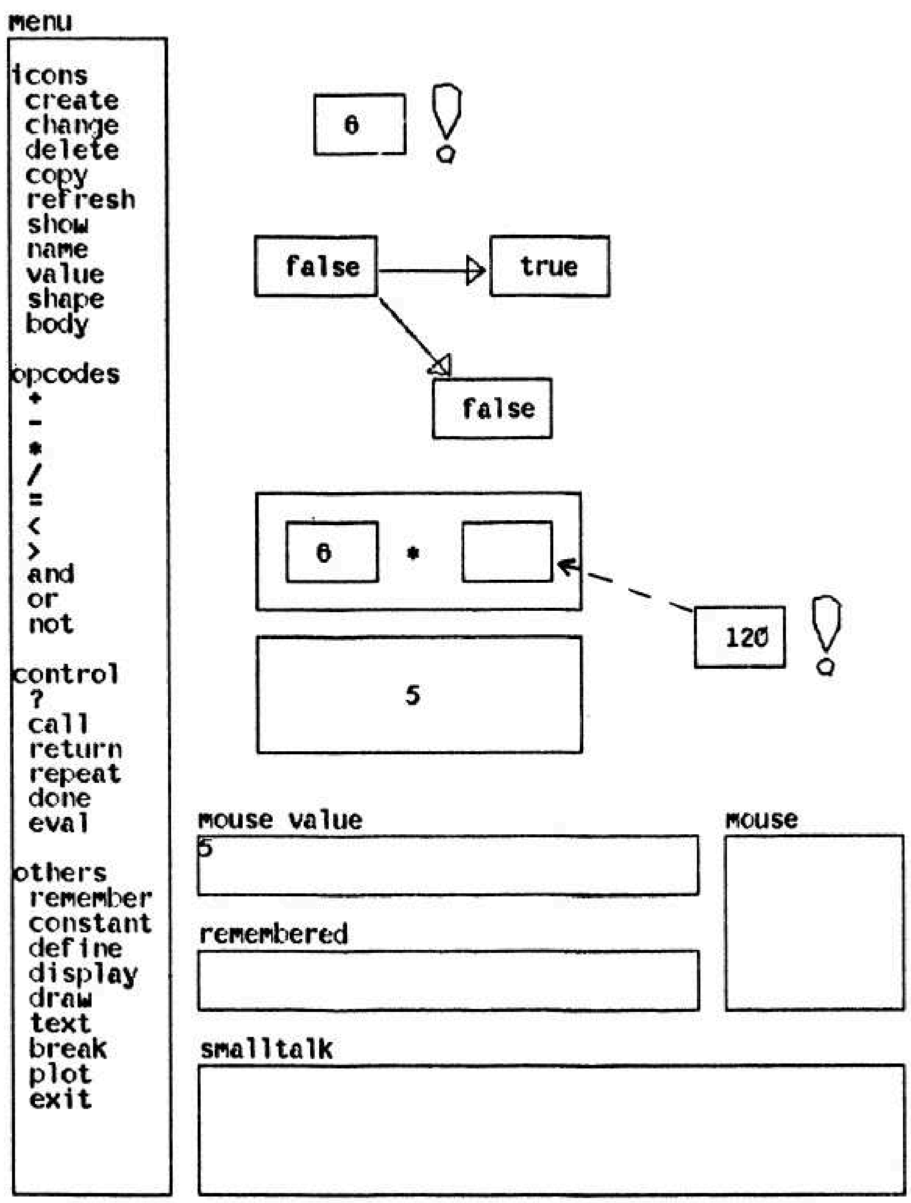


Figure 2.5 Another step of the definition of factorial, taking the result of the nested call and placing in in the multipliction icon.

Issues

There are some aspects of this that are not well explained in the original thesis. The first is the creation of the nested factorial icon. In the example images, the icon just appears on the screen. According to the author, creation of multiple identical icons is done using the `COPY` operation. The copy operation copies by value and not by reference, so the nested factorial would have its own body and shape attributes. This would mean that defining the nested factorial would not affect the original factorial icon. Another issue that arises is that the copied factorial icon will only contain the actions that were performed up until the copy itself. With these two problems combined, we would basically have to define every single recursive call of the factorial function until we hit the base case of 1, which in our opinion is not feasable.

In *Watch what I do: programming by demonstration* [2], specifically in the chapter that discusses Pygmalion, the author used the `CALL` operation to create the nested factorial icon. This operation is not explained in the original thesis nor in the book where it was used, so it is not clear how it works. We can only assume that is somehow creates a new icon by reference, not by value.

If we assume that the definition of the nested factorial also defines the original factorial, we run into another problem. The screen shown in figure 2.5 shows the result of the nested factorial being placed into the multiplication icon. This would not make sense if the nested factorial shared the body with the main factorial. Since the nested factorial is already evaluated, that means that its body already has the necessary actions to evaluate a factorial. This would then mean that even our main icon should already have these actions, so we would not need to repeat them.

In the following subsections, we will discuss some of the issues that we encountered when trying to understand the Pygmalion system in more detail.

Factorial base case

We will assume that there is a way to have nested icons share the `BODY` attribute with their parent. We will also assume that step show in figure 2.5 is just for demonstration purposes. With these two assumptions, we can now discuss our understanding of how the factorial of 5 is defined in the original example.

When the factorial of 1 is evaluated, the `If` icon will evaluate to true, and the code list for a different branch will be executed. Since this would be the first time that this branch is evaluated, we can assume that it only contains a trap. When this trap is hit, the program shows us the state of the screen as it was when the trap was hit.

This would only contain the comparison and `If` icons, both evaluated to

True. We then just have to place 1 into the result of the factorial icon, and finish the definition. Since the factorial of 1 is now defined, we can assume that the execution continues. The next trap will be hit in the case of the factorial of 2, since that was where the factorial of 1 was evaluated. We will again see the state of the screen as it was when the trap was hit, which will be similar to figure 2.5, but with the parameter being 2 and the result of the factorial being 1. We can then place the result of the nested factorial into the multiplication icon and finish the definition of the factorial. Since the factorial is now fully defined, we can assume that the execution continues until the factorial of 6 is evaluated and the final result is shown.

A similar example of this can be seen in *Watch what I do: programming by demonstration* [2], where the steps the author performs are very similar to the ones we specifed here.

Alternative factorial base case

If our assumptions are wrong, and the nested factorial does not share the body with the main factorial, we would have to define each nested factorial separately. Our first steps would be the same as they were in the section above. When we create the nested factorial icon, it will copy the `BODY` from the original, which includes all the actions up until the creation of the nested factorial. We fill in the parameter of the nested factorial and evaluate it. This will hit a trap at the point where the copied body ends, which is right before we created the nested factorial. We then would have to repeat the steps of creating the nested factorial, filling in the parameter and evaluating it. This would happen with every nested factorial until 1. The factorial of 1 just requires us to place 1 into the result of the factorial icon and finish the definition. We would then have to go back to the factorial of 2 and place the result of the nested factorial into the multiplication icon. This would have to be repeated for every nested factorial until we reach the factorial of 6 and finish its definition.

Subsequent evaluations of this icon with values greater than 1 would mostly work. The `BODY` of the factorial icon now contains all the actions that copy a new factorial icon, evaluate it and multiply its result with the parameter. However, the original factorial icon does not know about the actions that were performed in the nested factorials, so it does not know what to do when a 1 is encountered. Even if we define the steps for this case again after hitting a trap, they would not be saved within the original factorial icon. The only way to fix this would be to run the original icon with a 1 as a parameter and then finish defining it that way.

Because of these issue, we think that our assumptions from the previous sections are more likely to be correct, and that the definition of the factorial that we described in the previous section is the correct one.

If icon

The `If` icon is only explained briefly, but we can assume that it contains two branches of actions, one for when the condition is true and one for when it is false. The paramter that is evaluated is a boolean. When the `If` icon is evaluated, the corresponding branch is chosen, and any further actions are appended to this branch. The Pygmalion example from *Watch what I do: programming by demonstration* [2] shows an updated version of the `If`, which better highlights which branch is currently chosen. This feature is not present in the original system. We have to read the parameter of the `If` icon to know which branch is currently chosen.

2.3 Other examples

The original document shows four potential use cases of Pygmalion. One of them was the factorial, which we discussed in section 2.2. There are three other examples that were shown in the original document. The first was a circuit simulator. The other two were a Smalltalk-72 Interpreter and a LISP 70 memory organization visualizer. These examples are less well-documented then the factorial example, and they are out of scope of our thesis, so we will not delve too deep into their explanation.

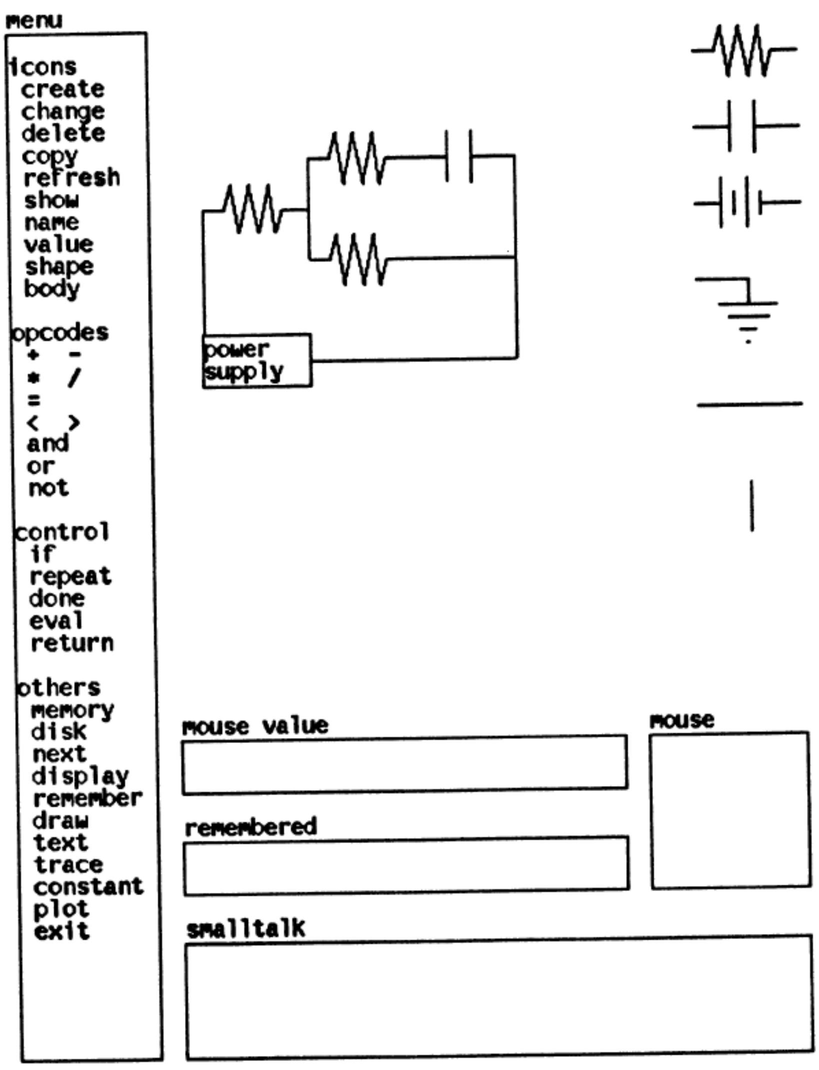


Figure 2.6 A circuit simulator/visualizer within Pygmalion, notice the base icons on the right. [1].

This example shows a circuit simulation being made. It can be seen in figure 2.6. The user starts off with some custom icons representing simple components of a circuit, such as a power source, a resistor, a capacitor and multiple types of traces. The user then copies these icons multiple times and arranges them to create a simple circuit. What is vague about this example is whether these icons actually form a circuit that can be simulated, or are just used to create a sketch of circuit. There is no detailed explanation on how these icons might communicate with one another.

The rest of the examples mentioned in the thesis are even less well-explained, and we will not talk about them in detail.

