CodeChains: Visual Message-Based Programming System

undergraduate thesis of

Lefteris Kotsonas

csd4264@csd.uoc.gr, lefteriskotsonas2@gmail.com

supervised by

Prof. Anthony Savidis

as@ics.forth.gr

COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF CRETE

Προσθέστε κεφαλίδες (Μορφή > Στιλ παραγράφου) και θα εμφανιστούν στον πίνακα περιεχομένων.

Abstract

Code-Chains is a visual-programming editor for a subset of the programming language JavaScript. The goal of this project is to increase novice and expert programmers' happiness and productivity by allowing programmers to create and understand programs in new and different ways. At its' core, CodeChains is a node and link based representation of code. When given that representation, the Code-Chains editor produces JavaScript code.

1. Introduction

This thesis presents and validates parts of the design of Code-Chains, a general purpose Visual Message-Based Programming Language (VMBPL), intended to reduce the gap between programmers and non-programmers in creating serious, reliable and scalable programs.

1.1 Visual Programming

One of the existing system tools that enables everyone to compute is visual programming languages and visual programming environments. Visual programming is a programming paradigm that uses visual elements, such as graphical icons, blocks, or lines, to represent code, data, or control flow. The idea of importing these elements is what differentiates Visual Programming Languages(VPLs) with traditional programming languages and defining VPLs as multi-dimensional, while text-based as one-dimensional[1]. These visual elements can be manipulated, connected, and arranged in a way that represents the logic of a program, making it easier for new programmers to understand and create software.

So far VPLs have been limited to specific domains such as music(Max), video games (Blueprints in Unreal Engine) and test equipment (LabVIEW). VPLs may be further classified, according to the type and extent of visual expression used, into icon-based languages, form-based languages, and diagram languages. In recent years the kinds of VPLs that prevailed are the block-based and flow-based, but that doesn't mean that VPLs settled down to these forms. Visual programming continues to evolve as a way to make programming more accessible, intuitive, and user-friendly for a wide range of users, and each program visualization is only a point in the multi-dimensional space of visual representations[1].

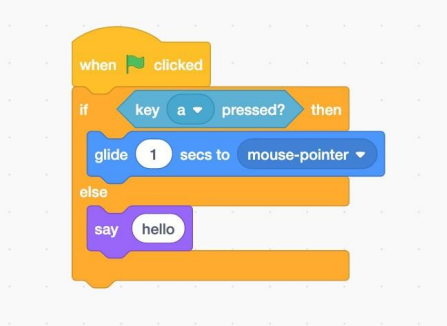


Figure 1: Block-based visual programming in Scratch

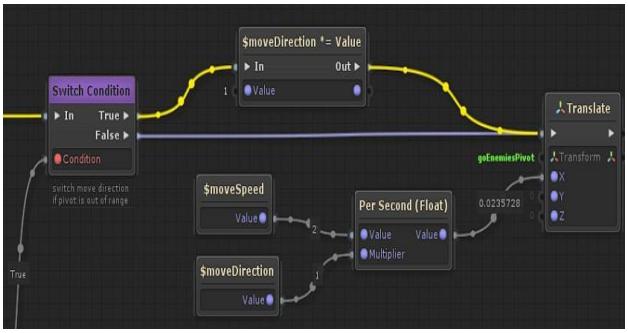


Figure 2: Flow-based visual programming in Unity

1.2 Message-Based Programming

With the terms Message-Based Programming (MBP) or Message-Oriented Programming we refer to the concept of passing data between processes. These data are called messages, and they can be communicated selectively from one process to another, or to a group of processes, or broadcast within a group as well as to all processes on a multiprocessor[4]. Furthermore, messages have a specific type and can contain as many arguments as required. Consequently, each process can be a caller (sending message), or a responder (receiving message), where their interaction is the execution of a program.

In MBP processes and expressions can be strongly related. The join-calculus[2] possesses two base syntactic classes: processes and expressions. Processes form the skeleton of the join-calculus as a concurrent language, whereas expressions are more oriented toward functional programming. Both processes and expressions have a similar concrete syntax. Generally, they are discriminated by context: expressions are adequate in a context where a result is needed (such as arguments to message sending), whereas processes occur where some actions are to be performed.

