Creating a Visual Programming Language

Louis Eppler
Betreuer: Jens Maue
Maturitätsarbeit MNG Rämibühl
6 Januar 2020

Abstract

For my Matura project I have created a visual programming language. The program contains several elements that can be placed onto a field. These elements represent 'something' the programming language can do. For example simple arithmetic operations, defining and calling functions and comparing numbers. These elements can be connected through lines that represent the variables. The editor is written with Processing (Java) and the code gets compiled into Assembly (Netwide Assembler).
My goal with this project was to create and translate a visual programming language into a low-level programming language. I also wanted to write some simple programs for example printing the Fibonacci numbers, prime factorization and a task from the Swiss Olympiad in Informatics (SOI). Despite some limitations such as missing arrays and being restricted to integers, the programming language is a fun and alternative way to solve coding problems.

Source Code

The source code for this project is available at:
https://github.com/louiseppler/visual-language

Contents

1 Introduction

In this paper I will explain how I created a visual programing language, how to use it and what my motivation was behind it.

1.1 What is a programming language

A programming language is a formal set of instructions that can be understood by a compiler. A compiler is another program which translates the program to something a computer can understand and run. A programming language is used to compute different problems. There are several levels on how a computer program can be written.

The lowest level is Boolean logic. This level is not a programming language, but the way a computer is built up. Boolean logic works with binary values, variables that can be either on or off. To work with these variables a computer uses logic gates. A logic gate is an operator which gives a certain output for a given input. For example an 'AND-gate' gives an output of 'on' only if all inputs are 'on'. The 'OR-gate' gives an output 'on' if one of the inputs is 'on'.

The next levels are the low-level programing languages which are machine code and assembly. Machine code is the set of instructions that can be directly read by a processor. These programs are written in binary and thus hard to understand. Assembly is very close to machine code, but each instruction is translated from binary into words understood by humans.

The highest level of programming languages are the high-level programing languages that most people are familiar with. These include Python, Java and C++. [1]

1.2 Goal of this project

The goal of this project was to create a visual programing language, that gets translated into a low-level programing language. In addition, the instruction set should be close to the low-level language one. I aimed to include basic features of a programming language, such as variables, simple arithmetic operations, conditionals and a way to repeat parts of the code (loops). With these features I wanted to write some simple programs and solve a problem from the Swiss Olympiad in Informatics (SOI) [2].

1.3 Motivation

The idea of such a programming language came from a computer game called Minecraft [3]. Minecraft is a game in which there is a world with different landscapes represented in cubic meter blocks (Figure 1). These blocks can be destroyed and placed anywhere in the world. This way one can construct various builds.
The game also contains a block called redstone. This block is basically an electrical wire. It can be turned on with a lever and can power a light for example. (Figure 2). There is another block (a redstone torch) which can invert the signal

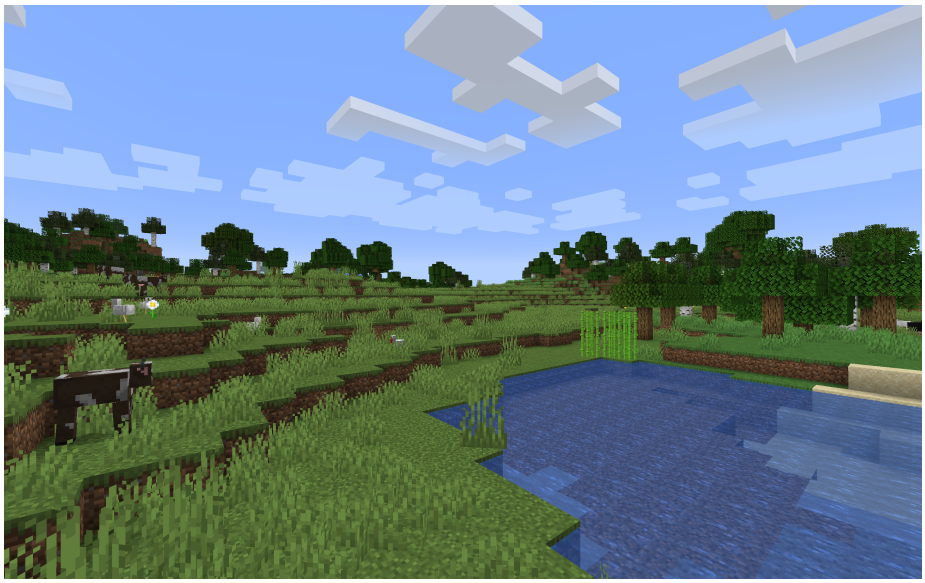

Figure 1: Example of a Minecraft landscape

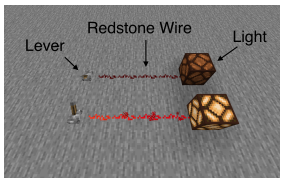

Figure 2: Powering a light using redstone and a lever

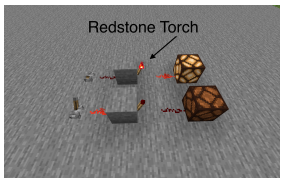

Figure 3: Inverting a signal with a redstone torch

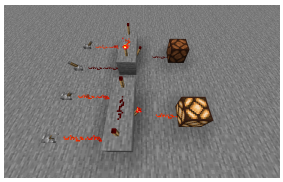

Figure 4: Example of an AND-gate with minecraft redstone

(Figure 3). By combining these blocks in a specific way, one can easily build simple logic gates (Figure 4). By building different logic gates and connecting them it is possible to build complex circuits, for example a binary calculator.

