Eindhoven University of Technology

MASTER

Structural Syntax Visualization of Programming Languages

Westra, Bouke J.

Award date:
2023

Awarding institution:
Norges Teknisk-Naturvitenskapelige Universitet

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Trondheim, July 2023

Department of Industrial Engineering & Innovation Sciences
Human-Technology Interaction

Structural Syntax Visualization of Programming Languages

Bouke Westra

1261967

in partial fulfillment of the requirements for the degree of

Master of Science

in Human-Technology Interaction

Supervisors:

dr. ir. Peter A.M. Ruijten-Dodoiu, from the Faculty of Human Technology Interaction at Technical University Eindhoven

dr. ir. Leonardo Montecchi, from the Faculty of Computer Science at Norges Teknisk-Naturvitenskapelige Universitet.

Abstract

Programming languages often are described in text but can also be described as **visual programs**. To address **challenges of scale** in Visual Programming a **Tree-Map Diagram** is utilized as the visualization type. The visualization is applied as **an abstraction** on top of the textual program specification in order to retain function parity. This forms a new design for a visual program, which is **qualitatively tested** and compared with the textual representation. It de-emphasizes issue with syntax and allows to **recognize** rather than recall the program structure.

Table of Contents

Abstract....2
1. Introduction: A Visual Abstraction in Programming....3
  1.1 Visual Programming....5
  1.2 Visualization in support of programming....5
  1.3 Challenges within Visual Programming....7
  1.4 Addressing Visual Programming Challenges....9
2. Designing the Stimuli: Tree-Map abstraction....10
  2.1 Pilot Study....10
  2.2 The Kernel map....11
  2.3 Evaluating the Design....13
3. Method: Testing two sides of the same coin....15
  3.1 Participants & Design....15
  3.2 Material & Procedure....15
  3.3 Analysis....16
4. Results: Content Analysis....17
  4.1. Consistency and standards....17
    4.1.1 Text....17
    4.1.2 Shared text....18
    4.1.3 Visual....18
  4.2. Recognition rather than recall....18
  4.3. Aesthetic and minimalist design....20
5. Discussion....22
  5.1 Limitations and Future work....23
  5.2 Reflections....23
References....24

1. Introduction: Abstractions in Programming

At some point a computer programmer must have thought: “it would be great if I could explain in my language what I want the computer to do!”. However, the computer does not automatically understand human language, because its language is binary. A computer fundamentally exists of connected logic gates which carry out Boolean logic. The AND gate for example takes two inputs, both being either 0 or 1. It then propagates one output based on the two inputs. For the AND gate the output is 1 if both inputs are 1, otherwise it is 0. For the OR gate the output is 1 if either or both inputs are 1. By connecting logic gates the programmer can create different functions, which take one or several inputs and give one output.

Figure 1: Punched card.

To program back in the day meant to edit a long sequence of 0’s and 1’s. Punch cards were used, which encoded 0’s and 1’s by punching holes in the card. This card was then fed to the computer, which would print the output on paper. Figure 1 shows an example of a punch card. Can you understand what it does? Likely not, and so it was an error-prone and slow process to program with punch cards. Essentially the intention of the programmer had to be mapped to binary.

But today we can relatively comfortably program in programming languages, such as Python, by using text. This allows us to program at a higher level with less concern for the hardware of the computer. This is achieved by abstraction. **An abstraction** does not have a physical or concrete existence but needs an exact mapping to the concrete fundamentals of the computer in order to be useful. For example, the programming language Python has to be mapped to binary in order to be executed by the computer hardware. Additionally, abstractions for the computer also have to be exact, because the computer cannot compute with ambiguity. Thus, to create an abstraction in this context, a**n exact mapping** is needed.

Python is referred to as a high-level programming language and co-exists with low-level programming languages, such as C. The high-level refers to the level of abstraction with low-level programming languages having less abstraction and thus being closer to the hardware (Grout, 2008). **A higher-level abstraction** has a certain level of abstraction layers, vaguely being described with higher-level. The programmer requires less detailed knowledge about computer hardware, because of these abstractions (Grout, 2008). These abstractions can get abstracted as well and that is how abstraction layers form. For example, binary can be mapped to hexadecimal, which is a numerical notation that uses 16 as its base compared to a base of 2, which binary uses. Thus, with this abstraction programmers could write 101100 as 2C, which is more compact. Another abstraction could be applied on this abstraction by mapping 2C to the mnemonic ADD. A mnemonic assists memory and in this case helps the programmer remember that 2C is an instruction for addition. This last abstraction was called assembly, which was done automatically by the Assembly programming language, one of the first programming languages (Grout, 2008). It is essentially a mapping of hexadecimal instruction numbers to three-letter mnemonics making it easier to program. This abstracting continues until a **chain of abstractions** is created that allows one to program the computer at a high-level.

As stated before, the programmer has to essentially map their intention to binary in order to program the computer. By using abstractions, they can automate part of this mapping and interact with the computer at a higher-level. Ideally the higher the level of abstraction the closer it is to the cognitive process of the programmer and thus the easier it should be to program. Thus, an axis can be defined with on one end the computer language and the other end the human language (see Figure 2), which is assumed here as the **highest-level programming language,** since it requires no mapping effort from the programmer. Recent Artificial Intelligence tools take human language as input and output program code of the selected programming language and offer insight on what computer programming with human language could be like. However, these tools still need a human reviewer who validates the program, because the mapping from human language to code is probabilistic.

Figure 2: Axis of programming language levels

The programming language can be seen as an interface with on one side the computer and on the other side the programmer. This programmer role could dynamically switch between different humans over time, because the computer has the ability to store the program code. This allows programmers to make incremental edits to the program over time, a process called iteration. Thus, the program must communicate precise binary instructions to the computer, but also the intention to the current programmer. As the programming language becomes higher-level it becomes easier to interpret by the programmer, which makes it easier to collaborate between programmers, who can review the program, give feedback or directly edit it.

Until now human language was stated as the highest-level programming language, however this is not the only option for abstracting programs. **Other paradigms** exist within programming which take a different approach to represent programs.

1.1 Visual Programming

One of these programming paradigms is visual programming. Visual programming differentiates itself by using graphics, which allows a user to specify a program in a **two-dimensional** fashion (or sometimes more) (Myers, 1986). In contrast, plain text is considered one-dimensional since it has to be processed linearly and in order.

Visual programming has been an effort to make the programming processes easier (Myers, 1986). It tends to be a higher level of abstraction and **de-emphasizes syntax** issues (Myers, 1986). Visual programming languages tend to be easy to read, simply by using a clean notation or by leveraging domain-specific notification (Hirzel, 2022). It has been shown that non-programmers can create fairly complex programs with little training (Halbert, 1984). Citizen developers, which are amateur programmers with little professional programming education, may also be easily trained to use a visual programming language as indicated by Scratch (Resnick et al., 2009). It was also found that visual programming has extensive **potential in educating** basic programming concepts (Chun-Yen, 2019). From the above it seems that visual programs are often applied to beginner-level programmers, though semi-developers and professional developers can still readily use them (Hirzel, 2022).

Other interest in visual programming comes from the enthusiasm surrounding Low-Code, which is a cryptic marketing term that comes from the desire to minimize the use of textual programming, preferring techniques that are closer to how people naturally think about their task (Hirzel, 2022). Low-Code encompasses Visual Programming, but also other techniques such as Programming by Demonstration and Programming by Natural Language with a core goal of reducing the need to learn a programming language (Hirzel, 2022). Low-Code enables domain experts to become citizen developers, though professional developers should also be enabled to be **more productive** with Low-code (Hirzel, 2022).