Chapter 3

First System Design

In the previous chapter, we described the Pygmalion system and its main features. We also saw that there is a very limited amount of information about Pygmalions actual implementation. Because of this, the first design was mostly used for exploration of the system and how it might have worked internally. We later abandoned the first design in favor of a different one.

Building upon the knowledge of Pygmalion from the second chapter, we will first explain the terminology that we use and the main idea behind our representation. Then, we will move on to data structures used to represent a program in our system and how the program is evaluated. This will be illustrated by using an example program that computes the absolute value of an integer. Finally, we will discuss the various issues around this implementation and why we ultimately decided to move on to a different design.

Our reconstruction was created with the goal of allowing users to experience a small subset of the proposed Pygmalion use cases. The choice we ended up with is an example showing how to create a custom operation computing the factorial of an integer [1]. This resulted in both designs only containing functionalities necessary for reproducing said example, meaning that we only support using operations with integer parameters.

3.1 Important terms

An *icon* is a single object that is drawn on the screen. It takes some parameters and can perform an operation on them. This operation is set when the icon is created. It can be a simple mathematical operation, or something more complex, like an if branching operation, or calling a user-defined operation.

We call these user-defined operations *custom operations.* Custom operations can be thought of as functions in a traditional programming language. They take some parameters and return a value. Custom operations are defined by the user. The user can use icons to perform various simple and custom operations. The results of these operations can be used in the evaluation of further icons, leading to a complete definiton of a new operation. The icons, together with the connections between them, are stored within the custom operation, forming a tree-like structure. This tree-like structure defines the custom operation.

That is the main idea behind the first design. The tree-like structure that we build could be considered an expression. When we evaluate this expression, we end up with the same result as the one the user ended up with when they initially defined the behaviour of a custom operation (if we use the same inputs). This metaphor will be explored in more detail later.

We will explain the implementation of the first design in the following sections. We will use `F#` types, because they offer a clear understanding of how the data is structured.

3.2 Data Structures

The most basic type in our representation is the *IconOperationParameter* type, or *IOP* for short. It represents a single parameter of an icon. It has four possible values:

- *Trap* - This value represents a parameter which is empty or not yet defined. The name is taken straight from the original system.
- *Constant* - This value represents a constant integer value.
- *BaseIconParameter* - Represents a reference to a parameter of the custom operation that the icon is a part of.
- *LocalIconReference* - Represents a reference to another icon within the same custom operation.

The next important type is the *IconOperation* type. It represents the operation that an icon performs. The parameters for each kind of operation are stored using the necessary amount of instances of the `IOP` type. The `IconOperation` type also has four possible values:

- *Unary* - Represents a unary operation, like negation.
- *Binary* - Represents a binary operation, like addition.
- *If* - Represents an if branching operation. It only has one parameter, and the reason for this will be explained later.

```
type IconTable = Map<IconID, Icon>

type CustomOperationType =
    { ParameterCount : int
      SavedIcons : IconTable
      EntryPointIcon : IconID option}

type CustomOperations = Map<string, CustomOperationType>

type IconResultsTable = Map<IconID, int>
type Icon =
    { Operation : IconOperation
      X : int
      Y : int }

type IOP =
    | Trap
    | Constant of int
    | BaseIconParameter of int
    | LocalIconReference of IconID
// Shortened type name for brevity
type IconOperationParameter = IOP

type IconOperation =
    | Unary of operator : string * IOP
    | Binary of operator : string * IOP * IOP
    | If of IOP
    | CallCustomOperation of customOpName : string * IOP list
```

Figure 3.1 Types used for representing icons and custom operations

- *CallCustomOperation* - Represents a call to a custom operation. The first parameter is the name of the custom operation, and the second is a list of parameters.

Following this type in the hierarchy is the *Icon* type. Its only purpose is to wrap an IconOperation along with information necessary for drawing the icon to the screen. Since most of the usefulness of this type comes from the IconOperation contained within, we won't spend much time on it.

The last type we will talk about is the *CustomOperation* type. It represents, as the name suggests, a custom operation. It only contains three fields. The number of parameters the custom operation requires, a map of all the contained icons , and the ID of the entry point icon. The entry point icon is the icon whose result is treated as the result of the entire custom operation evaluation. Custom-Operation instances are stored within the program state inside a map, where

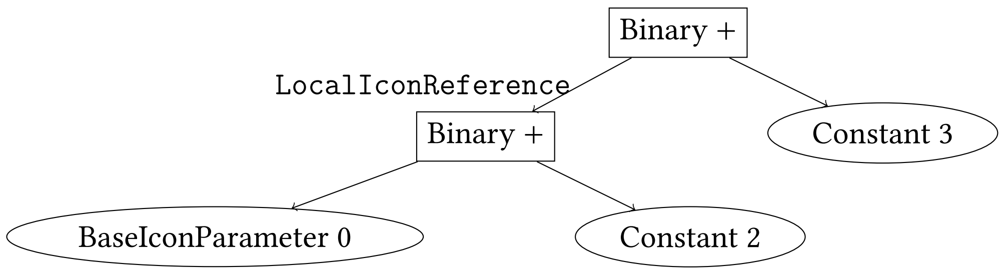


Figure 3.2 Expression tree for a custom operation that sums a parameter with 2 and 3

they are addressable by name.

Expression metaphor

We now have all the pieces necessary to understand the metaphor introduced in section 3.1. We mentioned that the internal program of a custom operation can be understood as an expression. With the insight gained in this section, we can elaborate on this further. The main building blocks of this expression are icons and `IOP`s. The icons act as nodes with 0 or more children, which are `IOP`s. In the case of the `LocalIconReference` type, the child of the node is not the `IOP` itself, but the node to which the reference points to. This is illustrated in 3.2, which shows a program computing the expression $(X + 2) + 3$, where X is a parameter of the parent custom operation.

To reiterate the main idea behind the first design, this expression is built internally for each custom operation when the user performs actions within it. It is then used when the custom operation is evaluated.

3.3 Example - Absolute Value

This section contains an example of how a custom operation computing the absolute value of an integer would be defined internally in our system. This is not something that the user would do themselves, it is just a showcase of how the data structures work. It is split into three figures: figure 3.3 contains the definition of the custom operation itself, figure 3.4 contains the definitions of the two branches of the if icon, and figure 3.5 contains the custom operation table definition.

```
let compareIconId = newIconID()
let compareIcon =
    { X = 40
      Y = 200
      IconOperation =
        Binary("<", BaseIconParameter(0), Constant(0)) }

let ifIconId = newIconID()
let ifIcon =
    { X = 40
      Y = 120
      IconOperation =
        If(LocalIconReference(compareIconId)) }

let mainIconTable =
    Map.empty
    |> Map.add ifIconId ifIcon
    |> Map.add compareIconId compareIcon

let absoluteValueCustomOperation =
    { ParameterCount = 1
      SavedIcons = mainIconTable
      EntryPointIcon = Some ifIconId }
```

Figure 3.3 A single custom operation definition using handwritten code

```
let negationIconId = newIconID()
let negationIcon =
    { X = 40
      Y = 40
      IconOperation = Unary("-", BaseIconParameter(0)) }

let falseBranchIconTable =
    Map.empty
    |> Map.add negationIconId negationIcon

let absoluteValueCustomOperationFalse =
    { ParameterCount = 1
      SavedIcons = falseBranchIconTable
      EntryPointIcon = Some negationIconId}

let identityIconId = newIconID()
let identityIcon =
    { X = 40
      Y = 40
      IconOperation = Unary("+", BaseIconParameter(0)) }

let trueBranchIconTable =
    Map.empty
    |> Map.add identityIconId identityIcon

let absoluteValueCustomOperationTrue =
    { ParameterCount = 1
      SavedIcons = trueBranchIconTable
      EntryPointIcon = Some identityIconId}
```

Figure 3.4 Definitions used for the true and false branches of an If icon

```
let absoluteValueCustomOpName = "absoluteValue"
let falseBranchCustomOpName, trueBranchCustomOpName =
    // This function generates custom operations names for
    // an if icon within a specific custom operation
    getCustomIfIconNames absoluteValueCustomOpName ifIconId

let customOperations =
    Map.empty
    |> Map.add
        absoluteValueCustomOpName
        absoluteValueCustomOperation
    |> Map.add
        falseBranchCustomOpName
        absoluteValueCustomOperationFalse
    |> Map.add
        trueBranchCustomOpName
        absoluteValueCustomOperationTrue
```

Figure 3.5 Custom operation table definition for the absolute value example

Example shown in program

The following example will show a brief overview of how such a custom operation would be created in the program itself by an actual user.

We first create a new custom operation with the name Abs and 1 parameter. We place this new icon on the screen and add a -20 to its parameter.

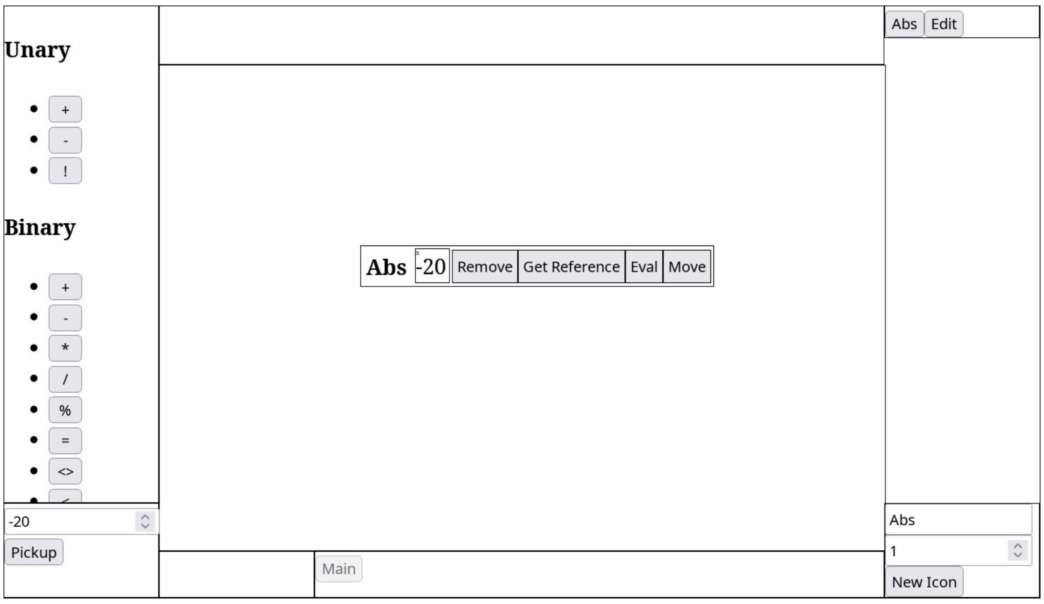


Figure 3.6 Program with Abs icon placed

After clicking the Eval button, we immediately hit a trap and are shown a new tab to define the Abs custom operation. The first icon we create is saved as the entry point icon, so we first place an If icon. Then, we place a comparison icon, to compare the input to 0.