I created a lot of different redstone builds. This made me ask if this principle could be used to write computer programs which could be run in the console.

2 How to code with my language

In this section I will explain how to use my language by introducing step by step the different features and then illustrating them with an example program. I will then demonstrate more complex examples to show how the features work together. At the end of this section there are some useful tips and tricks on how to use the program.

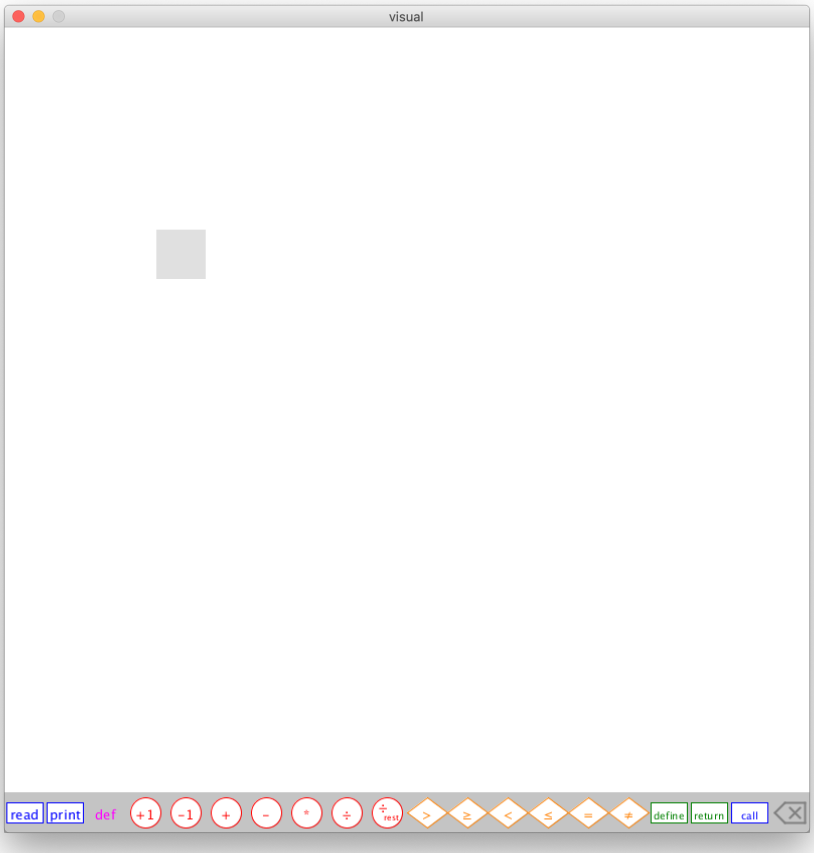

Figure 5: My program when first opened

2.1 A first example

When opening the program you are presented with a big blank field and a menu bar at the bottom (Figure 5). The menu bar contains different elements which can be added to the field. These elements represent different things the programming language can to. These things are simple mathematical operations such as adding two numbers, multiplying them or dividing them, defining and calling functions and comparing different numbers.

The big blank field is where you will code. This field is made up of a grid of rectangles. A single rectangle can be seen when hovering over it.
To start coding the first step is to add some elements on to the grid system. To do this select with your mouse an element from the menu bar. Then click anywhere on the field to place it. To remove an element select the delete tool from the menu (or press `backspace`) and then click the element to remove it.

The elements placed on the field have gray lines coming out of them (Figure

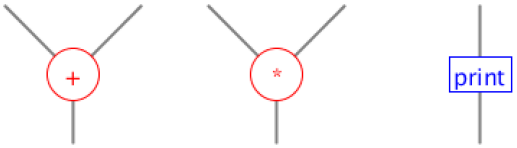

Figure 6: Elements with lines

6). These gray lines represent the integer inputs/outputs of the element. For example the element to add two numbers will have two lines as inputs, the summands, and one as an output, the sum. The input lines are always above the element and the output lines are always below the element.

Print Element The print element is an element which takes one input variable and prints it to the console. The output line is theoretically not needed, but it is there so that if you need that number again, you do not have to reconnect it from the top. When placing this element a window will pop up. This is the text editor window. See how it works in section 2.7.2. If you add any text it will be printed before the number is printed. If you do not want to have any text printed just press `n`.