A plethora of visualizations for visual programming exists, but they all let users' program by directly manipulating its visual representation. Two prominent domain-independent visualizations are boxes-and-arrows or interlocking puzzle pieces (Hirzel, 2022). They both have two aspects, one being the **instruction**, represented with the box or puzzle piece, and the second being the **data and control flow**, represented with the arrow or interlocking of pieces. Scratch, an example of interlocking pieces visualization, shows that structured interaction with the visual components improves program creation compared to text, which requires learning and memorizing the grammar first (Resnick et al., 2009).

1.2 Visualization in support of programming

Visualization also supports programming besides the programming task itself. Visualization is used in debugging, documentation, interaction and explanations of patterns. Though none of these efforts directly alter the program, they **support the reasoning** about the program to effectively edit the program later. This activity all falls within the area of Software Visualization.

In software engineering two visualization sub-fields are of particular importance, these being Visual Programming and Program Visualization (Roman & Cox, 1993). Visual programming is extensively covered in books and survey articles in contrast to program visualization (Roman & Cox, 1993) and this may be because of the simple fact that visual programming directly allows to manipulate the code while simultaneously using the visualization advantages. These two fields both utilize the words Visual and Program, but in different orders. This reflects the fields, as visual programming is concerned with **graphical specification** of the program, so the word Visual comes first. Program visualization already has a program in textual form and uses visualization to **explore, monitor or graphically present** the program (Roman & Cox, 1993), so the word Program comes first. However, if we consider a textual program, which is then graphically specified, it will be within both the field of Program Visualization and Visual Programming. If we utilize a taxonomy of Software Visualization it is specified that Visual programming is encompassed by Programming Visualization (Price & Baecker, Small, 1993) and at the intersections of Static Code Visualization and Static Data Visualization (see Figure 3). Taxonomies such as these are helpful, but there exist multiple different variations and people utilize the terms differently (Price & Baecker, Small, 1993). I would argue this is indicative of the many different approaches to visualization, which often also overlap with textual programming. For example, until now it has been stated that visuals are separate from text, but text is interpreted by the visual system, so in that sense it is definitely visual. But as said before, it is processed as one-dimension, which is about its function, which can be **disconnected** from how it is represented. This disconnect is also highlighted in the fact that visual programming is done with so-called visual programming languages. However, is it correct if we discuss them still as languages, as visual programming languages are mostly static diagramming systems, which are relatively more different from spoken languages (Ware, 2021) than textual programming languages.

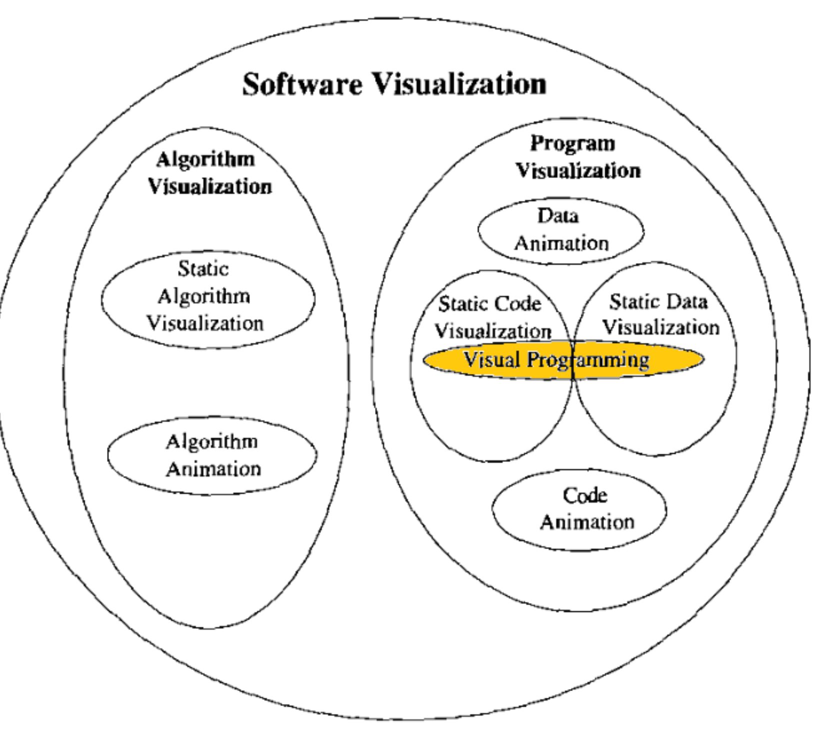


Figure 3: Venn diagram for taxonomy in Software Visualization. Size is irrelevant and the only intersection is shown for visual programming. (Price & Baecker, Small, 1993)

I would argue that, before continuing, it is important to discuss the **term program**. A program consists of a set of related activities which together create a particular result. An activity necessarily takes time to perform and thus there is a **sequence** to the activities. If we choose to verbally speak about a program, we are naturally forced to choose an order to explain the activities in the program. This order then also translates to text and makes it a natural way to **describe** the program in language. Visuals, however, are interpreted in **one fixation** and thus do not naturally have a sequence but are actually **parallel**. Another way of saying it is that visuals are s**patially-encoded**, whereas text (speech) is **time-**

encoded. Thus, I argue that even though a visual programming language is represented more visually, it still operates as a language, and at some point, must be translated to binary language for the computer to understand. There is again a **disconnect** between the **functioning** and the **representation**.

In the history of Software Engineering, visualization has played an important role, even if the programs themselves are written in text. It is not either text or visuals, but both have their use cases. Information should be displayed in the most **appropriate medium** (Ware, 2021). Say for example you want to explain **a relationship structure**. In the text it would be:

"

A is connected to B.

A is connected to C.

C is connected to D.

C is connected to E.

"

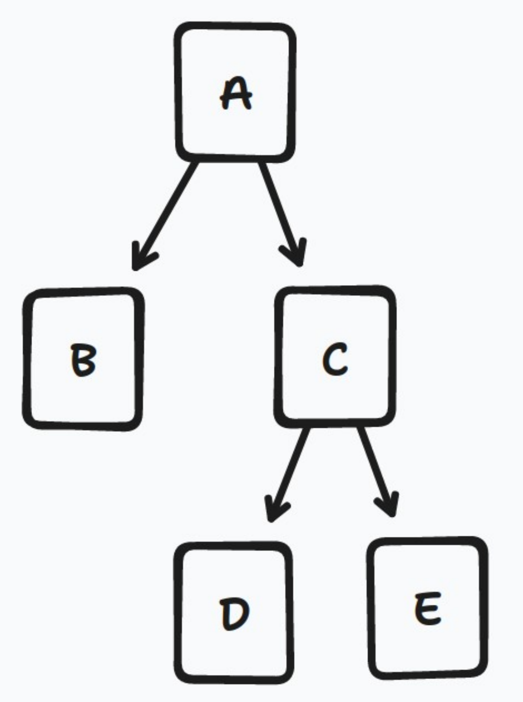


Figure 4: Tree Diagram of the relationship structure

Here each sentence must be read individually, and the complete relationship structure must be **cognitively constructed**. This pattern of relationships is far more clearly expressed in a diagram (see Figure 4). Diagrams are more **suited to express structural relationships** among program elements, whereas words are better suited to **express detailed procedural logic** (Ware, 2021).

It is proposed that visualization can amplify cognition in six major ways (Card, Akinglay and Shnedierman, 1999).