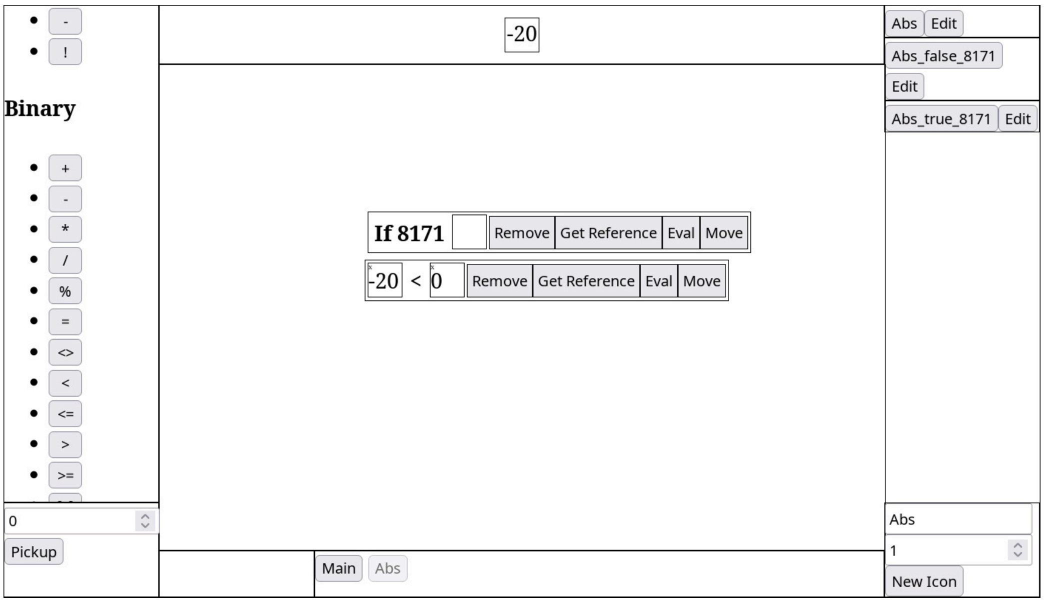


Figure 3.7 Start of definition of Abs, note the `false` and `true` internal custom operations

We then evaluate the comparison, place its result into the parameter of the If icon and evaluate the If. This starts evaluating the true branch custom operation (because -20 < 0). Because we have not touched it yet, we immediately hit a trap. Since we are doing the absolute value, we add a negation icon for this case, and place the parameter of this custom operation call into it.

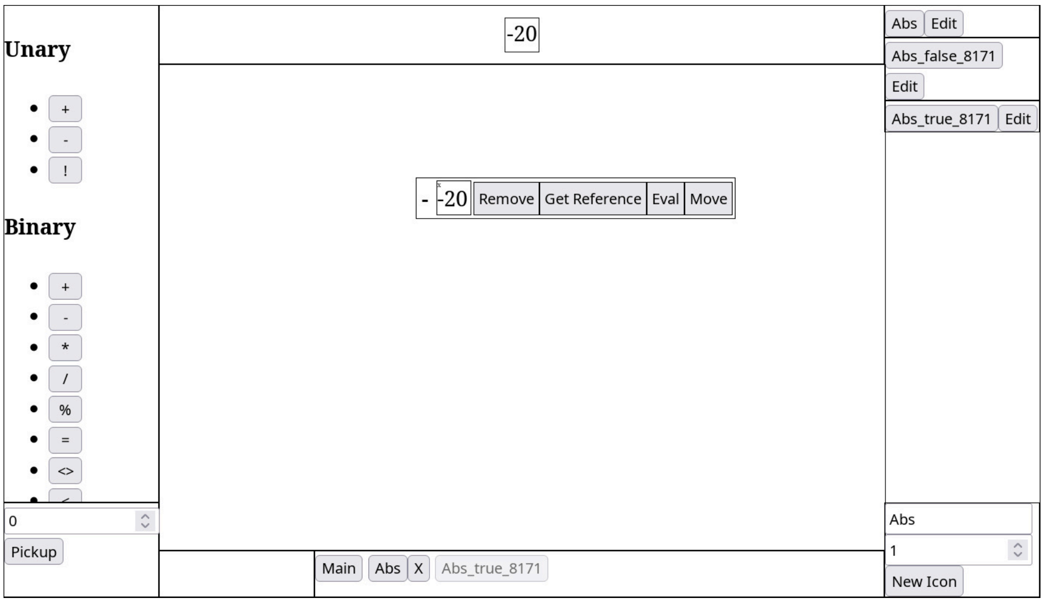


Figure 3.8 State of program inside the true branch, as evident by the call stack on the bottom of the screen.

We then have to evalute the negation, return to the call site and retry evaluation. In this case, this means the evaluation of the If, and then the evaluation of the Abs itself. As we can see in figure 3.9, the absolute value now works for negative values, but we still have to finish defining it for positive values.

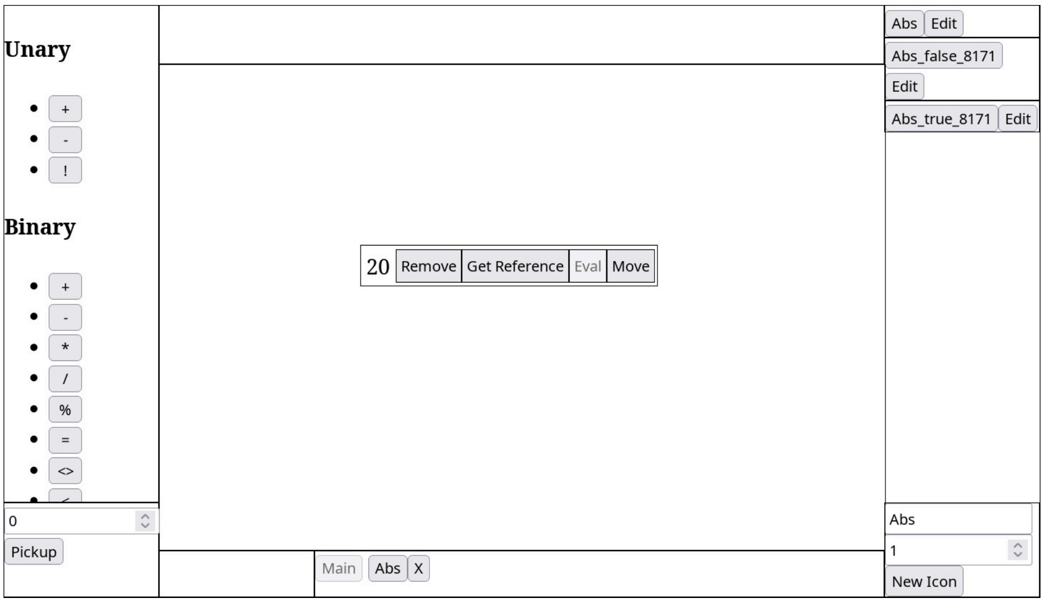


Figure 3.9 Finished Abs icon for negative values.

The last step is to define the absolute value for positive integers. We just have to add a new Abs icon to the screen, place a positive value into its parameter and hit the `Eval` button. Because our number is now positive, the comparison with zero evaluates to false, and the If executes the false branch. We have not defined that yet, so we hit a trap and are shown the false branch custom operation. We only have to add a unary + to the screen and evaluate it with the single parameter of our icon to finish the definiton of the absolute value.

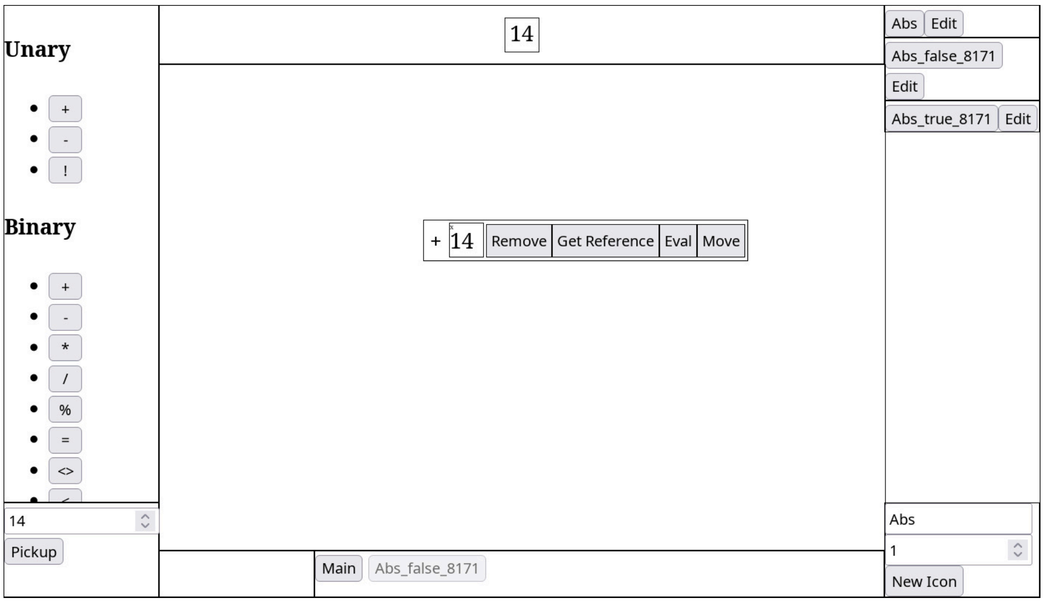


Figure 3.10 State of the program inside the false branch.

3.4 Evaluation

There are two different types of evaluation within our system. Evaluation of a custom operation, and evaluation of an icon. We will now briefly cover both.

3.4.1 Icon evaluation

icon evaluation is done by using a recursive evaluation function. The parameters of the function are the ID of the icon we want to evaluate, and the context necessary for the evaluation. This context contains various helper fields, but the most important are

- a list of parameters for the current custom operation
- the name of the current custom operation
- a map used for looking up custom operations

Before we can continue, we also need to specify how parameters are evaluated. Parameters are stored using the `IconOperationParameter` type. Evaluating parameters is simple:

- `Trap` → Throws an exception with the current state.
- `Constant` → Returns the number contained within the type.
- `BaseIconParameter` → Contains an index, returns the item at said index in the custom operation parameter list in the context.
- `LocalIconReference` → Evaluates and returns result of referenced icon.

In reality, all of these evaluations also return a *results table*. This is just a map from icon ID to a number, which is the result of the specific icon. This is necessary for drawing the results of intermediate evaluations, and also for recreating the state when a trap is thrown. The evaluation of an icon combines the result tables from the parameters and then adds its own result to it at the end. The table is then returned as the output of the icon evaluation.

Evaluation of different IconOperation types

The first step of any evaluation is to evaluate all the parameters. As mentioned above, each evaluated parameter also returns a result table, so we combine these tables into one.

Evaluation of the `Unary` and `Binary` icon operations is simple. We just apply the appropriate function to the evaluated parameters. Both of these types store a string that represents the operation. This string is used to choose the correct operation. Afterwards, we save the result of the operation in the results table with the current ID as the key, and return the table.

Evaluating an `If` icon operation is a bit more complex. After evaluating its single parameter, the result is interpreted as a boolean value, and the corresponding branch is then evaluated. Creating an icon with an `If` operation also automatically creates two internal custom operations with names that cannot be replicated by the user, because they contain invalid input characters. The name of these icons s the name of the parent custom operation, concatenated with *true* or *false*. These internal custom operations have the same number of parameters as the parent. When the only `If` parameter is evaluated, the boolean result of it is converted to a string and appended to the current custom operation name. This string is then the name of the correct branch to evaluate. A mockup of this can be seen in figure 3.3 and figure 3.4.

The evaluation of the correct branch itself just runs custom operation evaluation, with copied parameters. This is mostly equivalent to evaluating `Call-CustomOperation` with the correct branch. Since both `If` and `CallCustom-Operation` perform custom operation evaluation, we will move on to explaining it now.

3.4.2 Custom operation evaluation

Custom operation evaluation builds on icon evaluation. This evaluation requires the name of the custom operation we want to evaluate, and a list of numerical parameters. The evaluation itself is performed by running icon evaluation on the entry point icon of our custom operation. The icon evaluation returns a table of results. We extract the result of the entry point icon and return it as the result of our evaluation. The key difference from icon evaluation is that we do not have to store the temporary results of the icons anywhere[1]. Because of this, we do not have to return the entire table, just the result of the single icon.

3.4.3 Traps

Traps are a concept used for defining operations that have not yet been encountered. Whenever we try to preform an action that is not defined, such as evaluating an icon with an empty parameter, or evaluating a new custom operation, an expection is thrown with the current state. The system uses this state to show us where the trap was encountered and allows us to define the missing actions. As mentioned prior, this is the main method for defining custom operations.