Our focus on this thesis is the design of tool that supports MBP. Visually, MBP and block-based programs aren't that different. The main difference is that MBP have a sparse representation of code, that helps us distinguish the different concepts inside the structure of code. JavaScript[1] is well-suited for message-based visual representation due to its multi-paradigm and object-oriented nature.

CodeChains' design is based on visual constructs of JavaScript terms verbatim, making the language almost entirely self-explanatory.

---

[1]A JavaScript program consists of a sequence of statements. Each statement is an instruction to do something, like create a variable, run an if/else condition, or start a loop. Expressions produce a value, and these values are slotted into statements.

[2]The join-calculus is a process calculus developed at INRIA. The join-calculus was developed to provide a formal basis for the design of distributed programming languages, and therefore intentionally avoids communications constructs found in other process calculi, such as rendezvous communications, which are difficult to implement in a distributed setting[5].

1.3 Motivation

Although there are many VPLs serving our purpose, most of them are targeting students of lower secondary school. The essential target audience of Code-Chains are the students of upper secondary school. CodeChains will allow these students to participate in constructing programs in a highly-interactive environment with features resembling the ones in modern code editors. Our expectations from this tool is to teach said students the basics of programming and make them more familiar with textual-code editors.

Except educational purposes, another goal of CodeChains is to have a general use. So, like textual programming languages, we want to enable anyone to do anything with the computer. To make that possible, in CodeChains project we want to make VPLs scale to large programs and large systems. The most common complaint from people who have actually tried visual programming is that it works well in the small, but becomes untenable past a certain complexity threshold [2]. In other words, we want to see whether we can go past that barrier to make a general purpose VPL.

1.4 Overall Goals

The overall goal of our research project is to bring a programming environment to non-programmers with the following properties. It should (1) have a very low threshold to start programming, (2) allow a seamless scale from very simple to highly advanced programming, depending on the level of the user, (3) allow users to assist in creating highly reliable programs, (4) allow users to assist in the production of highly optimised, high performance, programs (5) have the benefits of modern programming languages: such as advanced features (6) Clean (doesn't look messy) and (7) Clear (meaning of code is unambiguous)

2. Code-Chains Description

Since every language, including visual languages, is used for different purposes, it is essential to explore new options to writing programs. Code-Chains is a new and in-development visual-language editor, with the hope of eliminating the previous problems. Code-chains has linear structure, meaning that users can construct code that has a one-way order of executable commands. That way users can read and analyze the code easily. Moreover, CodeChains generates JavaScript code. That feature allows programmers to match visual to text code and vaguely learn text code and its debugging. Followed up is a more in-depth description of this editor.

2.1 GoJS

GoJS is a JavaScript library for building interactive diagrams, such as flowcharts, org charts, state diagrams, and other types of diagrams. It provides a visual programming environment for creating and editing diagrams, allowing users to create and manipulate visual elements such as nodes and links using a drag-and-drop interface.

GoJS includes a variety of features for creating and editing diagrams, such as automatic layout, data binding, and built-in support for working with common diagram types. It also provides a variety of customization options, such as styling and templating, to allow users to create diagrams that match their specific needs.

Some of its features include :

- support for creating many different kinds of diagrams
- a rich and extensible object model
- an intuitive and easy to use API
- support for custom shapes and templates
- built-in editing and event handling
- automatic layout and routing
- support for grouping, zooming, and panning
- built-in serialization and undo/redo
- support for touch and gesture input
- various types of data binding for binding to your own data
- performance optimization for diagrams with many objects
- support for Internet Explorer, Edge, Chrome, Firefox, Safari, and iOS and Android devices.

GoJS is typically used in applications that require the creation and manipulation of diagrams, such as business process management, workflow, and data visualization applications. It is a commercial product that can be licensed and integrated into an application, however they do offer free and demo version to evaluate the product as well.

For the implementation of CodeChains, GoJS has everything we need. Since CodeChains is in the context of creating nodes and links, GoJS fulfills most needs for the editor. GoJS supports graphical templates and data-binding of graphical object properties to model data. It is

only needed to save and restore the model, consisting of simple JavaScript objects holding whatever properties the app needs. Many predefined tools and commands implement the standard behaviors that most diagrams need. Customization of appearance and behavior is mostly a matter of setting properties.