1. By increasing the memory and processing resources available to the users.
2. By reducing the search for information,
3. By using visual representations to enhance the detection of patterns,
4. By enabling perceptual inference operations,
5. By using perceptual attention mechanisms for monitoring,
6. By encoding information in a manipulable medium

Programming is mainly a cognitive task and thus all these ways of cognitive amplification can serve the programmer. It is however not yet clear how to utilize all these points well for visual programming.

1.3 Challenges within Visual Programming

Visual programming systems have been attractive to use, but still share some unsolved challenges (Myers, 1986). In **small use cases** Visual programming has appeal but **scaling it up** effectively to be used in larger programs is not straightforward (Burnet, et al., 1995). It is noted that almost all visual representations are physically larger than the text they replace (Myers, 1986). Within this time new visualizations have been developed, but compactness is still a challenge with the visual notation still

taking up a lot of **screen real estate** if details are not elided (Hirzel, 2022). This is difficult to resolve as a text character only needs a size at which it is discernible, whereas visuals have more factors impacting size.

Next to that, the Visual Program will inevitably be **compared to** a textual program in many aspects as people will be faced with the question whether to use a visual program. Comparisons are made on **functionality, efficiency, program structure and static representation**, which are often unfavorable for visual programming languages (Myers, 1986). Notably virtually no visual programs have a place for comments (Myers, 1986).

Also, it should be noted that the text used in programming languages is not displayed as one-dimensional as human text. It utilizes different spatial formatting, and its syntax components can be highlighted with colors in code editors. **Indentation**, which refers to the space before a line of text, is used to improve readability in most programming languages, though in the programming language Python it is functionally used to indicate blocks of code. I would argue this indentation utilizes the **second dimension** more, as the empty space in front of the text is used to make the **program structure** more visible. Nevertheless, this arguably still does not quite make it as two-dimensional as the boxes-and-arrows or interlocking-puzzle pieces visualizations.

Take the following program: “fun {Max Number1 Number2} if Number1 > Number2 then Number1 else Number2 end end”. This is a function in the programming language Oz, which takes two numbers and returns the bigger one of the two. It makes sense for humans but can also be relatively easily translated to Boolean logic that the computer understands. The **formatting** is displayed as traditional text, being just one sequence ,which is wrapped to the next line around if you reach the end of the horizontal space. This program could also be displayed with indentation:

“
```
fun {Max Number1 Number2}
     If Number1 > Number2 then
          Number1
     else
          Number2
     end
end
```
“
This displays the **hierarchical structure** of the program more clearly but would still operate the same as the previous example. This shows that we can **utilize the second dimension** better to quickly understand the program. This is what we see throughout all developments in programming; we tend to add more dimension to it, as it enhances our understanding. That is what the field of visual programming is striving for as well, but simply on a **bigger scale**. The question is how we can utilize this second dimension well and also make it scalable?

1.4 Addressing Visual Programming Challenges

The Tree Diagram in Figure 4, which is clearly a two-dimensional display, is actually not the only way to display a hierarchical structure. A Tree-Map (see Figure 5) is more **compact and space-efficient** in displaying hierarchies and can give a quick overview of the hierarchical structure. It was originally developed as a way of visualizing a large directory structure of files on a computer, without taking up too much space on the screen (Johnson & Shneiderman 1991). This a visualization **made for scale**, though a downside is that it does not show the hierarchical levels as clearly, which the Tree diagram uses rows for. However, this gives the freedom to choose the layout of the blocks, only being constrained by having no overlap with other areas. This visualization can represent the hierarchical structure that is already displayed with the indentation of blocks of code. The visualization will require more sophistication than the Tree-Map can provide, so just the layout structure, which is based on the **property of containment** (Johnson & Shneiderman 1991), can be used and will be extended as needed.

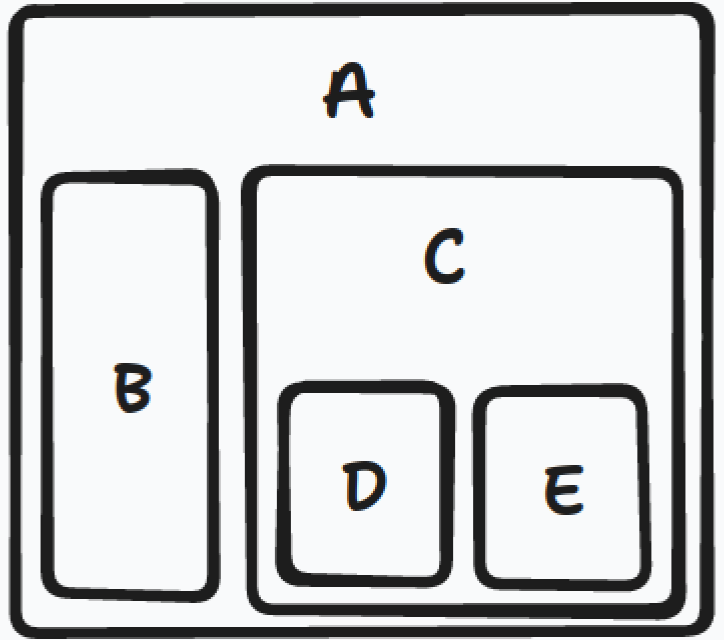


Figure 5: Tree-Map of the relational structure.

To address challenges with **comparisons** of the functionality of textual programming languages, **a mapping** is proposed from the kernel language of a textual programming language to a visual representation. This part of the design was done in collaboration with the second supervisor from the Computer Science department. The kernel language defines the semantics of a programming language and consists of a set of rules that can be individually mapped to a visual component. This means all the functionality of the textual programming language is **inherited** in the visual program and the textual and visual representations co-exist. Textual features such as Search or Version Control would still be available. The visual program would be **another abstraction** on top of the chain of abstractions already present in programming languages and not something separate from a textual programming language.

Notably, the **first** solution has more of a **human nature**, whereas the **second** solution has more of a **technical nature**. To test these new solutions the following research question is applied:

> To what extent does visualization, with the property of containment, influence people's understanding of a software program's functionality compared to traditional textual programming?

2. Designing the Stimuli: Tree-Map abstraction

To test this new visualization for a programming language it has to be designed first. Oz, a multi-paradigm language, is chosen as the textual programming language **basis**. It is somewhat of an experimental programming language, but well documented in the book “Concepts, Techniques, and Models of Computer Programming” (Van Roy, Peter & Harridan). It has a **compact** kernel language to describe the whole language and is on par in its **expressiveness** as other programming languages. Beside the above reasons, it was also chosen because of **familiarity** with it. The kernel language consists of a set of statements and rules (See Table 1, Column 1) and these are **mapped** to a visual design. A web tool called tldraw was used to design the visualizations. A lot of effort went into finding a scalable design and creating a technical basis.

2.1 Pilot Study

To **validate** the visual design before the real study, a pilot study was performed to see what other students with **equal constraints** would design and contrast that with the state of the visual programming language (see Figure 6) at that time.