[1]This is not always true, we do return the temporary state through an exception if we encounter a trap at any point

3.5 Issues

3.5.1 The issue

Our first system evaluates things based on an icon tree. An icon is evaluated only if it is in this tree. The tree root is the entry point icon. In other words, an icon is only evaluated if it is reachable from the entry point icon through `LocalIconReference` parameters. This reachability is transitive. To preserve the interaction style of Pygmalion, our system only allowed the user to pickup a reference to an icon after it has been evaluated. If this reference is then placed inside the parameter of another icon, that icon becomes its parent in the tree. This style of interaction forces us to build computations from bottom up, since we first have to compute operations This is what caused the main issue however. Since we build the computation from bottom-up, but icons have to be evaluated to be added to the tree, recursive calls within our implementation cannot work properly, because we cannot connect a recursive call to our tree before its defined.

We can look at this from a different perspective. Logically, we need to have the action of evaluating the recursive call saved immediately, so it is performed within that recursive call again. However, because the recursive call needs to successfully evaluate before it can be saved, there is no simple way to achieve this. The only way to do this without changing anything within the system is to painstakingly step through every recursive call until we reach the base case of the recursion. Then, the base case can be defined. After that, we can finally add the factorial call to the tree in the second to last recursive call.

3.5.2 Possible fixes

We considered a number of possible fixes, which we will explain in the following text:

- Fix #1 → Allow the user to pick up a reference to an icon before it is evaluated
- Fix #2 → Change how If behaves
- Fix #3 → Try evaluating most icons on the screen
- Fix #4 → Save the order of evaluation

Fix #1

Fix #1 is actually implemented in the last version of the first design, to allow the system to create a factorial custom operation. This, however, went against the

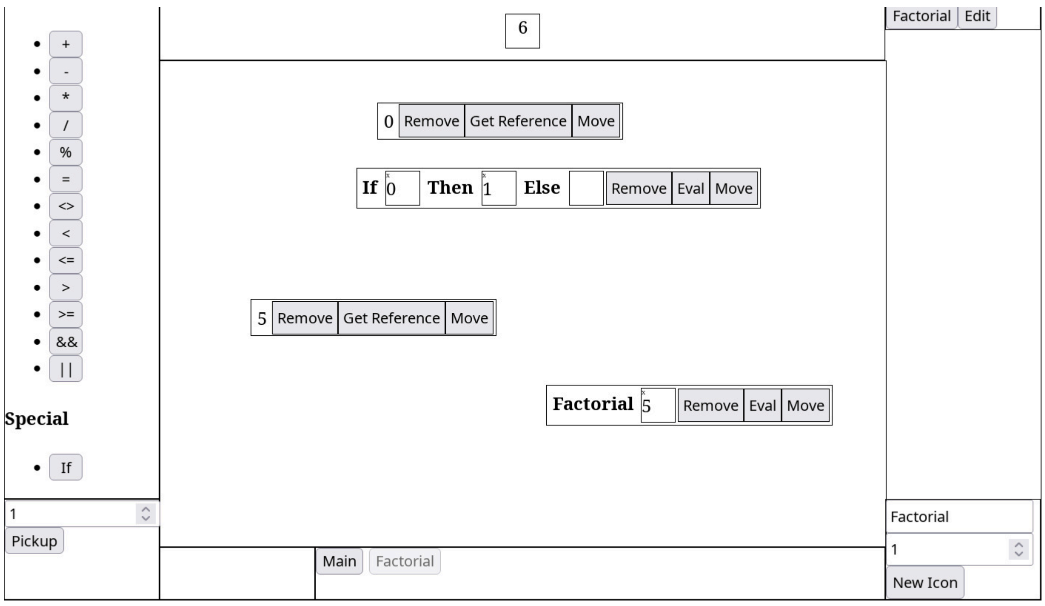

+
-
*
/
%
=
<>
<
<=
>
>=
&&
||
Special
If
1
Pickup
6
0 Remove Get Reference Move
If 0 Then 1 Else Remove Eval Move
5 Remove Get Reference Move
Factorial 5 Remove Eval Move
Main
Factorial
Factorial Edit
Factorial
1
New Icon

Figure 3.11 Recreation of the recursive issue in an older version of the software. There is no way for the user to connect the "Factorial" icons result to the false branch of the "If" icon prior to evaluation.

core ideas of Pygmalion. We wanted to retain the interaction of moving an icons result to the parameter of a different icon. Having an explict "Get Reference" button felt too disconnected from this. So we searched for a different fix. Our implementation of this is visible in figure 3.11. We didn't remove the fix when trying the other ones to allow for experimentation.

Fix #2

Fix #2 is also implemented in the most recent version of the first design. Since we encountered this issue for the first time with an If icon, we thought that it was a inherent problem of the If operation. Our idea to fix this was simple. If we can't connect something to the If before it is evaluated, force that connection somehow. We force a connection by creating mock custom operations for both the true and false branch. It follows then, than the entry point icon within the false and true branch custom operations is "connected" to the If even without it being evaluated. This fix did not help, it only managed to move the issue elsewhere.

It could work if we made every parameter of every icon its own custom operation. This would basically allow us to add items into the tree before their evaluation. This approach is clearly unusable, and in the end only does the same thing as fix #1.

Fix #3

When thinking about fix #3, it very quickly runs into issues. If we have, for example, a recursively defined icon, it would get evaluated on every level of recursive nesting. This means it would run into an infinite loop. So a simple extension of this idea is only evaluating all icon trees on the screen. A tree being a structure composed of icons connected by the LocalIconReference parameter type. This also runs into issues, because we don't know what order to evaluate the trees in. For example, in figure 3.11, if the system were to evaluate the "If" icon first, it would hit a trap, because the branches are not defined, and stop immediately. If it were to evaluate the tree with the recursive "!" call first, we might have successfully recursed, but we would never stop at the base case, since the "!" icon would just get evaluated over and over forever. There might be a way to get around this issue, but we moved on to try a different idea.

Fix #4

This was the last fix we considered. We would a have list of which icons were evaluated and in what order. It seems that this fix would work, since it would fix the main issue of the recursive call not evaluating. We did not implement this fix however. This is because we decided to pursue a different design altogether instead of trying to fix this one. We will discuss our reasoning for this in later sections.

3.6 Conclusions

We first created the first design with the core idea of building an expression tree to represent the computations done within custom operations. After implementing it, we ran into the issue discussed above. We then noticed a connection between fix #4 and Pygmalion itself.

3.6.1 How Pygmalion does it

When a user performs an action within Pygmalion when it is in "remembering" mode, the action is saved inside its data structure for further use. In this section, we will call this data structure an *action list*[2] When a user-defined icon is later evaluated, these actions are played back internally to get the result of the evaluation. In the case when a user tries to evaluate a custom operation "A" while

[2]In reality, Pygmalion creates a representation of user actions with its own internal language and adds them to a *code list*. For our purposes however, understanding it as a list of actions with occasional branching is enough.

defining said custom operation, the action ”Evaluate A” is immediately saved into the action list of A and played back during that same evaluation. This causes the program to recurse, until some input leads us down a different action path. An example of this is if some input results in the program moving down a false branch instead of a true branch. This other path could then result in the end of the evaluation, or it could contain a trap, requiring the user to continue defining the icon. Either way, the key idea here is that for recursive definition to work, the action of trying to evaluate a recursively defined icon must be saved into the action list immediately, so it is played back in said evaluation. Otherwise, evaluating an icon within itself would just cause the action list to end right before the evaluation, causing the user to hit a trap. This would mean that they would have to redo said evaluation for every recursive iteration. This is exactly the issue with our implementation.

3.6.2 Conclusions continued

Creating a list of icon evaluations would be similar to the action list in Pygmalion with everything except evaluations removed. We then arrived to the conclusion that using an idea similar to an action list for every action instead of just for evaluations would be a simpler representation of the program. Such a change would also fix the main issue with the first design. These factors combined led us to abandon this design in favor of implementing the proposed idea. That implementaion will be discussed in the following chapter.

Chapter 4

Second System Design

Similarly to the previous chapter, we will describe the second iteration of our system. We will first start by describing the core differences between the second design and the first one. Afterwards, we will discuss the data structures that are used within the systems, their evaluation, and show their usage by manually building a program that can compute the absolute value of an integer.

4.1 Core differences

At the end of the last chapter, we discussed why we switched to a different design. Now we will talk about what we changed in more detail.

The core idea of the first design was using user actions to modify and build something akin to an expression. That expression was then used to replicate the same calculation that the user performed by hand. The second design does not build an expression, it just saves most user actions into a list to be replayed when a computation is replicated (e.g. when a custom operation is being evaluated). When we talk about user actions, we mean something that the user does that changes the state of the program. This includes things like creating a new icon, evaluating an icon, picking up a constant etc.

4.1.1 State simplification

The idea of storing user actions directly greatly simplifies the architecture of our program. Because we use the Elmish framework[14] for our implementation, we already need to have a state, multiple message types, an update function that takes the current state and updates it based on the received message and a view function that renders the current state. In short, messages are sent everytime the user interacts with the UI, and we get to decide what messages with what data get sent during which interaction. When a message is received, the state is updated with the update function.

If we want to define a custom operation, we could just save all the messages that were received during its creation, and replay them when we evaluate it. This replaying could be done by repeatedly applying the update function to some base state for every saved message. In reality, the process is slightly more complicated, but this is the main idea behind it.

Since the users actions don't have to be saved as an expression anymore, we can make their effects a lot simpler. The first major change was to the `IconOperationParameter` (`IOP` for short) type. In the first design, this type was used by icon operations to know where and how to source their parameters. For example, if we placed the result of one icon into anothers parameter, it would create a `LocalIconReference` instance and save it into the parameter. This reference would then followed and recursively evaluated by the program when evaluating the icon. In the second design, this is greatly simplified. The `IOP` type is now either a number or a `None`. We will discuss this in greater detail in the following sections. With the second design, we can just store the action of the user picking up the result of one icon, and the action of placing the result into a parameter of another icon. When this is replayed later, the same actions are performed and the end state is the same (if we used the same inputs). The action of placing the result of one icon into the parameter of another can be generalized into an action of placing a number into a parameter. This number placement action in combination with "pick-up number" and "pickup custom operation parameter" actions cover every previous use of the `IOP` type.

The evaluation of icons is also simplified. We only have to check if all parameters contain a number, and if so, we can immediately perform the computation defined by the icon operation. In other words, we don't have to do any recursive icon evaluation (except evaluating custom operations).

4.2 Data structures

In this section we will explain the main data structures used for the logic of the system. Similarly to the first design, we will talk about the data structures representing icons and custom operations first. An analogous explanation will follow for data structures representing user actions and the core system state.

4.2.1 Icons and custom operations

The `IOP` type represents the parameters of icons, which can have two states. It either holds a number or nothing. This is represented by the `int option` type.

```
type IOP = int option

type IconOperation =
    | Unary of operator : UnaryOperation * IOP
    | Binary of operator : BinaryOperation * IOP * IOP
    | If of IOP
    | CallCustomOperation of CustomOperationPrism * IOP list
type IconOperationParameter = IOP

type Icon =
    { X : int
      Y : int
      Operation : IconOperation
      Result : int option }

type CustomOperation =
    { Name : string
      ParameterCount : int
      ActionTree : ExecutionActionTree }
```

Figure 4.1 Types used for representing icons and custom operations

The `IconOperation` type is mostly equivalent to its counterpart from the first design. The only difference is the usage of the `CustomOperationPrism` type. This will be a common theme in the second design, because we replaced most usages of indices with optics. Optics are a way to access and modify data structures in a functional way, as discussed in section 1.3.2. Another change is in the behaviour of `If`, but that will be discussed along with evaluation.

The `Icon` type is also similar to the first design, with one key difference. The result of the icon is now contained within it, instead of a disconnected table. Due to the simplification of evaluation, evaluating an icon will not affect any other icon. Because of this, we do not need a results table anymore.