Now, with that tool, we can graphically implement our editor and use in our advantage its features for our purposes. The only thing left is to implement internally a way to give these graphical objects a functionality. With the use of GoJs, it is up to the program to modify each node and link to give it content.

2.2 Code-Chains Overview and Architecture

GoJS on its own doesnt give life to its components, for that reason we need to form the objects as we like. To handle these objects, we need to translate the model generated from GoJS to a model more acceptable, generally used and flexible. AST model is used in most kind of applications like this one to store data. Furthermore, now that we have a more concretely data representation, we need to utilize it for the generation process.

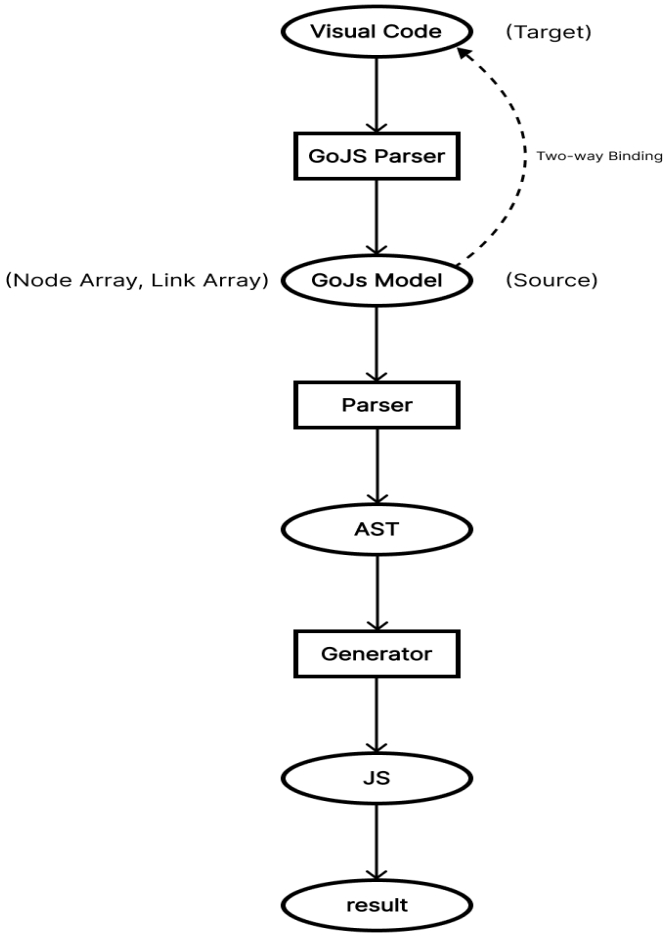


Figure 3: Architecture of code using GoJS

Visual Code is the graphical representation of code, the one we see. GoJS parser is a function inside GoJS library that identifies each component represented and translates it to a simplified element in an array, consisting of its properties. Editing the properties on the component has an impact to the components data and similarly to the visual representation.

After the generation process we get the main product, a JS code that can be executed.

2.3 Message-Based Syntax

The core idea is that textual structures and graphical representations are but points along a smooth continuum. CodeChains is a node and link based representation so graphically these two are variables to move along the axis..

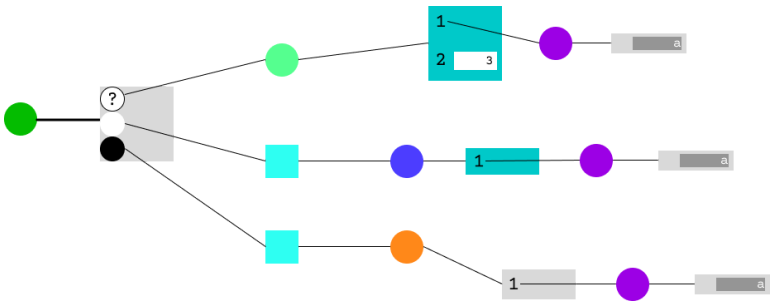


Fig 4.1: Phase 1 of Graphical Representation

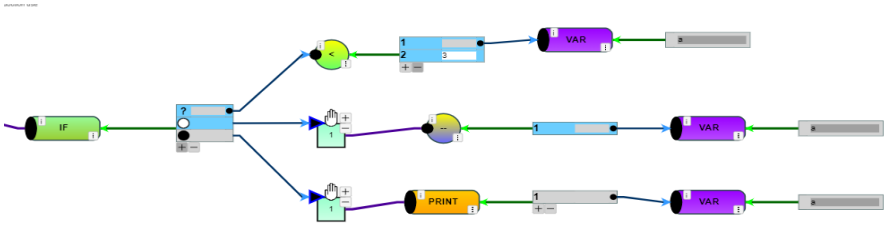


Fig 4.2: Phase 2 of Graphical Representation and the representation of CodeChains