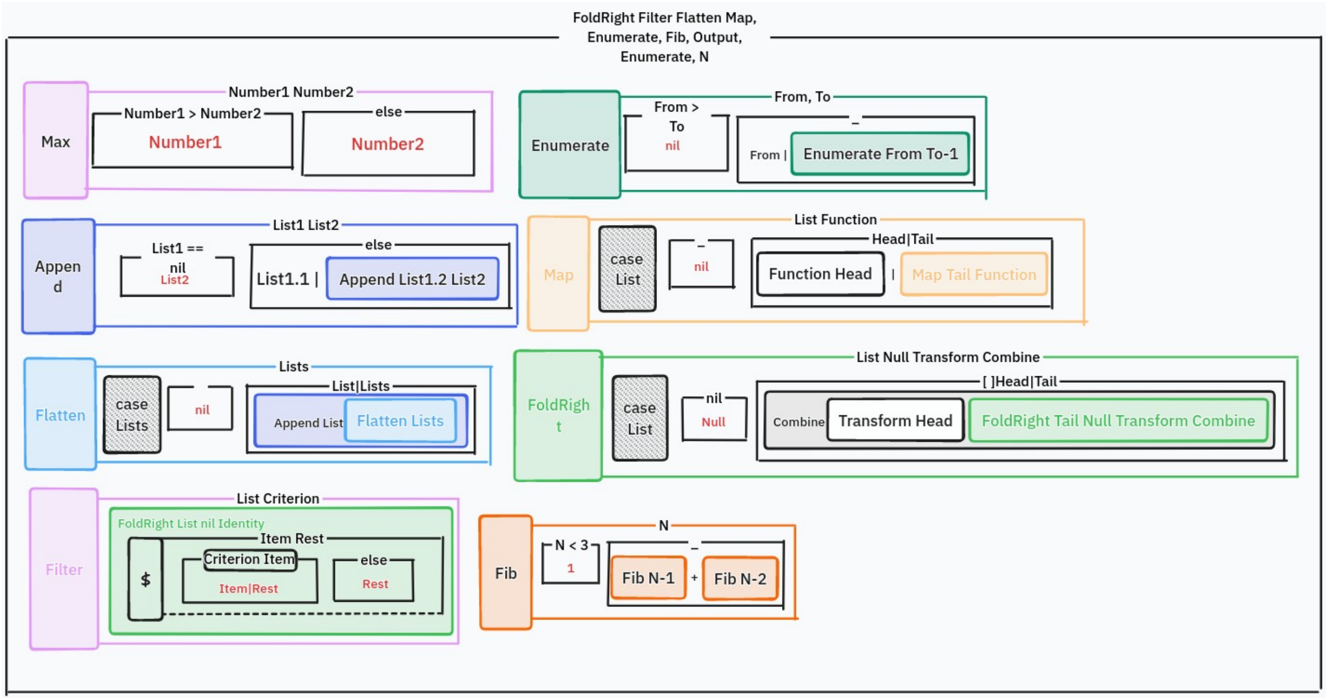


Figure 6: Status of the visual program design at the time of the pilot study.

Two male students participated with minimal experience in visual programming with one having practiced visual diagramming for a course. They were tasked to design their own visual programming language with the same constraints as the current design. These constraints are defined by the kernel language of Oz and choosing a two-dimensional visualization type. The desire for improving cognitive support came quite naturally to the participants, but the benefits of visualization were anyway introduced. Participants are using **pen & paper** to draw the visualization, so as to not to constrain the

visual expression of participants. A slide presentation was used to guide and display the tasks in the experiment. It starts with the participant introducing its background in functional programming and visualizations of programs. They were introduced to the goal of making their own visualization for a couple predefined functional programs and that they had to **verbally explain their thought process**. The programs included the Max, Factorial, Enumerate, Fibonacci, Map, Flatten, Fold Right and Filter functions. The predefined programs were all single functions with each next function introducing a more sophisticated program structure. Thus, the challenge was slowly increased for the participants, giving them the opportunity to iterate on their design and increase its versatility. A short introduction is given on how visualization can support cognition after which the kernel language of Oz is shown, which they will have mapped to a visualization at the end of the experiment for a complete programming language. This sets the desired output and shows them that all the components have to work together in the end. After being done with their drawing tasks, the participants were shown my design and together we compared the designs while being informed of the purpose of the pilot. The drawings were collected as measurements.

Both participants **utilized arrows** in their design, and both came across **issues when scaling** up their designs. One participant heavily tried to replace the arrows with another representation, but only partially exceeded. He actually applied the property of containment sometimes. When shown my design he was convinced in one fixation as the design had **no arrows**. The other participant preferred the textual representation and arrows more in his design. This resulted in a visualization more like the Tree-Diagram. When shown my visualization he was not convinced about the scalability and would still likely prefer the textual representation better. He noted a disconnect in the **data-flow** of the components which was clearer indicated with the arrows. This made it harder to view the architecture of the program for him.

Furthermore, both participants started with writing down the function name and making an encapsulation by drawing a border with the name inside at the top. This made me **move the name** to the top instead of to the left. It also made sense space-wise with the horizontal nature of text. This new position overlapped too much with the parameters at the border and thus they were **integrated** in the same line, which is conveniently also how Oz did it. Both participants showed a return of the function with **a line/arrow exiting** the encapsulation, though this was not applied in favor of an **implicit return**, which is also how Oz implements it. Participants did not have a preference for how the colors were used. So, colors were changed to highlight the different components instead of identifying the different functions, which is more **consistent** with syntax highlighting in textual programming languages and scales better.

2.2 The Kernel map

The **kernel language** of Oz and the visual mapping is displayed in Table 1 & 2. In the first column the values for an **Oz statement**(<s>) are given in textual form. Notice that statements can be contained within statements if they contain a <s>. In the second column the **visual equivalences** are given and in the third column the description for both columns. Not all textual Oz statements are mapped to a visual representation as text and visuals both have their use cases.

A visual components consist out of maximally **three parts**, the header, the body and the body membrane. The different parts are applied to different statements to form different components. The header is a part for **naming**. The body is a part to insert the **program logic** (text). The membrane is a part that **filters inputs**. Only the body and membrane have the property of containment and thus can hold components. These parts are abstractions of statements which operates differently in Oz but with a **similar structure**. Some statements need multiple visual components.

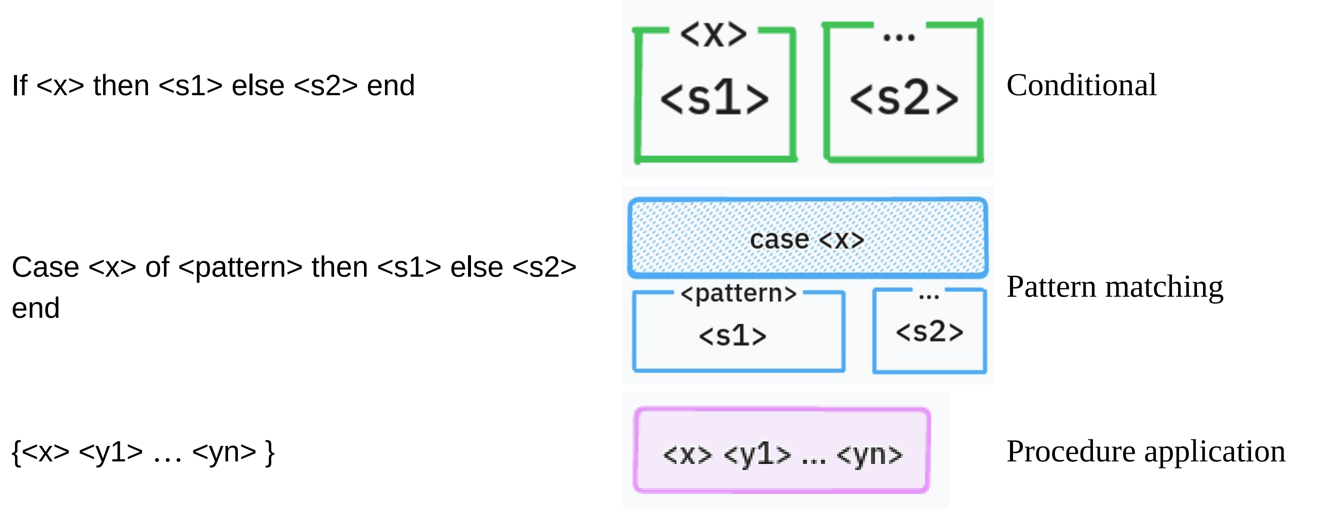
The design is almost **language agnostic**, with only the case keyword remaining for clarity of what the component is. The `else keyword was replaced with `...` for the same reason.