The final type is the `CustomOperation` type. It contains the name of the custom operation, the number of parameters it takes, and an *action tree*. In the previous sections, we mentioned that we could save actions in an action list. This was a simplification. In reality, we actually need a tree-like structure, to save multiple branches of actions, because of `If` icons. This will be discussed in the next subsection.

4.2.2 State and actions

```
type MovableObject =
    | NoObject
    | ExistingIcon of IconPrism
    | NewIcon of IconOperation
    | Number of UnderlyingNumberDataType

type MovableObjectTarget =
    | Position of x : int * y : int
    | IconParameter of target : IconPrism * position : int

// Some actions ommited for brevity
type SimpleExecutionAction =
    | EvaluateSimpleIcon of IconPrism
    | PickupNewIcon of IconOperation
    | PickupIcon of IconPrism
    | PickupNumber of UnderlyingNumberDataType
    | PickupExecutionStateParameter of parameterIndex : int
    | PickupIconResult of IconPrism
    | PlacePickup of MovableObjectTarget

type BranchingExecutionAction =
    | EvaluateBranchingIcon of IconPrism

type FinalExecutionAction =
    | SaveResult
    | Trap

type ExecutionActionTree =
    | Linear of SimpleExecutionAction * ExecutionActionTree
    | Branch of
        action : BranchingExecutionAction *
        falseBranch : ExecutionActionTree *
        trueBranch : ExecutionActionTree
    | End of action : FinalExecutionAction

type ExecutionState =
    { HeldObject : MovableObject
      LocalIcons : Icon list
      CurrentBranchChoices : bool list
      Parameters : int list
      Result : int option }
```

Figure 4.2 Types used for representing state and actions applicable to it.

As we can see in figure 4.2, the `SimpleExecutionAction` type is used to store relevant user actions. An instance of this type is passed as a message to the program when the corresponding user action is performed. This type contains more actions than are shown, such as the action of picking up an existing icon to move it, or removing an existing icon. Basically, every action that is necessary to reproduce a custom operation is present in this type. For example, the action of changing the text inside a number input is not contained within this type, but the action of picking up a number from said number input is. This is because we do not need to replay the actions of typing out every digit of a number one by one, we just need the action of picking up the resultant number. There are two more types of actions in figure 4.2. The first is the `BranchingExecution-Action` type, which encompasses actions that can divert the flow of the program. This will be discussed further when talking about how actions are stored. The only value this type can have is `EvaluateBranchingIcon`. There is only one type of branching icons in our system, icons with the `If` operation.

The last action type is `FinalExecutionAction`. This type is used for actions that end the evaluation of a custom operation. The only two actions which do so are traps and saving (returning) the result of a custom operation. Traps represent paths in the custom operation that have not yet been defined. Naturally, this means that a finished custom operation returns a result in every branch.

Storing actions

Actions in our system are stored using the above mentioned types saved inside a tree-like structure. This structure is built using the `ExecutionActionTree` type. The values of this type wrap the already mentioned action types and allow them to connect as a tree. They work in the following way:

- `Linear` → `SimpleExecutionAction` | always has one child
- `Branching` → `BranchingExecutionAction` | always has two children
- `End` → `FinalExecutionAction` | has no children, marks end of current branch

State

The last data structure we will talk about is the `ExecutionState`. This structure represents the current state of a calculation. The items it holds are the current held object, a list of all the icons on the screen, a record of choices for the branches, and a list of parameters and result for the current custom operation call.

4.3 Example - Absolute Value

Again, we will create a manually written example of an operation computing the absolute value of an integer. This is mainly to show the vast difference between the first and second designs. To reiterate, this is not something the user will do themselves. This is just for showcase purposes. Note that this will be simplified. We will use indices instead of prisms for icons. Unary and binary operations will be defined just by using functions. Finally, we will omit most of the tree structure wrappers and store actions in plain lists. This is not how these data structures work, but writing out a tree manually is confusing and cumbersome. The code is available in figure 4.3. The resultant tree of actions can be seen in figure 4.4.

UI Example

We will not provide a UI example as the user interaction is very similar to the example in section 3.3. The only major difference is the usage of the `If` icon. A better example of how the second design works can be found in section 5.1

4.4 Evaluation

Similarly to the first design, we could split the evaluation into icon evaluation and custom operation evaluation. However, due to design changes, icon evaluation is now very simple. It consists of checking if every parameter has a number. If all of them do, the internal `IconOperation` instance is evaluated. Due to this simplicity, we will only talk about evaluation of pre-defined and custom operations.

4.4.1 Pre-defined operation evaluation

The simplest type of evaluation is the evaluation of `Unary` and `Binary` operations. Both contain a function with the correct amount of arguments. The function is applied to the parameters and its result is the result of the evaluation.

Evaluation of the `If` operation is a bit different. The single integer parameter is first converted into a boolean. This is done by understanding anything non-zero as `true` and zero as `false`. Then, this boolean is saved into the `CurrentBranch-Choices` table of the state. Afterwards, this list of branch choices is used when traversing the action tree. For example, this is done when a new action is being saved.

Lastly, `CallCustomOperation` triggers custom operation evaluation, which we will talk about now.

4.4.2 Custom operation evaluation

Evaluation of custom operations is done by taking a base execution state and applying actions from the tree to it sequentially. When a branching icon is encountered, its parameter is interpreted as a boolean and the evaluation continues with the corresponding branch. Due to how the action tree is structured, the end of branches in the tree always end with either a trap or the action of returning a number. Because of this, every custom operation evaluation ends in two ways. Either the evaluation succeeds, and it returns a number as the result, or it hits a trap. When a trap is encountered, an exception is thrown with the current state. This state is then rendered to the screen to allow the user to continue defining the current branch of actions.

An interesting observation is that the only thing that can change the outcome of a custom operation evaluation is the `PickupExecutionStateParameter` action, since all the other actions deal with constants. This action is the only one that is ”impure” so to speak.

```
let absoluteValueCustomOperation =
    { Name = "Absolute Value"
      ParameterCount = 1
      ActionTree = absoluteValueActionTree }

let absoluteValueActionTree = [
    // Create new icon with id 0
    PickupNewIcon(Binary((<), None))
    PlacePickup(Position(40, 40))
    // Place operation parameter into '<' icon
    PickupExecutionStateParameter(0)
    PlacePickup(IconParameter(0, 0))
    // Place 0 into '<' icon
    PickupNumber(0)
    PlacePickup(IconParameter(0, 1))
    // Evaluate the '<' icon
    EvaluateSimpleIcon(0)
    // Create if icon with id 1
    PickupNewIcon(If(None))
    PlacePickup(Position(40, 80))
    // Move result of '<' into 'If'
    PickupIconResult(0)
    PlacePickup(IconParameter(1, 0))
    // Evaluate the If and pick correct branch
    Branch( EvaluateBranchingIcon(1),
            falseBranchActions,
            trueBranchActions )
]

let falseBranchActions = [
    // Value is positive, just return
    PickupExecutionStateParameter(0)
    SaveResult
]

let trueBranchActions = [
    PickupNewIcon(Unary((~-), None))
    // Creates icon with id 2
    PlacePickup(Position(40, 120))

    PickupExecutionStateParameter(0)
    PlacePickup(IconParameter(2, 0))
    EvaluateSimpleIcon(2)
    PickupIconResult(2)
    SaveResult
]
```

Figure 4.3 A single custom operation definition using handwritten code

$$\begin{tikzcd}
& \texttt{PickupNewIcon(Binary((<), None))} \arrow[d] & \\
& \texttt{PlacePickup(Position(40, 40))} \arrow[d] & \\
& \texttt{PickupExecutionStateParameter(0)} \arrow[d] & \\
& \texttt{PlacePickup(IconParameter(0, 0))} \arrow[d] & \\
& \texttt{PickupNumber(0)} \arrow[d] & \\
& \texttt{PlacePickup(IconParameter(0, 1))} \arrow[d] & \\
& \texttt{EvaluateSimpleIcon(0)} \arrow[d] & \\
& \texttt{PickupNewIcon(If(None))} \arrow[d] & \\
& \texttt{PlacePickup(Position(40, 80))} \arrow[d] & \\
& \texttt{PickupIconResult(0)} \arrow[d] & \\
& \texttt{PlacePickup(IconParameter(1, 0))} \arrow[d] & \\
& \texttt{EvaluateBranchingIcon(1)} \arrow[dl, "\textbf{False}"'] \arrow[ddr, "\textbf{True}"] & \\
\texttt{PickupExecutionStateParameter(0)} \arrow[d] & & \\
\texttt{SaveResult} & & \texttt{PickupNewIcon(Unary((-), None))} \arrow[d] \\
& & \texttt{PlacePickup(Position(40, 120))} \arrow[d] \\
& & \texttt{PickupExecutionStateParameter(0)} \arrow[d] \\
& & \texttt{PlacePickup(IconParameter(2, 0))} \arrow[d] \\
& & \texttt{EvaluateSimpleIcon(2)} \arrow[d] \\
& & \texttt{PickupIconResult(2)} \arrow[d] \\
& & \texttt{SaveResult}
\end{tikzcd}$$

Figure 4.4 Action tree for absolute value custom operation

Chapter 5

Evaluation

This final chapter serves as an evaluation of our design. First, we will demonstrate how a factorial icon can be created using our system. We will follow the steps outlined in the original article [1, pp. 126-138] as closely as possible. Afterwards, we will discuss the differences between the style of interaction with the two systems and what we have learned from these differences. Finally, we will share some closing thoughts about Pygmalion as a programming system.

5.1 Factorial example

First, we create a custom operation with the name *factorial* and with 1 parameter. We then create an ican with this custom operation, and save a constant of 6 in its parameter box.

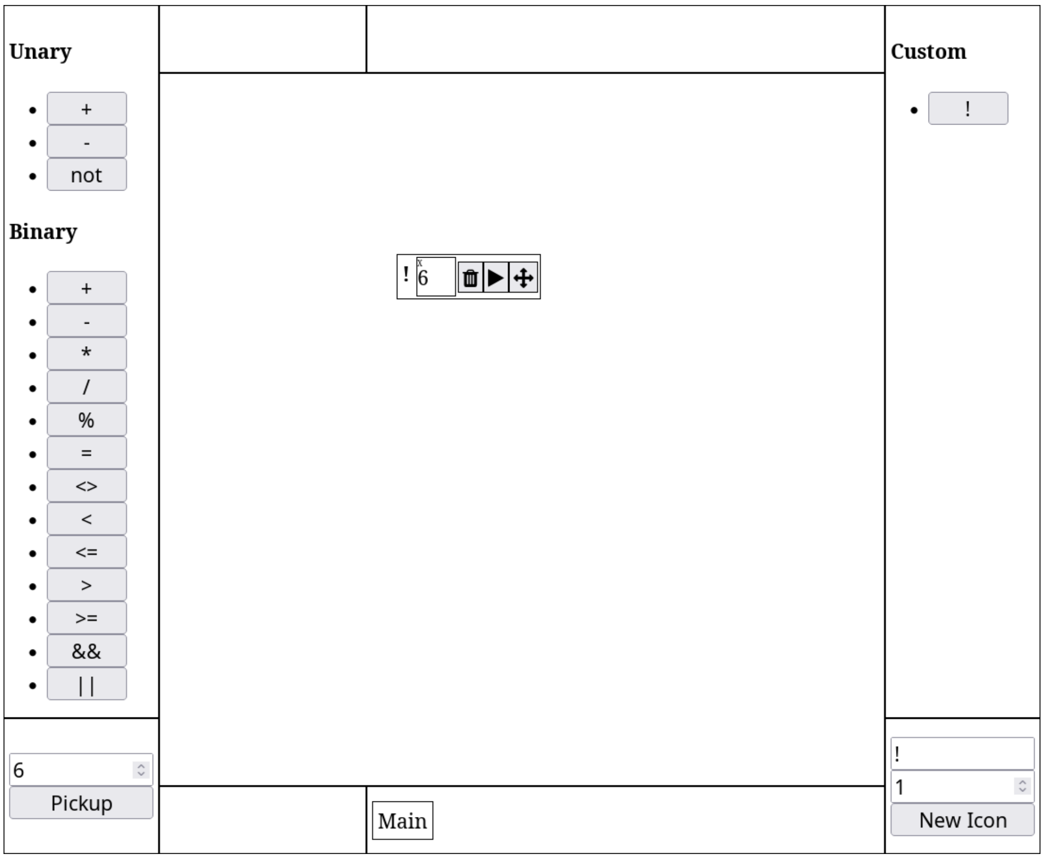


Figure 5.1 Main tab with a new factorial icon and a constant 6 as a parameter

After we click the evaluate button on the factorial icon, we hit a trap. This is because the custom operation is new and there are no steps defined within it. Hitting a trap opens the custom operation which caused it so we can define the missing steps. In the bottom of the screen we can see the current nesting level, which confirms we are defining the factorial operation. We also can see that the 6 we input as a parameter is available to pick up from the top of the screen.