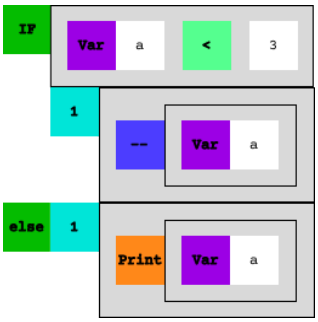


Fig 4.3: Phase 3 of Graphical Representation

```
Var a = 5;
if( a > 3 ) {
    a--;
}
else {
    console.log(a)
}
```

Fig 4.4: Phase 4 and final representation (Textual)

To illustrate that we start with a very graphical representation where each node is separate and then we can slowly collapse nodes into each other. We can go from a sort of a more graphical freeform layout to one that uses indentation more clearly. We start cropping unecessary information to make things compact, remove links and get a more structured feel for it. Finally from the structured representation we make it more textual. That way we've gone all the way from graphical code to textual code as a series of continuous changes.

2.3.1 Nodes

Here are some of the nodes representing the syntax.

Fig 5.1: Function Node

Fig 5.2: If Statement Node

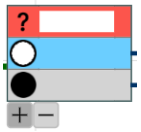


Fig 5.3: If Arguments Node

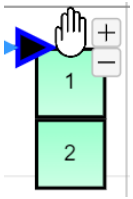


Fig 5.4: If Blocks Node

Each category of node has a different meaning in the code itself. We can easily distinguish these categories by the shape of the node. Furthermore, each category consists of nodes of different use. We also distinguish these nodes based on color.

2.3.2 Links

Link syntax by default is just a line that connects two nodes. Sometimes links clutter things up so we can make an alternative link replacing appropriately two links related. Links are identified based on the adjacent nodes. Their purpose is to index the position of the right-hand node into the others sub-elements. Links are also distinguished based on color.

For CodeChains syntax, first we define the main block, the block of where the executable code will be. The main block shows the order of commands to be executed, as in the text-based systems. Now, for the main part, we are going to look into the command syntax. For each command the syntax is alike, for example, we can take the print statement, which consist of the name of the statement (“print”) and its arguments. We separate these two concepts with two different nodes, and we link them with a category of link. Each category of link is used for different reasons.

Now that we know the syntax of the statement, we can link it to the main block so that command can be executed. For the command to be complete, we need to pass values to the arguments. The content of the argument can be a value, a number or string, or another command. The expression that command holds will be formulated to the argument. In other words, the contents of the argument can be a selected combination of commands and values, resulting in a program of data calculations.

Fig 6: Definition and Declaration of an JS object

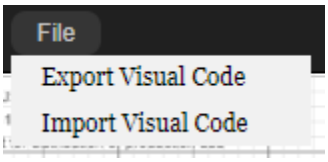
We notice that Node and link representation results to an AST-like program. The meanings of these concepts are strongly related, which is something very useful for learning JS grammar.

3.Features

In the UI we grasp basic features that exist in most code-editors. Since CodeChains targets young learners, the design approached has to be fancy, friendly and trending. The design actuates the programmers to use the features to their full potential, giving weight to the UX of the system and also making programmers more productive.

3.1 Save/Load File, Copy/Paste, Undo/Redo

A user, in whatever application he is using, is familiar with these tools. Copy/Paste, Undo/Redo are basic shortcuts for formatting a text, giving the user a more fast experience. Saving and Loading our work is a guaranteed feature in every application. CodeChains is no different. Specifically, in CodeChains we are not modifying the graphical objects when we use these tools, we are just modifying the model that it translates to. The diagram is a representation of the model.