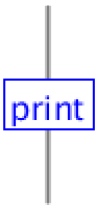

Figure 7: Print Element

Read Element This element allows you to read a number from the console. The number read from the console will be its output variable. Like the `print` element, the `read` element also has the ability to print some text before reading the number.

Figure 8: Read Element

Define Element This element will define any number as a variable. After placing the element on the field it will be set to 0. To change the value press the element and drag the mouse to the neighboring field (Figure 9). Dragging the mouse to the right will increase the value by 1,2... , to the left by -1,-2... . Dragging the mouse to the bottom will multiply the value by 10, 100 ... .

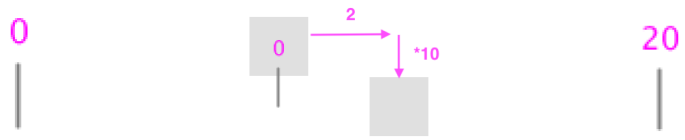

Figure 9: Changing the define element to the value 20

Example: Dividing a number by 20

This example program will read a number from the console, divide it by 20 (integer division) and print the result to the console. To do this, place a `read`, `define`, `divide` and `print` element as shown in Figure 10. Then change the `define` field to 20 as explained before (Figure 9). After placing the elements, it is time to connect them. This is done by pressing one end of a line and dragging it to the end of another line (Figure 11). When this is done (Figure 12) press `enter` to compile. This will create the assembly file. Lastly open the 'run.command' file. This will compile the assembly code and run it in the console which will appear. You can then enter any number and it will print out the the number divided by 20.

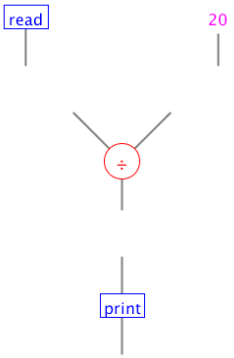

Figure 10: Layout of the elements

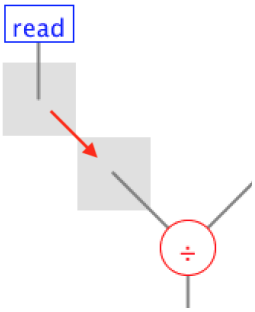

Figure 11: Connecting two lines

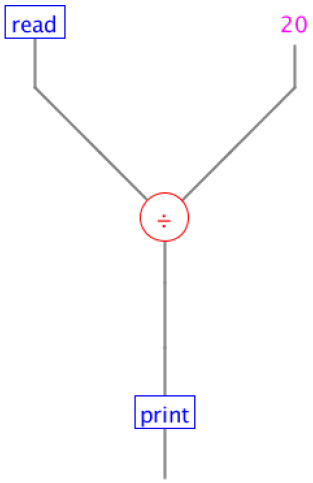

Figure 12: The finished program

2.2 Using functions

If you want to use a part of the code several times without laying the elements out repeatedly, you will have to use a function. A function in my language is a part of the code which takes some variables as an input, computes something and gives back (returns) optionally some variables. This function can then be called several times from different parts of the code.
To define a function place the `define` element from the menu onto the field. The text editor will appear to give a name to the function. To define how many input variables the function needs (the number of parameters), drag the last field of the rectangle to the right/left (Figure 13).

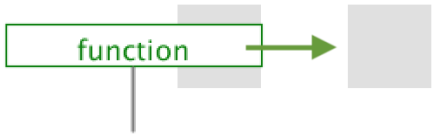

Figure 13: Changing from 1 to 3 parameters.

To call a function place the `call` element onto the grid. Then connect the first field of the rectangle of the `call` element with the first field of the `define` element by dragging the mouse. Then the call element will change its size to the number of variables needed and change the name to 'call function'. (Figure 14)

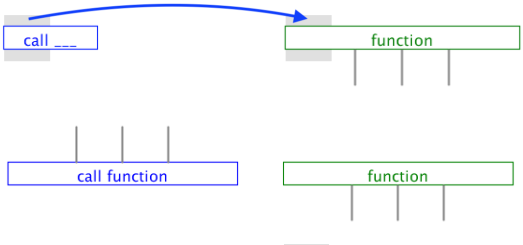

Figure 14: Connecting a call element with a define element

If you wish to return a variable add a `return` element to the grid. Then connect the first field of the rectangle of the `return` element with the first rectangle of the `define` element by dragging. To change the number of variables to return drag the last field to the right/left (Figure 15).