Ware (2021) proposes some **principles**, which state how to utilize graphics and text well for programming. Statement one states that methods based on national language should be used to **express detailed** program logic. Statement two states that graphical elements should be used to show **graphical relationships**. Both statements are complementary applied in the design. The structural syntax of Oz, such as `if`, `then`, `else` and `end` is mapped to a graphical element. The program logic is still expressed in text, which can for example be inserted at <s1> and <s2> in the if-else statement If you take a tree structure in mind, the leaves of the tree will be text and the branches will be the graphical elements.

Statement three states that methods based on natural language should be used to represent **abstract concepts**. This coincides with the possibility to create textual names for variables or functions, which are abstractions of their values. Statement 5 states that **explanatory text** should be as close as possible to the related parts of a diagram. This is applied, since the naming is directly placed within the boxes.

Table 1: Visual Mapping of the declarative kernel language of Oz.

Oz Statement,	Visual Mapping	Description
skip		Empty statement
<s1> <s2>	<s1> <s2>	Statement sequence
Local <x> in <s> end	<x> <s>	Variable creation
<x1> = <x1>	<x1> <x2>	Variable-variable binding
<x1> = <v>	<x1> <v>	Value creation



If <x> then <s1> else <s2> end	<x> <s1> … <s2>	Conditional
Case <x> of <pattern> then <s1> else <s2> end	case <x> <pattern> <s1> … <s2>	Pattern matching
{<x> <y1> … <yn> }	<x> <y1> ... <yn>	Procedure application

Table 2: Visual mapping of the value expression in the declarative kernel language of Oz.

Value	Visual Mapping	Oz
<v>	<v>	<number> | <record> | <procedure>
<number>	<number>	<int> | <float>
<record>, <pattern>	<record>, <pattern>	<literal> | <literal>(<feature1>: <x1> … <feature>n : <xn>)
<procedure>	$ <x1> ... <xn> <s>	proc { $ <x1> · · · <xn> } <s> end
<literal>	<literal>	<atom> | <bool>
<feature>	<feature>	<atom> | <bool> | <int>
<bool>	<bool>	**true** | **false**

These visual components can be **put together** by containment to form **any program** (for examples, see Figure 7).

2.3 Evaluating the Design

The goal of the visualization is to support the programming process, so to evaluate this new visualization Nielsen's (2020) **Usability Heuristics** will be used.

The first applicable theme is "**Consistency and standards**", because there is a historical situation where people are trained in using textual programming languages. These introduce certain **expectations** in usage of programming languages and thus the visual designs should be evaluated on these standards. They will likely **relate** to the visual design from a textual programming standpoint.

Notably, the syntax of Oz, the used textual programming language, is unusual among the popular programming languages, though the visualization should de-emphasize these syntax issues. Lastly the the visual programs should be relatable to the equivalent textual programs and thus indicate that there is a map.

The theme “**Recognition rather than recall**” applies to the different properties of text and visuals. It is expected that visuals support recognition more and text supports recall better. The program structure is displayed in two dimensions to support **seeing the scope** of the code blocks more clearly, in comparison to text. This serves as a **memory aid**, which reduces mental load and enables programmers to focus more on the “content” of the program.

“**Aesthetic and minimalist design**” is about **redundancy** in syntax and visual elements. It is expected that a minimalist design will reduce the visual complexity of the visualization, which will make it easier to interpret. All the information will be **relevant** and not compete with irrelevant information. The design is reduced to its essentials and users utilize all aspects of the design. The participants can take the minimal amount of cues in shape, color and layout to interpret the visual program. The visuals consist mostly of boxes with text as the content and not much else.

Not all heuristics could be used. The themes “Visibility of system status”, “User control and freedom”, “Error prevention, flexibility and efficiency of use”, “Help users recognize, diagnose, and recover from errors” do not have an effect, because they focus on responding to users, but the design will only be a visualization and thus **static** and not interactive. The theme “Match between system and the real world” does not have a clear use case in programming, because the programming system is an abstraction of the computer system. The theme “Help and Documentation” states that it is best if the system doesn't need any extra explanation. Since the design is new it makes sense to test it **without documentation** and see if users will get it by themselves. It is expected that the experienced programming users will not need any help by utilizing the text in the design, because visuals are not self-explanatory. More Usability Heuristics will apply once the design gets further implemented as a programming language.

3. Method: Testing two sides of the same coin

3.1 Participants & Design

Four people participated in **qualitative** interview study where they reflected on the differences between the visual and textual programs while following a **think-out-loud protocol**. The stimuli were presented **within subjects** and consisted of equivalent Oz programs in textual and visuals representations. It was randomly decided which representation went first for each program and participant. The study was approved by the institutional ethical review board (HTI Ethical Review Board—experiment ID 1838). The participants (see Table 3) were **recruited** via electronic mail from a pool of students who did the course Programming Languages at the Norwegian University of Science and Technology. They were required to have some functional **programming language experience** to understand the Oz programs, which students in this course have been exposed to. The age ranged from 23 till 27 with a mean age of 25 and they consisted out of 3 males and 1 female. They all had experience with Oz from the course but were also knowledgeable about other programming languages such as Python. They did not have any significant visual programming experience but were familiar with visualizations for documentation and half of them had experience with diagram modeling.

3.2 Material & Procedure

The studies were conducted in the months April and May in 2023. The interviews started with the participants reading and signing the informed consent. The **demographics** were recorded with the participant speaking in the laptop microphone, which doubled as a test for the microphone and the surroundings, which were not always ideal. The microphone used was from a laptop, which was also displaying a presentation, which contained the steps of the procedure including the visual and textual programs, demographics and reflection. Thus, participants would face the direction of the microphone naturally when trying to explain the program they saw on the screen. Participants were reminded to use **verbal descriptors** (for example, the upper box) and not use visual gestures (for example, pointing) as that would not be transcribe-able during the experiment.

It was explained that they get shown textual and visual programs in a random order and that they would have to **think-out-loud** by verbally explaining how they interpret the programs functionality from the text or visual representation for each mental step. After they had seen the two representations of the same program, they were asked to relate the two. This task was done for **8 programs**, which includes the following functions: Max, Append, Length, Map, Flatten, Fold Right, Filter. The order of the functions introduced structure with an i**ncreasing complexity**. Max introduced function declaration, if and else and returning values. Append introduced function calls. Length introduces case. Map introduced two function calls in the same cell. Flatten introduced a function call with as argument a function call. Fold Right combined the new things from Map and Flatten. Lastly, Filter introduced the anonymous function as an argument. The visual design can be seen in Figure 7.

At the end the participant reflected about the two representations. This study was aimed to be 60 minutes. Participants gave their written informed consent and were compensated 116 Norwegian Krone, which was equivalent to 10, - euros at the time of the experiment.