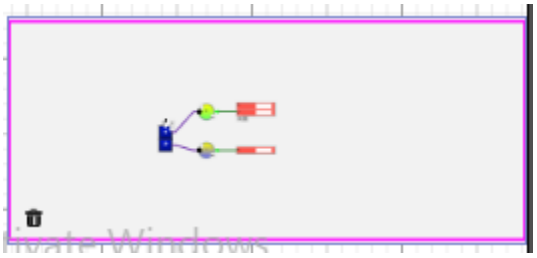


3.2 Palette

Since in VPLs we are handling a sum of components, the ways to place these components to our frame of code are limited. In CodeChains we used the classic drag-n-drop feature. The concept of it is easy in the eyes of users, since they know what they pick and where they place the component. As with flow-based VPLs, the location of each component is vague making the process of making a program vague too. A Palette that categorizes Statements is fitting for this kind of editor.

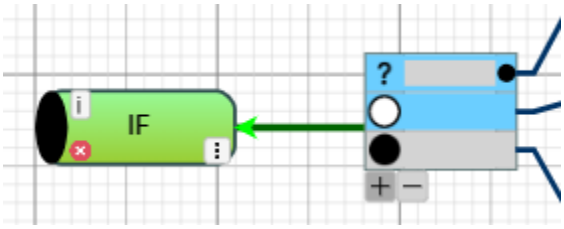
3.3 Program Overview

In this vague process of creating a program, users visualize concepts and implement them in a diagram. Users may feel comfortable in a sparse way of placing elements, where many elements can be collected in a small area in the diagram when the elements are relative. As with text-based VPLs, large scale programs are difficult to be traversed when you dont know the exact location of functions. But it is natural for us to have a general visualization of our code, meaning that some characteristics of our code are noticeable. For this reason, overview of our code can be quite helpful in retracing the flow of code. Code-Chains enhances that feature as each element represents different color so that can be a way to identify easier these clusters of code.



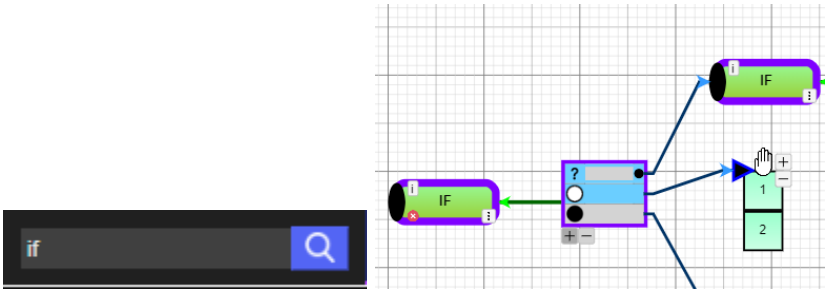
3.4 Syntax Warnings

Even though VPLs are meant to make programming easier, automating the process of creating a program doesnt offer the programmer choices. In order for the user to evolve in his programming skills he must learn how to code with best performance. Syntax in programming is important to avoid possible bugs in the code. CodeChains, since it must teach the user how the real programming can be, has the feature of showing warnings in the visual code. In each link, we are doing an action that must be examined. Such warning can be when we are place statements inside other statements. That kind of mistake has compile errors, but being able to see that mistake on-the-fly makes coding faster.