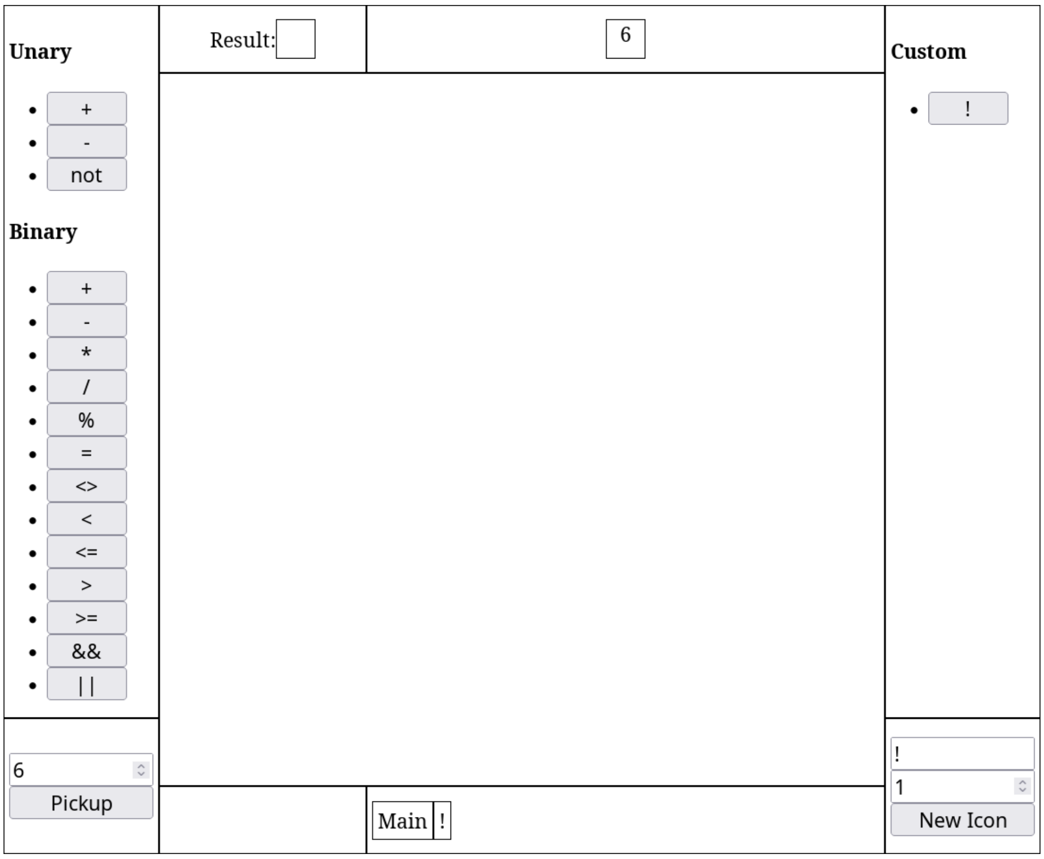


Figure 5.2 Program displaying the current nesting level (bottom) and the parameter of the factorial operation(top)

We then add the equality and branching icons. We use these icons together to check if we are evaluating the base case of the recursive factorial. The steps we need to perform to get to this state from the last one are as follows:

- Picking up the 6 from the top of the screen and placing it in one of the parameter boxes of the equality icon.
- Picking up a constant 1 and placing it in the other parameter box of the equality icon
- Evaluating the equality icon and placing its result into the branching icon
- Evaluating the branching icon

After evaluating the branching icon, the program will save all of our steps within the correct branch. In this case, this is the false branch.

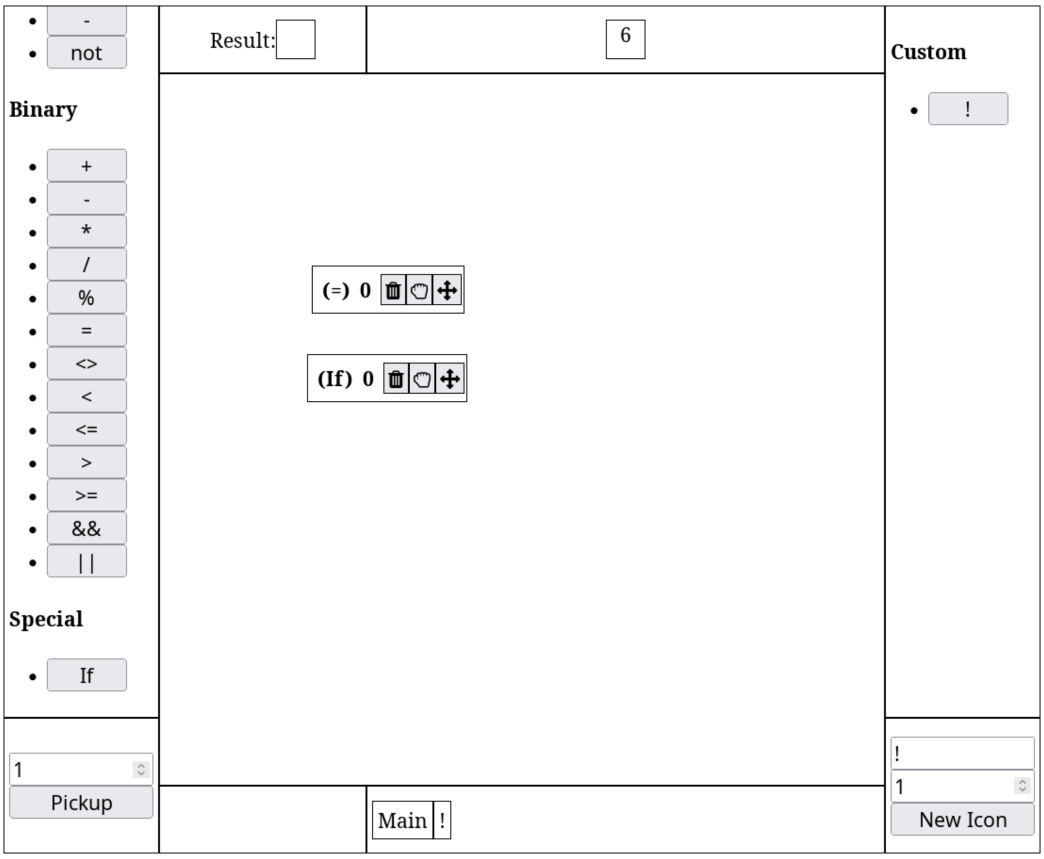


Figure 5.3 'If' icon is evaluated to 0 (false)

Now that we are in the false branch, we can start defining the recursion. We create a subtraction icon and another factorial icon. We use the subtraction icon to subtract 1 from the parameter, and place its result into the recursive factorial icon. This leaves us with the screen shown in figure 5.4.

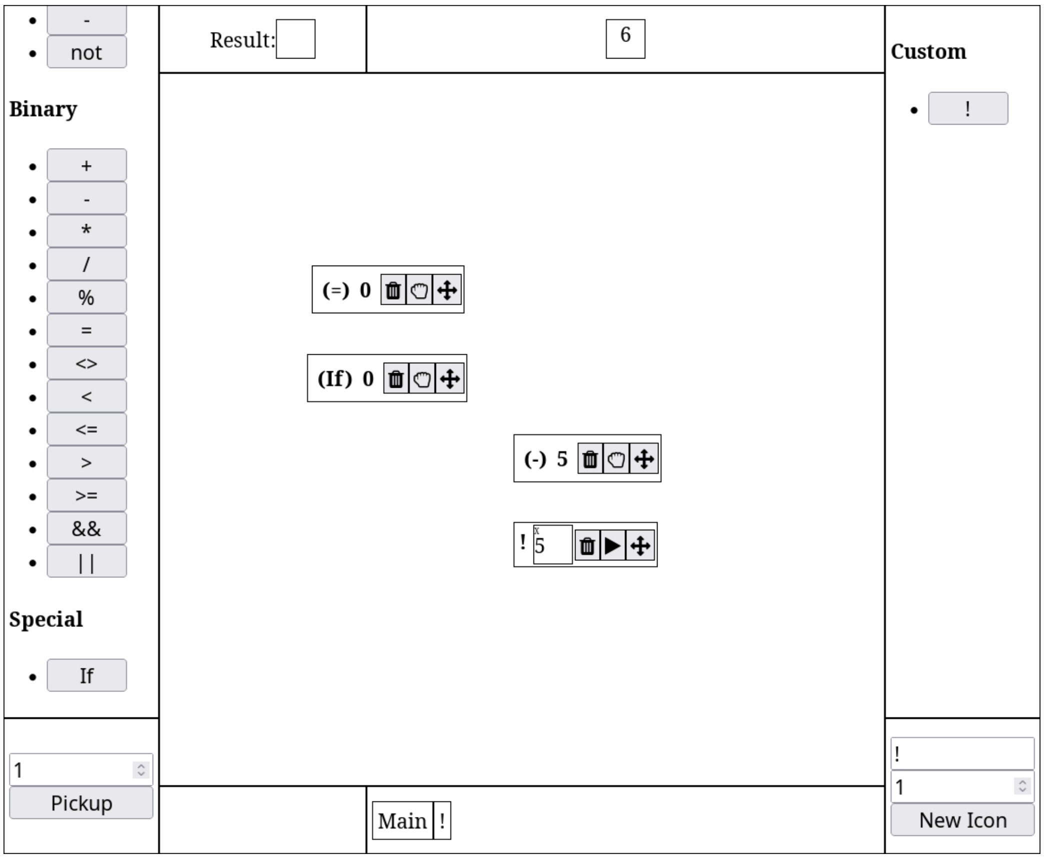


Figure 5.4 Icon ready for recursive evaluation

When we try evaluating the nested factorial icon, all of the steps we have already done have been saved, and start being played back internally. This results in the factorial custom operation being evaluated with the parameter of 5 first, which evaluates the custom operation with the parameter 4. This continues until the recursion reaches the value of 1. When this happens, the branching icon we added in figure 5.3 evaluates to true. Because we have only been defining the false branch up until now, we immediately hit a trap, and are placed within the offending custom operation call. We can see that the parameter in figure 5.5 is now 1 instead of 6, showing us that we have been stopped at the true branch.