The recordings were **stored locally** on the laptop and were locally **transcribed** with whisper, a general-purpose speech recognition model from OpenAI, using the large language model option. The recordings were deleted after encoding the themes, as the varying quality of the transcriptions sometimes required to reference the original recording.

3.3 Analysis

A **content analysis** was performed with the relevant **usability heuristics**, presented in the introduction, as themes. It was inspired by thematic analysis (Clarke & Braun, 2006), but with a simpler process by having predefined themes, which capture the researchers expectations. These expectations determine what you look for in the transcripts and give more focus.

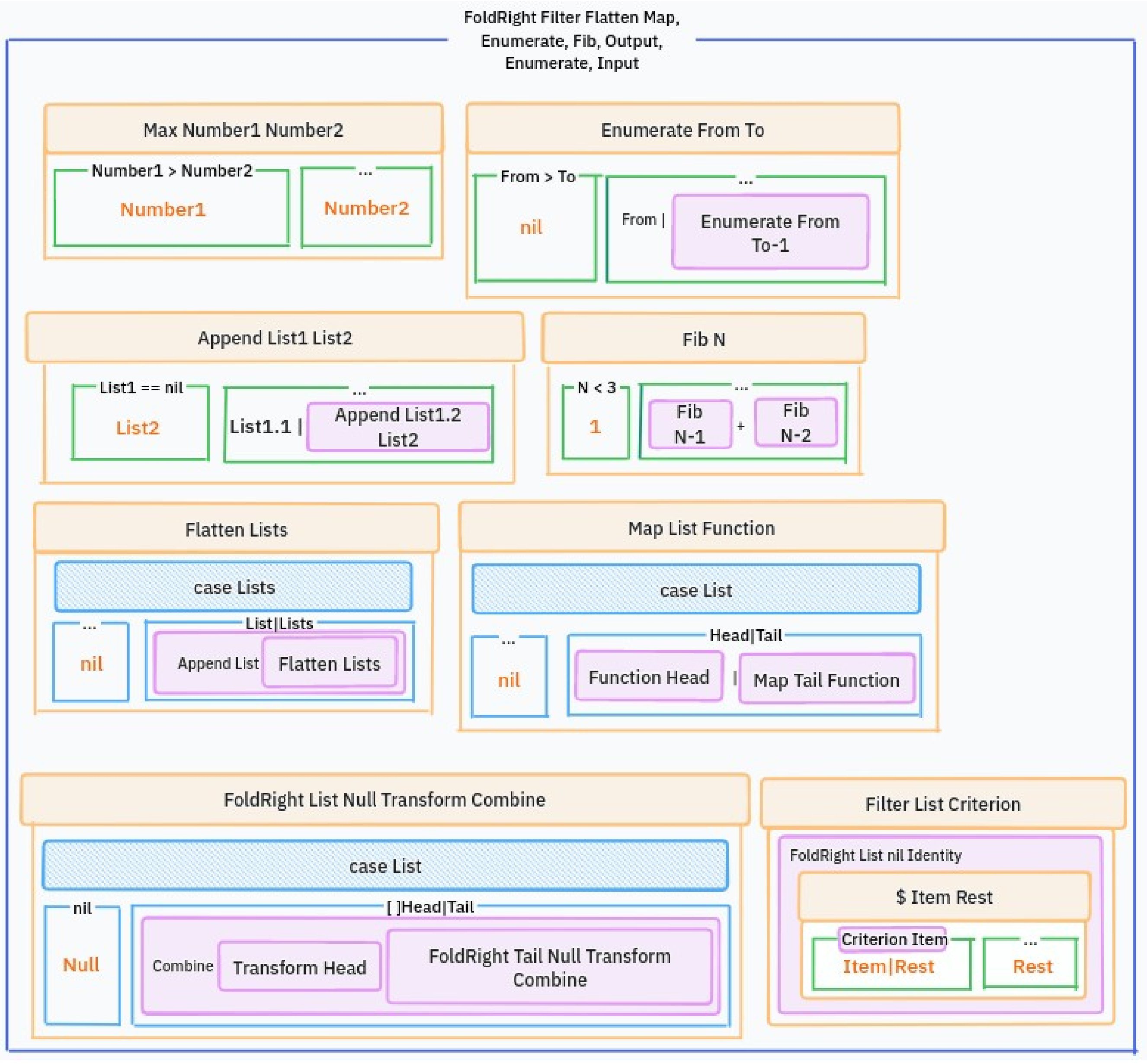


Figure 7: Final design of the visual programming language.

4. Results: Content Analysis

Participants experiment duration's ranged between 52 minutes and 92 minutes with a mean of 73 minutes. The analysis was performed by highlighting the three different themes with three different colors in the transcriptions and correcting any mistakes in the automated transcription of decent quality by referencing the audio track. The consistency and standards theme appeared on three different fronts, which were split later. The function names and the representation (text/visual) were made bold to quickly navigate the transcriptions. Much of the program explanation could be skipped, as it was more important how they understood it from the notation, which participants usually stated after the program explanation. There was some overlap between the themes, so it was categorized based on what the strongest theme was. The highlights of all the participants were brought together within the themes and **representative quotes** were selected as arguments.

4.1. Consistency and standards

Consistency and standards apply in three fronts. The first is the consistency from textual programming languages to Oz. Then there is the shared category, which functionality present in both representations. This mainly consists of such as names for variables or functions, which should be consistent with names already used in other programming languages. Lastly the visuals, they should have a consistent map with the Oz text.

4.1.1 Oz

Some of Oz's syntax was perceived **unique to Oz** with participants saying "S*o this is a very Oz syntax*" or "*And the brackets? Yeah, that's like OZ craziness*". This brought some issues with participants recalling the meaning of Oz's syntax with for example *"Now. This is some syntactic stuff. I don't remember at all. I don't remember what the square brackets do in oz. At all."* or "*I mean there's a dollar sign so I forgot that it was anonymous function I think it was.*". Some of the programs were recognized by the participants, because it were programs from the course saying "*I think, again, because I took the course, I also have created this function as an exercise. Yeah. So yeah, I'm used to also this way in which Oz works with tail recursion.*", which made the textual program faster to understand.

There were some **issues separating** function names and parameters of a function, which in Oz syntax is noted as `{FunctionName Parameter1, ….}`, but participants said: "*I think that is one thing with Oz maybe if we don't have anything that kind of separates the FoldRight from the parameters we have*" or "*I think there's a function. And then I would consider the parameters. Something separate.*". This was especially obvious in the visual FoldRight function, which had 4 arguments. It is not clear why the text suffered less, but one explanation could that recall is better in the context of text compared to the visuals, which rely more on recognition. One participant proposed a solution to make the function name bold.

4.1.2 Shared

Participants relied heavily on the **naming** of the functions to have a first idea of what they do. Though this text was in both representation, the naming was used more within the visual representation. Participants said: "*So from the name, it's already kind of obvious that there's an if statement. I read max and I read two arguments and I expect it then to return the larger of the two.*" or "*And then from from the naming, I can infer that max is maybe the name of some operation and number one and number two arguments*". This can be for two reasons, first the names are the most prominent text in the visual representation, whereas in text, it is one of the many words. The second reason is that the visual meaning is not self-explanatory, thus participants use the names as hints to figure out the meanings.

The implicit returning within Oz and the visuals was assumed naturally by all participants from the first function. For the text one participant mentioned: "*And again, also it's a bit weird. So it doesn't say return. But I know Oz.*". Returning a value seemed to be naturally implied by calling a function name.