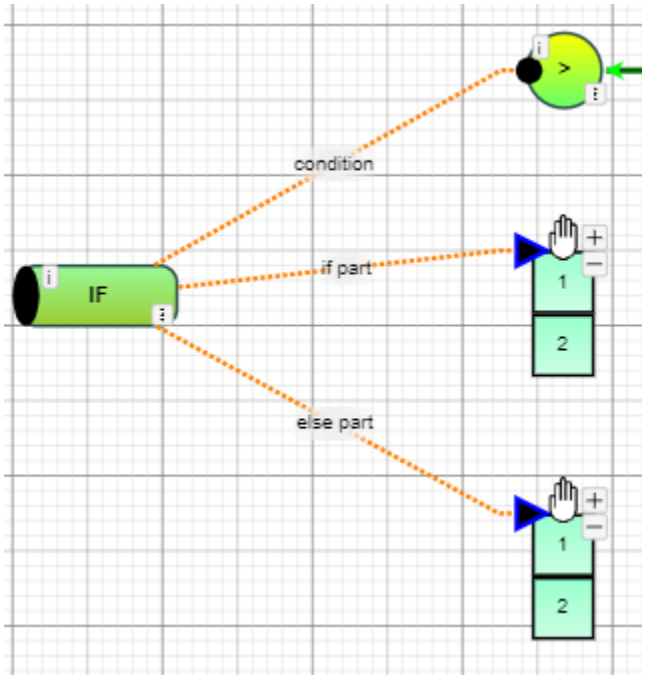
3.5 Search Bar

Another useful feature in CodeChains is to locate keywords. As with text, searching for specific words is time consuming. That problem becomes more intense the more the content there is. For that reason we should be able to search and focus on that blocks of interest.



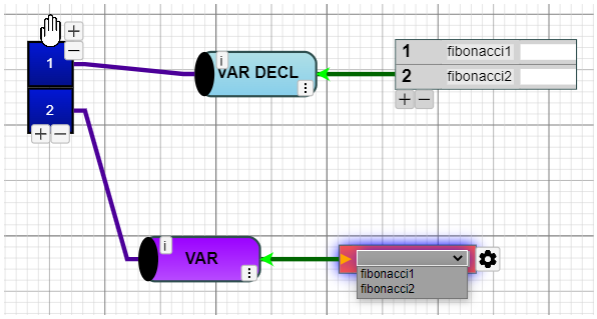
3.6 Collapse/Expand Arguments

CodeChains tries to squeeze as many features as possible that originate from text-based editors, one of which is to shrink and expand code. Generally, we are more interested in the statements. From statements we can see the flow of the code and as such, some statements we want to be visible at all times. CodeChains follows that rule too. From the context menu of a particular function node we can collapse its' arguments. That clears for us a little space and have our code a bit more compressed.



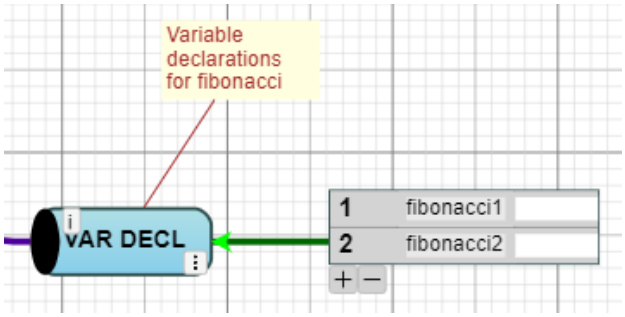
3.7 Variables Dropdown

One of the most break-through feature in CodeChains is “Variables Dropdown”. “Variables Dropdown” isn't for granted in most VPLs, but in CodeChains we do the step further. Since CodeChains generates code in JS, it is important to explain the concept of objects. Objects can be very complex but have many ways to be expressed, making them a main component in JS. To correspond to that term, and be able to reference it, we have nodes to reference this type. So, instead of the “Variables Dropdown”directed to the global single-level variables, with the node “GET ELEM” we can refer to variables inside objects. To refer to it we use a series of variables, starting from the most external object. That multi-level term in our code make “Variables Dropdown” have many phases, something that makes our editor special in referencing of variables



3.8 Comments

Comments are no different from the rest of the code, and they must be treated as such. In CodeChains we have the ability to create comments and place them wherever we like. Of course, we can hide and show them if we want, since they can be in the way of the code.