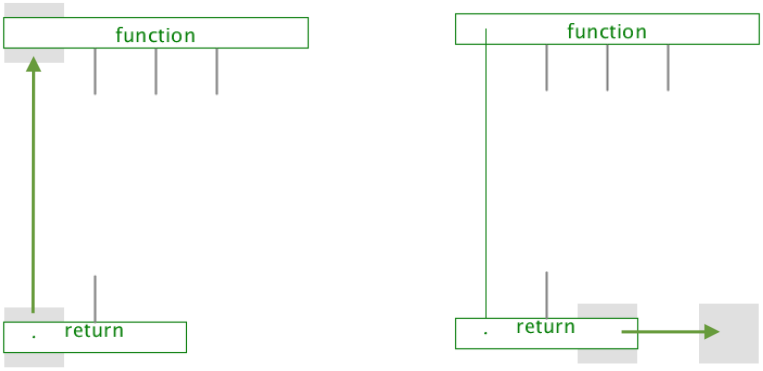

Figure 15: Connecting and adjusting the return element

Example: Squaring a number (using a function)

In this example (Figure 16) a number is read from the console. Then the squaring function gets called. The number read from the console will be passed along. Then inside the function the number will be squared (by multiplying it by itself). After that the number will be returned. Then outside of the function the returned value will be passed to the `print` element.

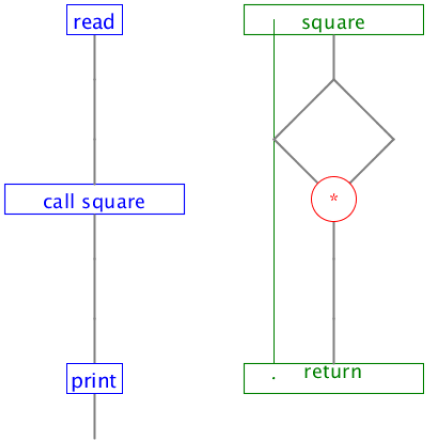

Figure 16: Squaring a number

2.3 Conditionals

In this part we will be looking at how to use conditionals. The conditional elements are the orange rhombuses (Figure 17). These elements can be added to a call element or a return element. If a conditional is added, the element will only be executed if the conditional is true.

Figure 17: The conditionals elements

Example: Biggest number

This is a function which takes two numbers `a` and `b` and returns the bigger one. This is done by defining a function which has two returns. The first return will return number `b` and the second return will return number `a`. By adding a conditional to the first return, the function will only return number `a` if this condition is true. In our case we want the function to return the bigger number. So we will have to compare if number `b` is bigger than number `a`. This is done by connecting the lines as seen in Figure 18.
Note: since the return element with the conditional is higher up in the grid this return will be executed before the second return. Therefore if the number `a` is bigger it will be returned and the rest of the function will not be computed anymore, i.e. the second number will not be returned.

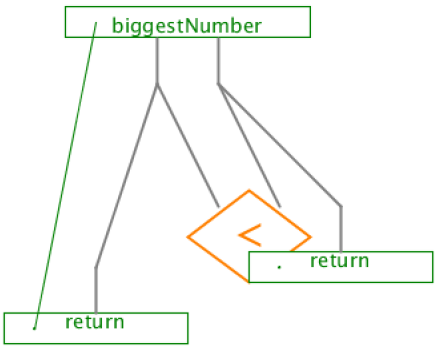

Figure 18: Biggest number

2.4 Using loops

My programming language does not have an element to do loops. However, this can be achieved using recursion. To do so, you define a function that calls itself. This way the function will be called several times. To stop the loop, a condition has to be added to the `call` element.

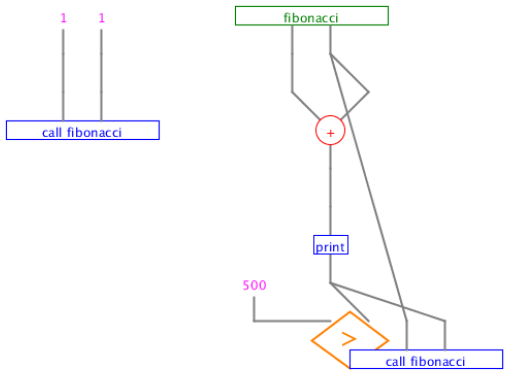

Figure 19: Fibonacci program

Example: Fibonacci

This program (Figure 19) will print all Fibonacci numbers up to 500. The Fibonacci numbers are given by:

$$fib(i) = fib(i-1) + fib(i-2) \quad for\ i > 1$$
$$fib(0) = 1$$
$$fib(1) = 1$$

leading to the sequence 1,1,2,3,5,8,13,21,... To calculate the numbers, a function called `fibonacci` is defined which takes the last two Fibonacci numbers and calculates the next one. At the end of the function the function gets called again leading to a loop. The stopping condition for this loop is the `conditional` element which checks if the number is bigger than 500.

2.5 Example: Prime factorization

In this part I will demonstrate a more complex program to show how to connect the different principles from before. The program (Figure 20) will calculate the prime factorization of a given number. The goal of prime factorization is to write a number as a product of prime numbers. Some examples:

$$10 = 2 \cdot 5 \qquad 96 = 2^5 \cdot 3 \qquad 180 = 2^2 \cdot 3^2 \cdot 5$$

- Firstly, a number is read from the console and passed along to the `main loop` function (*1). This function calls the `divide` function.
- The `divide` function is responsible to calculate how many times a number is divisible by another number. For example if the number 96 is input and the divisor is 2, the function will return the remainder 3 and a count of 5 ($96 = 2^5 \cdot 3$). To do this the divide function calls the `divide loop` function.