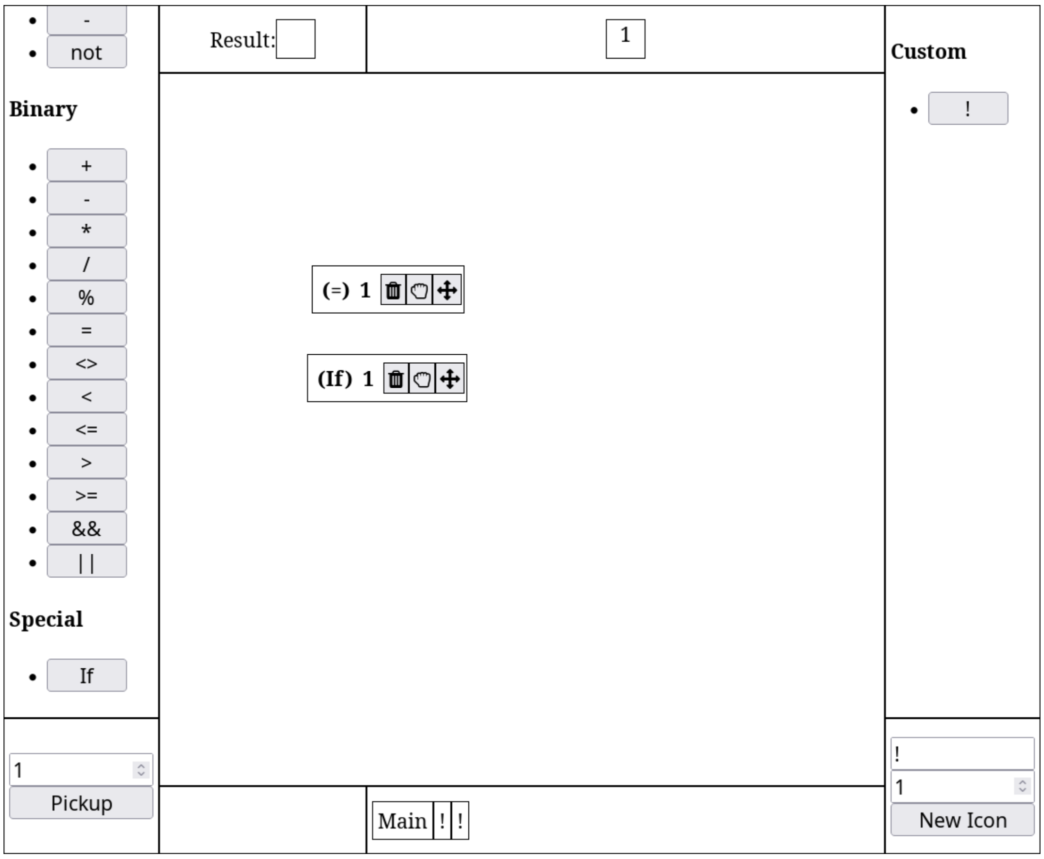


Figure 5.5 State of the program when the base case of the recursion is reached

The factorial of 1 is 1. We use the result box on the top left of the screen to save the constant 1 as a return value for this base case. This means that the factorial icon now returns the correct result when called with the parameter of 1.

After we save the result, the evaluation we started at the beginning continues. Since the factorial is now defined for a parameter of 1, it sucessfully evaluates in the evaluation of the factorial of 2. Since we haven't had the chance to define anything after this point, we hit a trap in the evaluation of the factorial of 2. As before, the program shows us the custom operation with the parameters that caused the trap, waiting for us to finish defining it.

In figure 5.6, we can see that the factorial icon evaluated to 1.

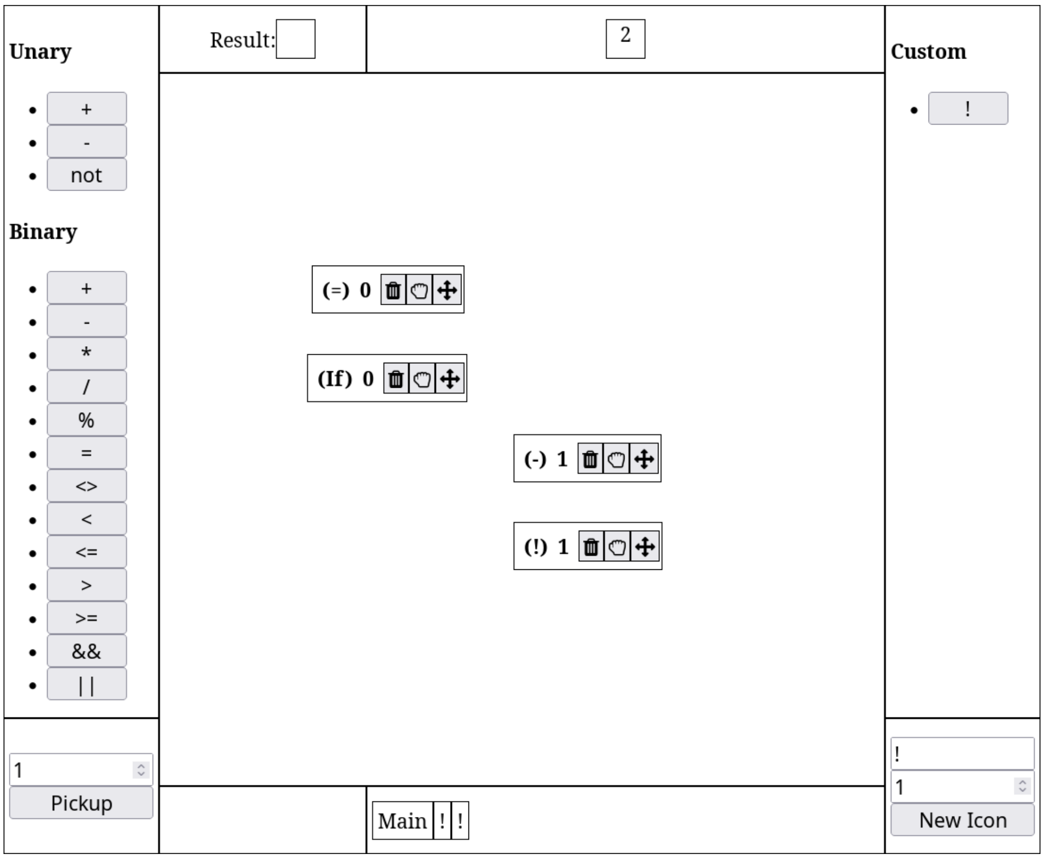


Figure 5.6 State of the program after the base case of the recursion is defined

Now, the only thing left is to create a multiplication icon, and multiply the parameter with the result of the recursive factorial call. We the save this result into the result box, finishing the definition of the factorial.

We now have an operation capable of computing the factorial for integers ≥ 1

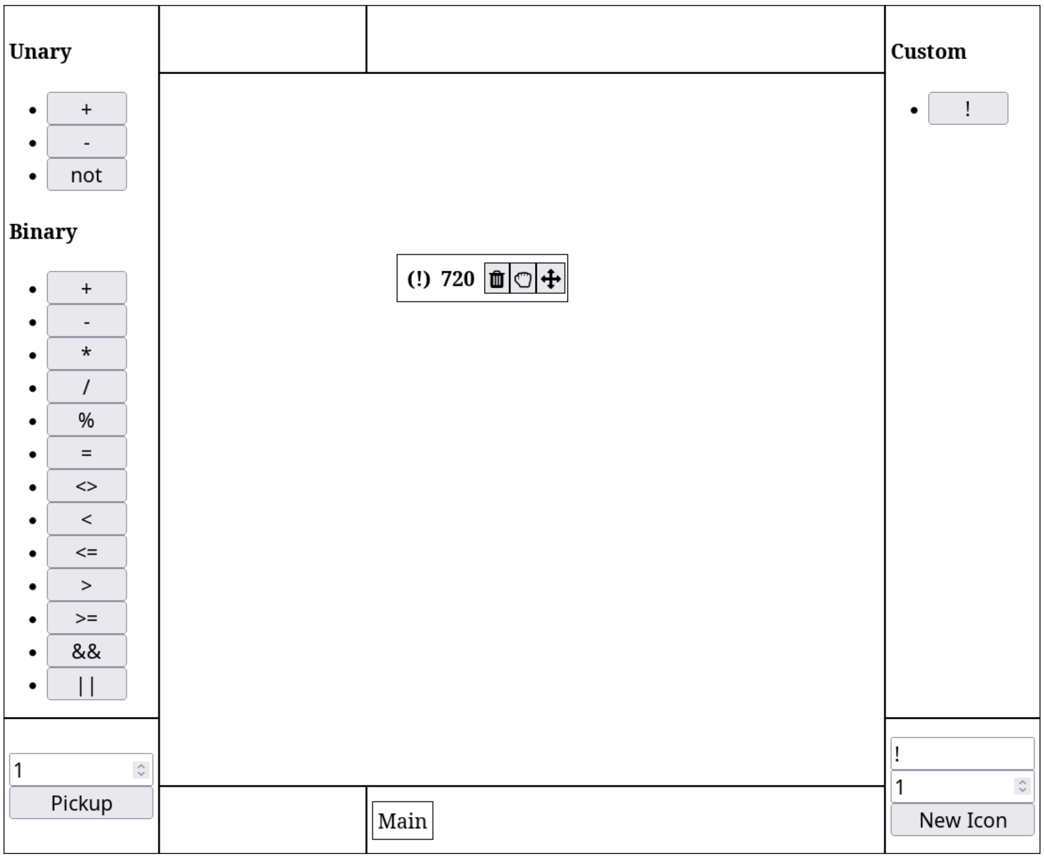


Figure 5.7 Working factorial icon

5.2 Discussion

In this discussion we will look at how our system compares to Pygmalion, and what we left out from our implementation. Then, we will look at Pygmalion itself, and discuss if its a user friendly and/or useful tool.

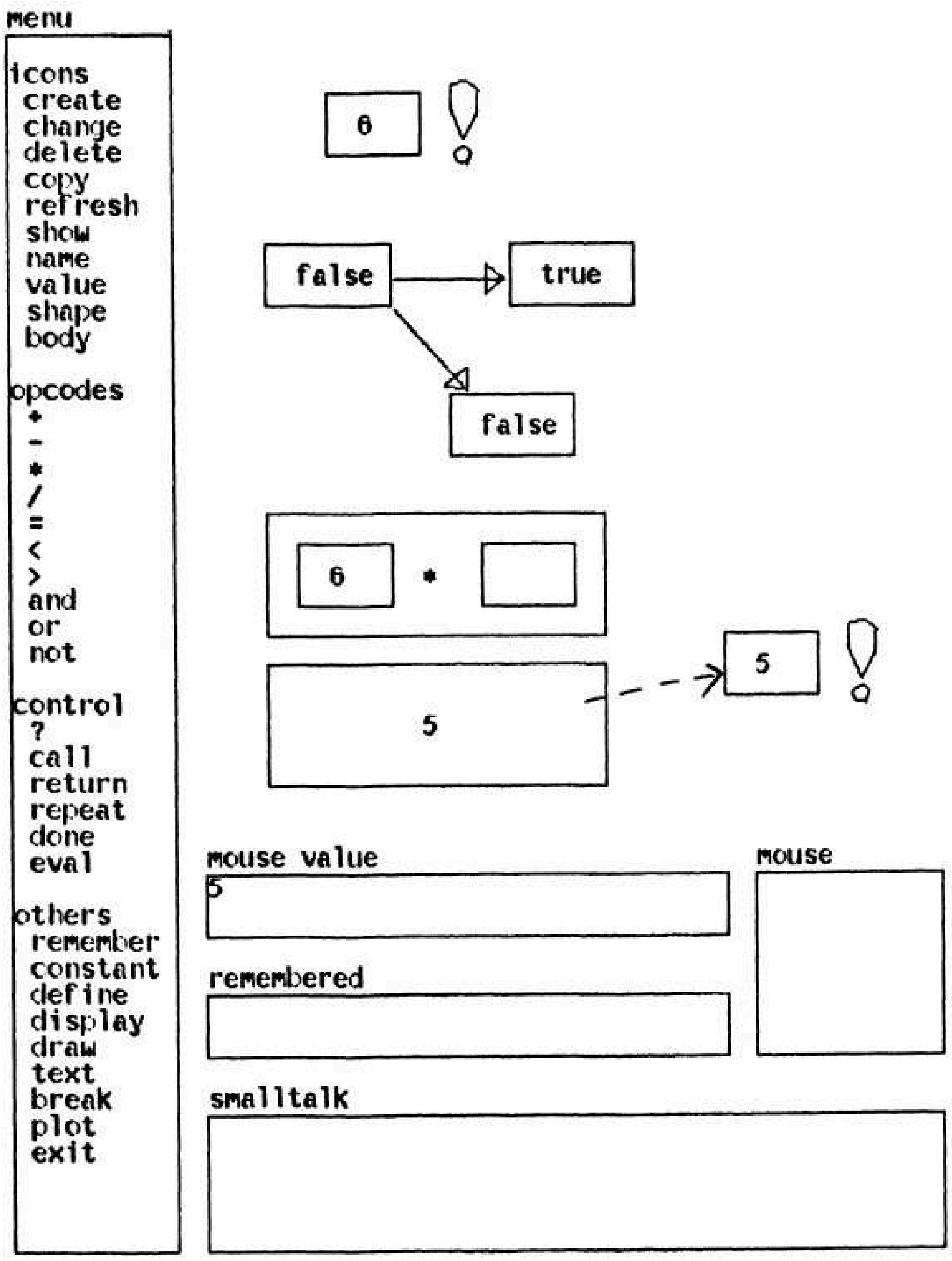


Figure 5.8 Snippet of a factorial step within Pygmalion[1]

5.2.1 Comparison

For a comparison of the user interface between our system and Pygmalion, we can use figure 3.9 which depicts our system and figure 5.8 which is a snippet from the original thesis. Both of these screenshots are at a similar step in the factorial definition in their respective systems. We will not discuss the differences between the respective UIs, as that was never our main concern.