4.1.3 Visual

Oz had some unusual syntax, but in the visualization, this was mostly abstracted and thus people did not describe the visual representation as weird anymore. This emphasizes the reduction of syntax. They would rather comment on unclear points in the visualization and how the structure would cleared or less clear in the text.

The meaning of ". . ." in the if-else statement was not quite as self-explanatory as "else", with a participants saying: "*I couldn't automatically associate the, yeah, the three dots with the else instead*". Other participants also "guessed" correctly what it meant "*I'm guessing dot dot dot means else*", but preferred to still reference it as "else" afterwards. They inferred the correct meaning from the context, which often comprised of the naming and the position among other visual elements. After that participants referred to \`...\` as "else", which is how it is stated in Oz, with one participant saying "*And then the else case represented by these three dots falling to zero.*".

The visual components remained constant among themselves and thus participants could rely on their consistent meaning with sayings such as "*So again, I see a box with a title. So I know that that's a function declaration or definition.*"

The visuals were quite naturally mapped to Oz, because in verbalizing the visuals the Oz syntax was used with participants saying for example "*So again, I see a box with a title. So I know that that's a function declaration or definition.*". This quote also highlights the consistency in the meaning of visual components. The structure was also relatable with a saying of :"*Yeah, if I compare it[text] back to the visual example. The white space is your boxes. You're kind of enclosing it into. Some visual constructs."*

4.2. Recognition rather than recall

This theme compares the recognition of programs, which minimizes the user's memory load. The visuals generally **support recognition more** compared to text. This reduces cognitive effort and thus

the program is perceived as easier. Grasping the scopes within a program, is required information when understanding program and thus should be easily retrievable.

Colors were recognized as relating to programming concepts with sayings such as "*I mean, yeah, I think I'm confirmed that purple represents recursion*" or "I*'m tying those together. The same color." or "And I know that pink is the color of function*" or "T*he color was very nice with categorizing.*". This was usually learned quickly.

Learning the visual syntax proceeded with basically no help, though still required some effort, "*Might just be the case that working with. A new type of syntax is. Also difficult. You just get used to it and then you start. Recognizing the patterns.",* but they learned it quickly.

The textual programs had different formatting expectations to the participants but the visual design only had one format. Participants used indentation to get the structure with one saying "*I mean, to me, the indentation kind of reads as. Just a way to structure things."*. However, when this indentation was different from their expectation, then they had to construct the structure themselves with one saying: "*You have to construct this scope yourself. While reading the code and that's not as easy.*". This was contrasted with the visual representation with one participant saying; "Y*ou kind of have to draw out the box yourself.*", which nicely contrast the text and visual representations. Navigation was easier too with participants able to follow the outline of boxes to keep track of their scope. The visual scope is fully encapsulated by a border and thus wherever your eye travels, you will see that border and know that it changed scope. The scope is recognized in the visual and recalled in text.

They started making a significant separation between content and structure, saying "*And also I think again, the moment you see where the, I don't know how to say it's like the meat of the function lies. Yeah, then significant, nothing significant but yeah, like the case where you can easily solve it what you're really interested in is the main part.*".

The structure and content seem to be processed differently with participants saying "*it's easy to glance over. Or like, you can easily skip to like the important parts*" or "*But like it's[text] another kind of. Reading like extra effort. Yeah I mean it's kind of going back into the details.*" or "*But I think I mean obviously my eyes traveled to the, you know, the purple boxes but why because it's eye catching"*.

A difference between the two processes seemed the speed with sayings such as "*So I pretty quickly just skim right down to the purple boxes.*" or saying "*And it's also really fast.* "or "*Recognizing it's also pretty quick.*" or *."* "*It's kind of obvious when you see visually where the different branches are and where you can attempt to, and you can just skip a lot of it.*" or "*Yeah, I think the strength with the visual representation is that it becomes very immediately clear to you the structure of like the function that you're looking at."*. These are many different occasions where the speed of visuals is used.

Color and shape were differently interpreted "*But the color for the syntactical meaning takes me a bit longer to parse. The shape is more important than the color. But also it, they need to be different. So I think the background color would be nice."* with color being recalled and shape being recognized.

Notably the lack of an indication of required number of arguments became more obvious in the visualization with people saying "*And then each of these functions, it's not as if they just took one argument. There's a different number of arguments as well. So you have to then remember, oh yeah, this*

function took four arguments this function took three arguments this function.". This might be because visuals use more recognition and thus this occasion where recall is needed stands out more.

4.3. Aesthetic and minimalist design

This is about any perceived redundancy in the designs of Oz and the visual design and whether the design gets too complicated.

The design consisted mostly of box shapes. This worked well for participants on a small scale saying *"Two cases easy. They're nicely boxed. And you can kind of. Easily distinguish them again."*, though on a bigger scale it started to become less easy saying *"Like the moment they got a bit too complicated, then it was. Yeah, it was less easy to see."*. This was particularly the case with the Filter function, which was the last and most complicated one, with one of them saying "I *think up until this filter list example, it was pretty easy."*.

Participants mentioned that colors would overlap if the same-colored component would nest too often in itself, saying "*Pattern matching within pattern matching within pattern matching. Within pattern matching. Then it might be difficult again if they are the same color.*"or *"You can't use color to distinguish between them. Because you're already using color to distinguish what kind of box it is."* or "*I think the boxes I think that many colors all of a sudden. I think, really until now the diagram has been working well, but the moment it gets into a very, very complicated. Then the, then the colors and the boxes can get a bit intense. And then I think it's best to just sort through the text.*". It was proposed to try a different way of differentiating than using color with a participant saying "*You could also start playing with shapes. Rather than just using boxes.*".

Though color was not always regarded as significant with participants saying "*Green is conditional yellow is function definitions. It's kind of, it feels arbitrary."* or *"Again, I see a box with body and the head, and it's orange, by the way."*, it still contributed to the design with participants saying "*And of course, like when you see just a big blob of text, it's way less interesting, I guess, like the brain really likes the bright colors. So it's easier to stare at this for longer, I guess.*". It did help with grouping elements with participant saying "*I'm tying those together. The same color.*".

Too many levels of containment also made the box shape less clear, participants said *"I think the weakness with that is that the moment we had a very complicated function where three different functions were nested inside. Then it became a bit too. It became a bit too complicated then you had to look inside a box inside a box."*. Another idea was proposed: *"I really like to draw it in this branch structure. I think it's just the fact that you're representing everything by a box. Yeah, and the colors really helped distinguish that but maybe sometimes because we end up with a case with so many boxes."*

For the Oz syntax some redundancy was perceived with participants saying *"Yeah, I think case or if else these two statements are very similar. As in they're just different ways of saying the same thing."*or *"When I'm reading it, I don't really look at the "then" because I know what if does."*. This would carry over to visual representation where with for example: *"And I would personally remove the square braces of the FoldRight, for example, in the visual representation."* and mentioning that the "case"

keyword was not required in the visualization. The “end” keyword was often also ignored in explanation and is lacking in the transcriptions.

There was one component, the “case”, which had a patterned background to indicate that it does pattern matching. This was missed by all participants with them saying “*But here, I see that it's somewhat different because it's shaded. There's a pattern matching. Oh, that's funny.*” after being helped to see it.

5. Discussion

Participants show that it is quite easy to infer the meaning of the visual components and indicate that it has potential as a educational tool, but still doubt about its scalability and indicate some lacking areas of the visualization.