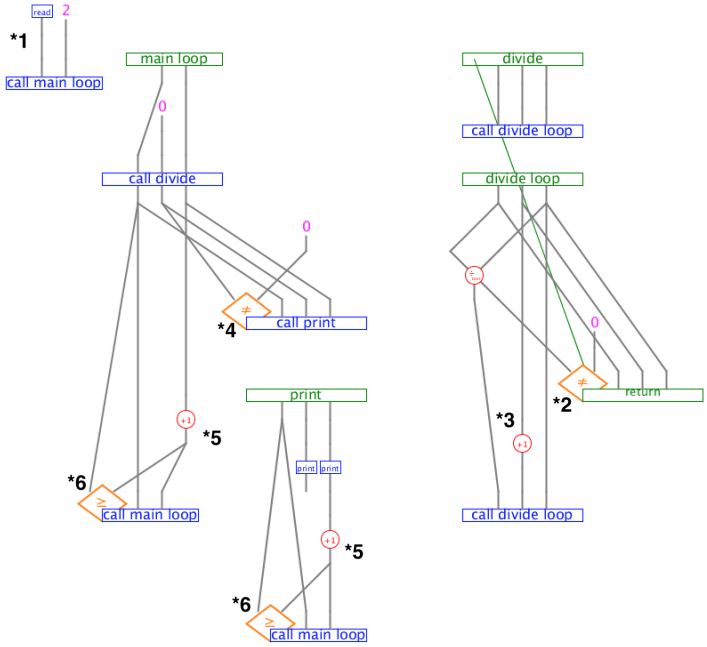

Figure 20: Prime Factorization Code

- The `divide loop` function tries to divide the number and if it succeeds (e.g. the remainder being 0 (*2)), 1 gets added to the counter (*3) and and the process is repeated. If the number is not further divisible then the count and remainder get returned.
- Back at the `main loop` function. The function `divide` has been called and tried to divide the number by 2. If this succeeds (e.g. count not equal 0 (*4)), the print function gets called which prints the result.
- Then the process (of the `main loop` function) is repeated again only this time with a divisor increased by 1 (*5). This is done until the divide function does not succeed anymore (e.g. the number is bigger than the divisor (*6)). After that the program is done.

  Note: this program does not test only the prime numbers but all numbers. It is not a problem, because the divide function never succeeds for non prime numbers because their primes have already been tested.

2.6 Example from the SOI

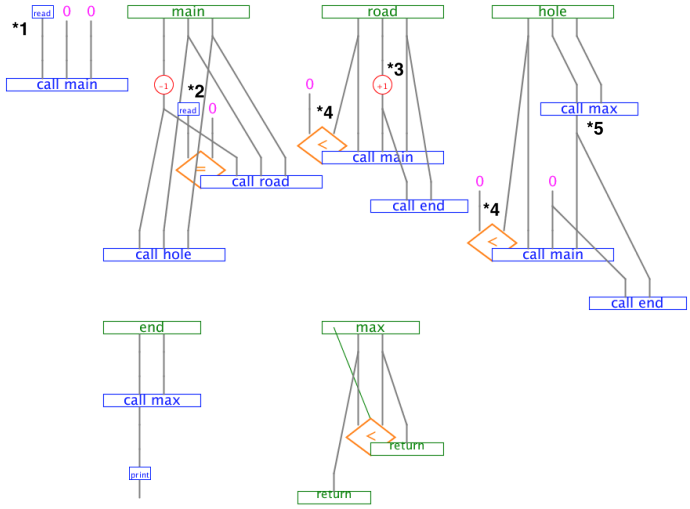

Figure 21: SOI Marathon task code

As mentioned in the introduction, one of my goals was to solve a task from the SOI. The task I solved is called 'Marathon' [4]. The task is to find the longest path in a road, which does not contains any holes. The road is given as a series of 1s and 0s. 1 representing a hole and 0 otherwise.
To solve this program (Figure 21) I used several functions.

- Firstly the `main` function gets called. There the length of the road, which is read from the console (*1), is passed along.
- The `main` function has three variables. The first one represents how many numbers are left to be checked. The second one is the current length and the third the maximum length. In this function the next number in the sequence is read (*2). If this is equal to 0, the function `road` gets called, else the function `hole`.
- In the function `road` one gets added to the current maximal length (*3). If the road is not done yet (e.g. the numbers left is bigger than 0 (*4)), the `main` function gets called again.
- In the function `hole` the maximal length gets updated by getting the bigger value from the current length and the previous maximal length (*5). This is done by the function `max` which is the same function explained in section 2.3.
- If the road is processed the function `end` gets called. This function updates the maximal length again and then prints the value to the console.