There are many other differences between Pygmalion and our reconstruction. First of all, many actions from the menu in Pygmalion were moved to other places. Icon movement, deletion and evaluation was all moved to a small menu near the icons. We felt that this would make it a bit easier to use these operations. Another big change is the introduction of custom operations as a concept. In Pygmalion, a user-defined operation was created by creating a new icon, turning on ”remember mode” and performing actions. When we were done, we would use the ”Done” action to signify that the current branch of our operation is finished. The created operation would then be saved within the icon. To create more icons capable of doing the same operation, we had to use the ”copy” operation on the original. In our implementation, all this is done using custom operations. Instead of having to copy icons, we just create a new icon with the requested custom operation.

Another major change we made was treating icons more like functions. We made icons have a set number of parameters and an explicit result value to be picked up. Pygmalion had no such concepts, at least not canonically. Icons that behaved like functions could be created by adding subicons representing parameters to the icon. An operation on this icon could then use these subicons to perfrom some calculation, and finally drag the resultant value into the icon. This is the type of interaction that we recreated. However, this was not a requirement in Pygmalion. As seen in Cypher and Halbert [2], an icon could also have another subicon that would hold the result after the operation was finished. For our purposes however, we decided to make the result and parameters explicit, as it made the system easier to understand.

The most promiment similarity between the two systems is the style and flow of interaction when creating a factorial icon. If we follow the steps from the original thesis, we can mostly create a factorial icon in our system. The only thing we cannot recreate is one of the last steps, which multiplies the result of the factorial of 5 and returns the result. As discussed in section 2.2, we think that this step does not make sense, and was likely included for demonstration purposes. Except this step, we can recreate the factorial icon in our system, which fulfills the main goal of our project.

```
let rec factorial n =
    if n = 1 then 1
    else n * factorial (n - 1)
```

Figure 5.9 Factorial functaion written in F#.

5.2.2 Evaluation of Pygmalion

In this final section, we will talk abouthow Pygmalion holds up from a usability and usefulness perspective.

A recursive factorial definition in a regular language can be found in figure 5.9 We can see that it took 3 lines of code to create this function, and even beginner programmers can reasonably be expected to write a factorial funcion in a few minutes at most. In Pygmalion however, creating such a function is a much more cumbersome process.

Take for example the expression `factorial (n - 1)`. In the code example, this is a simple expression that could be written in 7 characters when using F#. In Pygmalion however, emulating this same expression would require at least 7 actions. We would first have to create the `-` and `factorial` icons, then place the `n` and `1` into the `-` icon, evaluate it, move its result into the factorial icon and evaluate that. This is a lot of work for a simple expression.

Another aspect to consider is that thinking about problems in a way that is compatible with Pygmalion is hard. The user has to think about programs in a depth first aproach. What we mean by that is that in a branching program, the user can only define one branch at a time. The branch that the user can define is the one that corresponds to their current input data. In the factorial example, even if the even if the user knows that the factorial of 1 is 1, they cannot define this if they are currently defining the factorial of any higher number. This fact can make defining deeply branching operations very hard. The user would have to think of input to reach every possible branch to end up with a complete program.

Another problem with Pygmalion is how the user interacts with the system. The user has to constantly switch operations to perform different actions. These operations can only be switched using a menu on the left of the screen. Pygmalion was created during a time when the computer mouse just started being used. This explains why the UI of Pygmalion can be considered cumbersome to work with. Since it was one of the early adopters of the computer mouse, there were very little guidelines and best practices on how to create a user interface that works well with it. Combined with the fact that Pygmalion could be considered a prototype, it is not surprising that the user interface is not very user friendly.

Usefulness

As we mentioned when talking about user experience in the section above, Pygmalion is very cumbersome to work with. However, we think that factorial is also not a very good measure of the usefulness of Pygmalion. We chose this example for our reconstruction to showcase the programming by demonstration aspect. However, creating a mathematical function does not tap into the ability of Pygmalion to create interesting looking icons, since we only need simple icons representing functions.

Creating icons is an important part of Pygmalion. Icons can be defined by the user to have any shape that they wish, and with programming by demonstration can be made to perform various operations. We think that this idea is very interesting, and could even be useful in a modern setting. Creating a system that allows the same interactions as Pygmalion with a modern user interface would be an interesting project.

In our opinion, the most powerful aspect of Pygmalion are its ”display” mode, and visualization capabilities. For example, ”display” mode could be used as an algorithm visualization tool. It might even be possible to create a generic graph algorithm that operations on icons representing nodes and edges. ”Display” mode could then be used to visualize the algorithm as it runs on the graph.

We propose that Pygmalion is infact a useful tool, but it needs a modern user interface to be useful in a modern setting.

Conclusion

The goal of this thesis was to develop a system that resembles Pygmalion, and allows a user to experience the style of interaction the original system presented. The system is not meant to be a fully fledged tool. It should serve as a way to experience the interaction style of Pygmalion. We first started by creating an initial design that used user actions to create a reusable expression. This design ran into issues. While these issues were fixable, we deemed another approach better suited for the system. Thus, our second design was created. The second design stored user actions in a tree structure. When the user request a calculation to be repeated, maybe even with different parameters than the initial definition, these actions were played back to get the result. This second design ended up being the one we used to develop our final reconstruction. Our final program allows the user to replicate the factorial example from the original document with very minor deviations. Other aspects of the original system are not included. The program is deployed to the internet using GitHub Pages to allow for public access[1].

From our reconstruction, we learned that replaying user actions is a more natural way to represent a system such as Pygmalion, as opposed to building an underlying expression. However, this comes with performance costs since all actions have to be internally replayed. It might occur that an icon is evaluated but its result is never used. An expression approach might be easier to optimize in such situations.

Limitations and further work

From the start of this thesis, we did not want to reconstruct the entirety of Pygmalion. We only reconstructed the factorial example. This means that we left out several aspects of the original system. The first and most obvious is that we limited the type of values to integers. This was done because the factorial example only used integers and booleans. Booleans can also be represented as integers, so we only kept those in. Another aspect that we left out is drawing

[1]`https://a-habusta.github.io/Pygmalion-Like-Reimplementation/`

the shape of icons. Pygmalion allowed you to draw the shape of your icon inside the software. Our version only allows you to change the name. We also did not implement ”display mode”. As far as we understand, this mode would display the resulting state after every operation, turning a computation into a ”movie” [1].

There is much further work to be done on this subject. To our knowledge, there is no runnable full reconstruction of Pygmalion available. Creating such a reconstruction would allow us to experience more of what the software had to offer. Another direction to explore would be the original source code written in the thesis. Extracting and running this code would offer all the same benefits as a full reconstruction would, but would also be fully authentic.

Our reconstruction could be extended to support more types of values, and recreate more of the original built-in operations, such as the ”repeat” operation. Another aspect that would be useful for understanding the program would be the ”display mode”.

In conclusion

Pygmalion is an example of historical software that was lost to time. It contains very interesting ideas that are seldom used today. We think that reconstructing and, by extension, perserving such software is an important part of computer science, to prevent the loss of ideas which may even be beneficial to the field as a whole.

Bibliography

[1] David Canfield Smith. *Pygmalion: a creative programming environment.* Stanford University, 1975.

[2] Allen Cypher and Daniel Conrad Halbert. *Watch what I do: programming by demonstration.* MIT press, 1993.

[3] Adele Goldberg and Alan Kay. *Smalltalk-72: Instruction Manual.* Xerox Corporation Palo Alto, 1976.

[4] Warren Teitelman. *Interlisp reference manual.* Xerox Corporation, 1978.

[5] *Smalltalk Zoo.* Official Site. URL: `https://smalltalkzoo.thechm.org` (visited on 07/06/2024).

[6] Tomáš Petříček. *The Lost Ways of Programming: Commodore 64 BASIC.* Official Site. URL: `https://tomasp.net/commodore64/` (visited on 07/08/2024).

[7] Editors of Encyclopaedia Britannica. *computer program.* In: *Encyclopedia Britannica.* May 2024. URL: `http://www.britannica.com/technology/computer-program`.

[8] Jeff Kramer. "Is abstraction the key to computing?" In: *Communications of the ACM* 50.4 (2007), pp. 36–42.

[9] Tina Beranic, Patrik Rek, and Marjan Heričko. "Adoption and usability of low-code/no-code development tools". In: *Central European Conference on Information and Intelligent Systems.* Faculty of Organization and Informatics Varazdin. 2020, pp. 97–103.

[10] Zhaohang Yan. "The impacts of low/no-code development on digital transformation and software development". In: *arXiv:2112.14073* (2021).

[11] Sean Kandel et al. "Wrangler: Interactive visual specification of data transformation scripts". In: *Proceedings of the sigchi conference on human factors in computing systems.* 2011, pp. 3363–3372.

[12] David L Maulsby, Ian H Witten, and Kenneth A Kittlitz. “Metamouse: Specifying graphical procedures by example”. In: *ACM SIGGRAPH Computer Graphics* 23.3 (1989), pp. 127–136.

[13] *Fable*. Official Site. URL: `https://fable.io/` (visited on 05/12/2024).

[14] *Elmish*. Official Site. URL: `https://elmish.github.io/elmish/` (visited on 05/12/2024).

[15] *Aether*. Official Site. URL: `https://xyncro.tech/aether/` (visited on 05/12/2024).

[16] Albert Steckermeier. “Lenses in functional programming”. In: *Preprint, available at https://sinusoid.es/misc/lager/lenses.pdf* (2015).

[17] *Feliz*. Official Site. URL: `https://zaid-ajaj.github.io/Feliz/` (visited on 05/13/2024).

[18] *Fable.React*. GitHub Repository. URL: `https://github.com/fable-compiler/fable-react` (visited on 05/13/2024).

[19] *Vite*. Official Site. URL: `https://vitejs.dev/` (visited on 05/13/2024).

Appendix A

Using Pygmalion-Like-Reimplementation

The second design of the software is available on the project GitHub pages site[1]. The first design must be built manually from source, available on the `first_design` branch in the project repository[2].

To compile and run the software, you need to have `dotnet` and `npm` available on your system. The required .NET Runtime version is 8.X. Compilation can be done using these commands (run in the root directory of the project):

```
dotnet tool install fable
dotnet tool restore
dotnet restore
npm update
npm run build
```

The webpage is then available in the `dist` directory. It can be run by opening `index.html` in your browser.

In case your browser refuses to open the script and stylesheet files, you can run a local server by running the command:

```
npm run server
```

This will open a local HTTP server on `http://127.0.0.1:5173/` which hosts the software.

[1]https://a-habusta.github.io/Pygmalion-Like-Reimplementation

[2]https://github.com/A-Habusta/Pygmalion-Like-Reimplementation