It is difficult to draw inferences from the results as the Usability Heuristics are still very general and are not specifically specified for programming languages. The visualization is also not yet fully optimized and could be iterated upon after this study. However, there are some fundamental aspects to this implementation that worked, which can be replicated.

The first is the technical mapping of the textual program to the visual program. This approach works well and gives a solid foundation to the visual programming language. The fixed set of statements from the kernel language is restricting but it also gives the visualization a natural structure and when participant have to explain the visual design, they can use verbal language, with terms from the textual representation. The visual and text are not separate but can be used based on the programmers need. If in doubt about the program, participants could also still look at the text as the validation. This hints that text is considered more fundamental for the participants too. A mapping from text to visuals makes sense for the implementation and for the programmers too.

The **consistency and standards** were perceived unusual for the Oz syntax, but less so for the visual design, indicating de-emphasis on the syntax . The visual language has the potential to be entirely language agnostic. Names served a documentation purpose and were used a lot by participants to get a first idea of what the program would do, which was all what was needed to learn this unfamiliar visual syntax.

Recognition was supported more by the visualization compared to text. This supports Ware's (2021) guidelines that graphics can better show visual structures. Recognition of the program structure was faster for participants. With the possibility today to generate code fast, this visualization can help to quickly validate the generated code, which can still produce errors. The visual representation can also be used in education, since the participants were already quite effectively explaining the visual programs to me. I could look at the diagram and still follow them verbally but with text it is harder to read it and verbally follow what they say.

The **aesthetics and minimalism** in the design was quite good already. It was intentionally designed with as little "meat" on it to have focused design to test. There were only nick-picks on a small scale, but on the bigger scale, there were some fundamental issues. The **box shape** was reused a lot and became unclear with too much containment, which makes sense as borders can lay very close. It could make sense to only display backgrounds and not the border, as is possible in Tree-Diagrams. The **color application** of highlighting different components had some edge cases in the bigger scale programs. In text they also syntax highlighting to indicate components, but text cannot be nested and thus the stated issue of having the same-colored components nested, never came up. Other color use cases could apply, with one participant suggestion to indicate the depth. It can also be dynamically defined depending on what the user needs, giving the user the choice. There are many possibilities here which can be directly integrated in the visualization, such as heat-maps of often used components or a unique color identifier

for each function. Color can act as a sort of visual filter. It was also suggested to change the shape, though the rectangle should remains an efficient way to encapsulate text, though the border could be dotted or dashed to indicate a different shape. Changing to a patterned background was not noticeable and thus should be used for subtle hints.

The visual design also stands on its own and can be used complementary to the textual representation. The method of the visual design can be relatively easily applied to other languages beside Oz, as long as they have a hierarchical structure. The if and else for example exists in many other languages with same encapsulating structure. This would open up more possibilities and reach a larger audience. The the visualization could even look the same for the same program structures, but with different languages under the hood. The layout of the Tree-Map is also responsive in its layout, meaning it can adapt to different screen sizes, which opens up visual programming to more devices, such as mobile phones, which also works with the compactness of the visualization. The visual program design format will always look the same, in contrast too text. This will make interpretation more consistent and nullify formatting work.

Participants understood the structure of the program generally quicker and easier even with their training in textual programming. They could use recognition to observe the scope instead of recall to construct the scope. This together will help to make the visual programming language more usable.

Taken all together, the most important learning is that the structural syntax of programming languages can abstracted further with visualization.

5.1 Limitations and Future work

The tested visual design, with the property of containment, is potent for this abstraction on a big scale, but needs more refinement to make the design more minimalist and minimize competition of information. If the colors started overlapping or if the boxes were getting too close to each other, then the visualization became harder to interpret. This may have led participants to guess that the visualization would not scale well. This is not necessarily the case as these are edge cases, containing the same component in itself is uncommon and usually a block of code consists out of more than one statement. Though this does indicate an inefficiency in the design, which is not as visible when you look at its components in solitude. The combinations of them can be infinite and so it is good to get edge cases of the design to improve it. In the future the design can be improved to perform well in these edge cases, possibly by using background more instead of border.

Depending on the structure and the goal the colors can support or inhibit your goal. In the design the colors are applied beforehand with the goal of separating components. This did not work for all use cases of the participant and gave some doubt about the scalability. So, a single use of color is not yet clear and multiple uses of color could be more useful. Thus, the color can be dynamically applied, which will also make the design more accessible for impaired color perception. This can be explored in the future, utilize the structure and apply different color applications to explore the use cases.

Finding a better way to compare visual and textual programs and having a taxonomy could also benefits further analysis. The Usability Heuristics only generally apply, but don’t give any support for contrasting the representations. Future work could have a method, which handles this better.

It is concluded that neither the textual or visual form is better but each have their use cases. Even if the visuals are an abstraction on top of text, it will still be its foundation. This foundation of technology and other technologies should be appreciated in order to utilize it the best.

References

Braun, V., & Clarke, V. (2006). Using thematic analysis in psychology. *Qualitative Research in Psychology,* 3(2), 77–101. https://doi.org/10.1191/1478088706qp063oa

Burnett, I., Baker, M. J., Bohus, C., Carlson, P., Yang, S., & Van Zee, P. (1995). Scaling up visual programming languages. *Computer, 28*(3), 45–54. https://doi.org/10.1109/2.366157

Card, S. K., Mackinlay, J. D. & Shneiderman, B. (1999). *Readings in Information Visualization: Using Vision To Think*. Morgan Kaufmann.

Grout, I. (2008). Electronic Systems Design. In *Digital Systems Design with FPGAs and CPLDs* (pp. 43–121). Elsevier. https://doi.org/10.1016/B978-0-7506-8397-5.00002-7

Hirzel M. (2022). *Low-Code Programming Models.* ResearchGate. https://www.researchgate.net/publication/360410556_Low-Code_Programming_Models

Johnson, B., & Shneiderman, B. (n.d.). Tree-maps: a space-filling approach to the visualization of hierarchical information structures. *Proceeding Visualization* '91, 284–291. https://doi.org/10.1109/VISUAL.1991.175815

Myers, B. A. (1986). Visual programming, programming by example, and program visualization: a taxonomy. *ACM SIGCHI Bulletin*, 17(4), 59–66. https://doi.org/10.1145/22339.22349

Nielsen J. (2020). *10 Usability Heuristics for User Interface Design*. Nielsen Norman Group. https://nngroup.com/articles/ten-usability-heuristics

Price, B. A., Baecker, R. M., & Small, I. S. (1993). A Principled Taxonomy of Software Visualization. *Journal of Visual Languages & Computing*, *4*(3), 211–266. https://doi.org/10.1006/jvlc.1993.1015

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Millner, A., Rosenbaum, E., Silver, J., Silverman, B., & Kafai, Y. (2009). Scratch: programming for all. *Communications of the ACM*, 52(11), 60–67. https://doi.org/10.1145/1592761.1592779

Roman, G.-C., & Cox, K. C. (1993). A taxonomy of program visualization systems. *Computer, 26*(12), 11–24. https://doi.org/10.1109/2.247643

Tsai, C.-Y. (2019). Improving students' understanding of basic programming concepts through visual programming language: The role of self-efficacy. *Computers in Human Behavior, 95*, 224–232. https://doi.org/10.1016/j.chb.2018.11.038

Ware, C. (2021). Images, Narrative, and Gestures for Explanation. In *Information Visualization* (pp. 331–358). Elsevier. https://doi.org/10.1016/B978-0-12-812875-6.00009-8