2.7 Additional Information

2.7.1 Moving several elements

Several elements can be moved by right clicking a field and then dragging the mouse to another one. This will create a selection box. You can then drag the selection to the destination you want. To delete several elements, select the elements in the same way and press the delete button from the menu bar.

2.7.2 Text editor

Figure 22: Text editor

When any text is required, for example to set the name of a function, the text editor window will appear (Figure 22). Because the software I used for the font-end does not provide an easy way to edit text, the value from the clipboard is used. To change the text, type the text in another text editor and then copy it to the clipboard. Press `r` to reload and `enter` to confirm.

2.7.3 Saving files

My program contains a way to save programs. To save a file press the key `s`. This will open a menu with several options. When selecting 'save local' the file will be saved to the local storage (this is also done automatically when compiling the code). To save several programs you will have to save the program to the data folder. To do this press 'save to data folder' and then enter the name you want the file to be saved as.

2.7.4 Troubleshooting

In case your program does not compile, here are some possible causes.

When placing an element, make sure that the lines connected to the new element do not overlap existing lines. If so, the program will not recognize that these elements are connected.

Double check that all lines are connected (except for print elements). The compiler assumes that the lines are always connected.

Sometimes when placing elements at the edge of the grid, the program will not compile. Some elements check what their neighboring fields are (to check if a conditional is connected for example). If the field does not exist the program will not compile.

When changing the amount of input/return variables the `call` element will

not be updated. To fix this, simply reconnect the `call` element with the define element.

3 The Code

My project is set up into two major components. The first one is the editor for the programming language, the front end. For the front end I used Processing [5]. Processing handles all the graphics and user interactions. The second part is the compiler. For this part I used the Netwide Assembler (NASM) [6]. My program will output a file in NASM which can be run by the computer.

3.1 Processing

Processing is a library for Java wich allows you to produce simple graphics with only a few lines of code. For example:

```
size(500,500);
rect(100,300,100,50);
```

Running this program will open a new window with height and width of 500 pixels. On to this window it will draw a rectangle at xy-coordinates (100, 300) of width 100 pixels and height 50 pixels.
Processing also provides various functions and variables for different user inputs. For example the variable `mouseX` represents the x-coordinate of the mouse pointer in the window.

```
size(500,500);
rect(mouseX,mouseY,100,50);
```

This program will draw a rectangle at the position of the mouse.
Another example is the function `mousePressed`. This function is called when the mouse is pressed.

```
void mousePressed() {
    rect(mouseX,mouseY,100,50);
}
```

This program will only draw a rectangle when the user presses the mouse. Another important function is `draw`. This function gets called 60 times per second. It can be used to do animations.

As shown in these examples, it is very easy to create simple programs with Processing. It is especially easy to do custom graphics and handle user inputs. These are the reasons I used Processing for the front end.

3.2 The Netwide Assembler

When running my program Processing will generate a file which is written with the Netwide Assembler. The Netwide Assembler is an open source assembler for Intel x86 architecture [6].

3.3 How they work together

Figure 23: Overview of compilation

As mentioned Processing handles the graphics and user interactions. Processing creates the part of the program which allows the user to "code". In that part the user can place the different elements and connect them together creating the visual code. When you start compiling Processing generates a file. This part of the compiling is handled by Processing. The generated file is written in NASM. This file can then be compiled again. This time it is compiled by the assembler and will generate an executable file which can be run in the console.

3.4 Class structure of front end

In this subsection I will explain how the front end (written with Processing) is built, by giving an overview of the class structure and what each class does.

The class `Container` is the main class which handles user inputs and some rendering. It instantiates all required classes:

- The `Menu` class renders the menu at the bottom of the program. It contains all the elements which can be added onto the field.
- The `Submenu` class is responsible for drawing the text that appears when hovering over an element. This displays for example the text which will be printed before printing a number when hovering over an element. It also displays the content of the comment field.
- The `Mouse` class is responsible for converting the mouse-coordinates to the corresponding x and y of the field which is pressed. This class also handles the repositioning and resizing of the fields.
- The array `fields` stores the data each field needs. This is done by the `Field` class which gets inherited by other classes for each element type.
- When any text is required, for example to set the name of a function, the function `startEditing` in the `Texteditor` class gets called. (See Section 2.7.2).
- The `FileManager` class is responsible for the menu to save/load programs. The program is converted into a string and saved as a '.txt' file in the 'data' folder.
- The class `Compiler` handles the compilation. This will be explained in more detail in the next section. For the compilation the class `Code` and `Element` get used.

3.5 Compilation to assembly

The compilation of the visual code to the assembly code is the most complex part of the code. This part involved writing different algorithms to get the different elements and their connections translated to the assembly language. It is done in several steps.
To illustrate how it works I will show what happens when compiling the Fibonacci program.

The first step is to assign a number to every line/variable. This number is used to reference the variable from an element. These numbers can be seen in red if `drawDebugNumbers` is set to `true` and are shown in Figure 24.

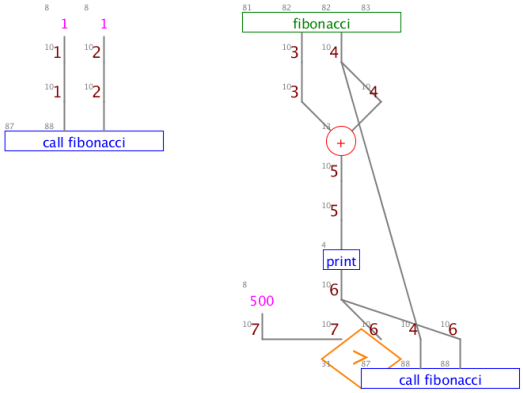

Figure 24: Numbers assigned to lines

The function `getElements` gets the relevant information from the `fields` array and stores it as an object of class `Element` in the `elements` array. The `Element` class includes following information: the ID of the field (these are the gray numbers in Figure 24) and a list of all variables references the element defines and uses (`defines` and `connectors`). The elements array gets printed to the console (for debugging purposes).

```
8   | 0 | [] -> [1]
8   | 0 | [] -> [2]
81  | 0 | [] -> [3, 4]
87  | 0 | [1, 2] -> []
13  | 0 | [4, 3] -> [5]
4   | 0 | [5] -> [6]
8   | 0 | [] -> [7]
87  | 0 | [4, 6, 7] -> []
```

Each line represents one element. The first number is the ID of the field. The second number is the segment number (explained later). Lastly the arrays `defines` and `connectors` are printed. An example for this line:

```
13  | 0 | [4, 3] -> [5]
```

The id is 13 which represents adding two numbers. It takes 4 and 3 as input, the variables to add together and defines the variable 5, the sum of the numbers.

The second step is to assign each element to a segment. A segment in my program is a group of elements that depend on each other. The other elements are not needed to run the specific segment. An example of a segment is a function block.
Assigning the segment number is done with a breadth-first search algorithm (BFS). Each element is a vertex and every line/variable is an edge. The BFS will be called on the first element with the segment number 0. Then every element which the BFS algorithm reaches gets assigned with the segment number. After that the process is repeated with the next element (which has no segments assigned yet).
This is done so that each segment can be called individually. Therefore if you call a function, in the background a segment is called. The segment number is the second number printed to the console.

```
8   | 0 | [] -> [1]
8   | 0 | [] -> [2]
81  | 0 | [] -> [3, 4]
87  | 1 | [1, 2] -> []
13  | 1 | [4, 3] -> [5]
4   | 1 | [5] -> [6]
8   | 1 | [] -> [7]
87  | 1 | [4, 6, 7] -> []
```

The two `defines` (ID 8) and the `call` element (ID 81) are in segment 0. The other elements (all elements required for the function 'fib') are in segment 1.

The third step is to sort the elements. This has to be done because the order of the elements in the array is also the same order of the instructions in NASM. This is done using two criteria. The first criteria is the segment number. This makes it possible to call a segment by itself. The second criteria for the sorting is the following: each variable must be defined before it is used. In the Fibonacci example nothing will change since these criteria are already met.

The fourth step is to merge the connectors. As mentioned before, each variable has a number assigned. This number represents the register used in NASM. Number 0 is for the register `r8`, number 1 for `r9`, number 2 for `rdx`. At the moment each line in the code has its own registers. The CPU is limited with the amount of registers though and it would be wasteful to save a value which is only used once. To solve this problem the code keeps track of when a variable is defined and last used. Each time a variable is defined it will be assigned to the lowest available register. As soon as a register is not used anymore it will be marked as available again. This will change the elements from:

```
8   | 0 | [] -> [1]
8   | 0 | [] -> [2]
81  | 0 | [] -> [3, 4]
87  | 1 | [1, 2] -> []
13  | 1 | [4, 3] -> [5]
4   | 1 | [5] -> [6]
8   | 1 | [] -> [7]
87  | 1 | [4, 6, 7] -> []
```

to

```
8   | 0 | [] -> [0]
8   | 0 | [] -> [1]
87  | 0 | [0, 1] -> []
81  | 1 | [] -> [0, 1]
13  | 1 | [0, 1] -> [0]
4   | 1 | [0] -> [0]
8   | 1 | [] -> [2]
87  | 1 | [1, 0, 2] -> []
```

Now the program needs only a maximum of 3 variables at once.

Now the data is arranged properly to write the assembly code. The fifth step is to call the function `writeCode()` on every element. This function varies between the different elements. The function will add some lines to the final assembly code so that the purpose of the element is fulfilled.
An example: The element for adding two numbers will take the registers 0 and 1 and defines register 0. The code to translate this will look like:

```
"add" + " " + code.registers[n1] + ", " + code.registers[n2];
```

and will be translated into

```
add r8, r9
```

`n1` and `n2` are the registers numbers which must be added. `code.register` contains an array which translates the register number to the register name.
Note: this code will only work if the first register `n1` is equal to the destination register. If this is not true the following lines gets added:

```
"mov" + " " + code.registers[d] + ", " + code.registers[n1];
"add" + " " + code.registers[d] + ", " + code.registers[n2];
```

First the register `n1` gets moved to the register `d` (destination register number). Then the register `n2` gets added to the destination register. If the destination register is `rdx` it will be translated into

```
mov rdx, r8
add rdx, r9
```

The last step is to add some general code which includes some basic functions such as converting a binary number to chars to be able to print. Then the file gets saved to the 'main.s' file. To see the whole code, refer to the appendix.

4 Reflection

4.1 How NASM influenced my language

One of my goals for this project was to compile my language to a low-level programming language and stay close to the instructions that exist there. Every element that can be used in my language corresponds to an instruction of NASM. All operators are available in NASM. For example the element `+1` gets translated into `inc`, the element to divide a number to `div`. The `call` element corresponds to a `jmp` or `call` instruction. The reason the conditionals are done by connecting them to a call element is because this is the way it is done in assembly. Instead of using `jmp` one can use `jg` (jump if greater) or `jne` (jump if not equal).

4.2 Next steps

Besides the small bugs mentioned in section 2.7.4 there are some additional features that would be nice to have.
Firstly a version to run on Windows and Linux. As long as these computers use an intel x86-processor, not many changes would have to be done because NASM is supported for all platforms. For the front end nothing would have to change, since Processing runs on Windows and Linux.
The lack of arrays is something that limits the functionality significantly. To add arrays the ram must be included and this makes the assembly code more complex.
Another feature that would be nice to add is additional data types. Only being able to use ints limits the language.

4.3 Comparison to similar languages

When starting out with this project I did not know of any similar concepts to mine. I did not do any active research because I wanted to come up with my own approach. Nevertheless I have heard of some similar programs.

Scratch

When telling people about a visual programming language most people thought of Scratch [7]. "With Scratch, you can program your own interactive stories, games, and animations". This is done by combining several code blocks from higher-level programming languages.
There is a big difference between Scratch and my language. Scratch is based on a high-level programming language. Coding with Scratch is like coding with a text based language, only each line is represented as a block.

Labview

Labview is a programming language used by engineers to connect hardware, control them and analyze the data [8]. Labview uses a very similar approach on how to do basic operators. There is an icon that takes some lines as an input and gives some lines as an output. The main difference is how functions and conditions are used. All my elements are based on a low-level programing language and correspond to instructions from assembly.

4.4 Conclusion

With this project I was able to achieve the goal I set myself. I was able to create simple programs and compile them into a lower-level programming language. I also solved a problem from the SOI but there I started to reach the limits of my language. I was only able to solve the exercises that do not need arrays.
Moreover, this project contained all the elements of what I consider programming to be: learning a new language (NASM), designing user interfaces and writing algorithms. Going through all these steps was very rewarding. In my opinion the result turned out very well. I really enjoyed using my visual language to write programs because it is a different and fun way of solving coding problems.

Source Code

The source code for this project is available at:
https://github.com/louiseppler/visual-language

References

[1] Noam Nisan and Shimon Schocken (2008): The Elements of Computing Systems. The MIT Press

[2] Swiss Olympiad in Informatics, https://soi.ch, accessed 19.12.19

[3] Minecraft, https://www.minecraft.net, accessed 19.12.19

[4] SOI Task Marathon, https://soi.ch/contests/2020/round1/marathon/, accessed 19.12.19

[5] Processing, https://processing.org, accessed 19.12.19

[6] The Netwide Assembler, https://www.nasm.us/doc/nasmdoc1.html, accessed 19.12.19

[7] Scratch, https://scratch.mit.edu/about, accessed 19.12.19

[8] Labview, https://www.ni.com/en-us/shop/labview.html, accessed 19.12.19

5 Appendix

This is the generated NASM code for the Fibonacci program:

```
start:
;define number
mov r8, 1

;define number
mov r9, 1

;call function: fibonacci
jmp segment1

call exit

;function: fibonacci

;add
add r8, r9

;print
push rax
mov rax, r8
call iprintLF
pop rax

;define number
mov rdx, 500

;call function: fibonacci
cmp rdx, r8
jg condition0
jmp condition1
condition0:
push r9
push r8
pop r9
pop r8
jmp segment1
condition1:

call exit
```

Note: there are some other lines of code that are added which include the other functions called in this program.

Der/die Unterzeichnete bestätigt mit Unterschrift, dass die Arbeit selbständig verfasst und in schriftliche Form gebracht worden ist, dass sich die Mitwirkung anderer Personen auf Beratung und Korrekturlesen beschränkt hat und dass alle verwendeten Unterlagen und Gewährspersonen aufgeführt sind.