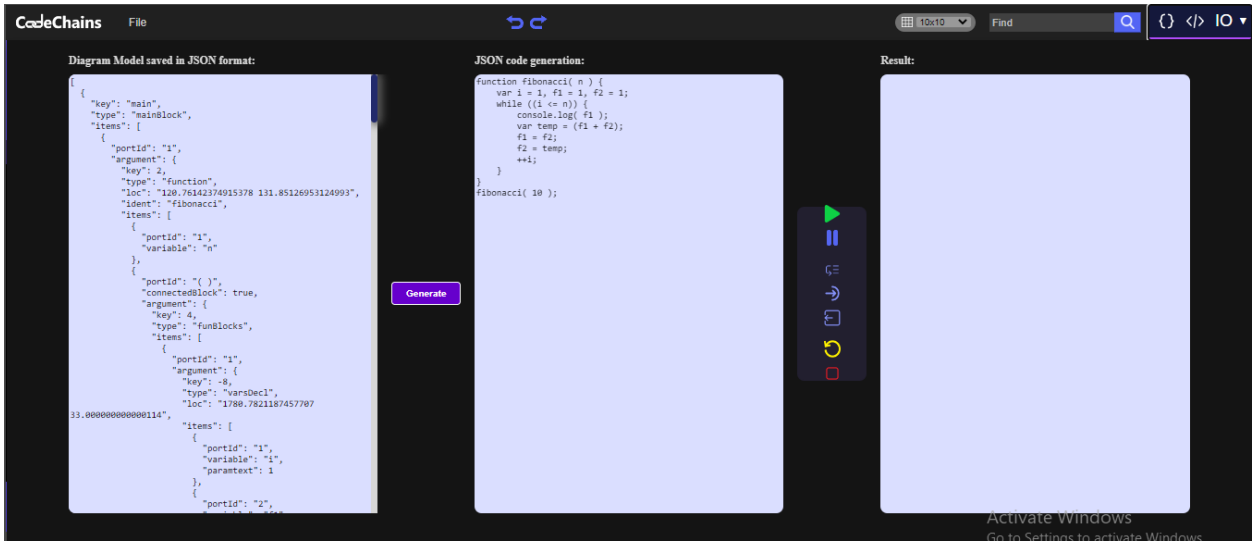


3.9 Breakpoints

Breakpoints are essential to the debugger too. Since we have a linear flow of code, breakpoint make sense to exist in this editor. In the debugger we stop before the execution of a statement with a breakpoint. When that happens, we can step into that statement, researching that chained statements implied, or step over to go to the next statement in main block.

3.10 ToolBox

Crucial part of a VPL is to produce the output of the code. Such a feature is still in development in CodeChains. The user at all times should be able to see the output of the code, otherwise his code wouldn't have meaning. Breakpoints have a important part in this, debugger ascertains the result of each statement, and breakpoints are used to specify until which one. Not only that, in that debugger we move between the statements as we wish, so that we can identify possible errors in the textual code generated.



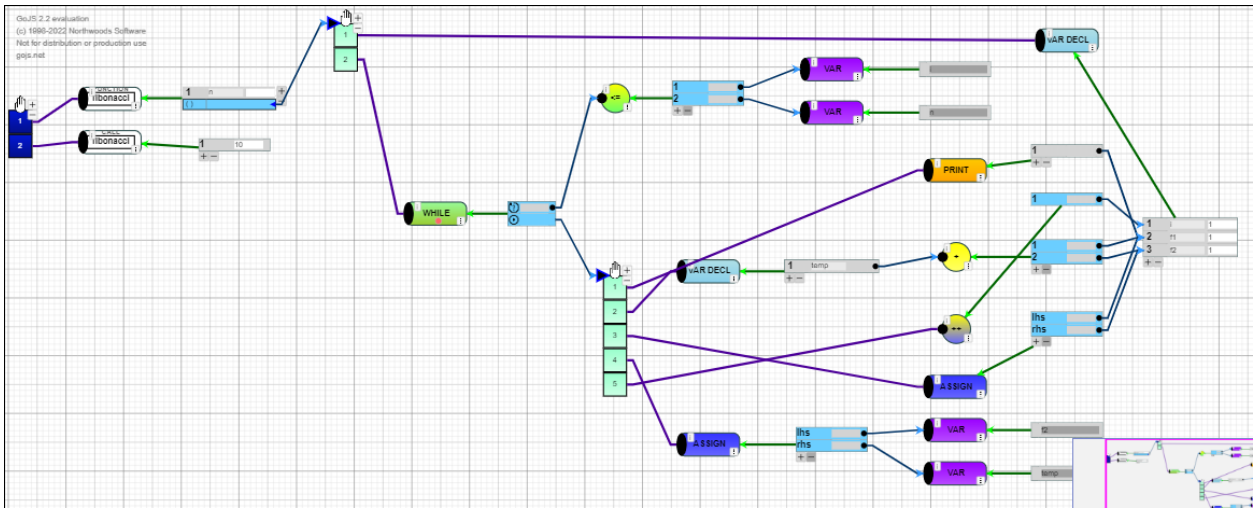
4. Future work

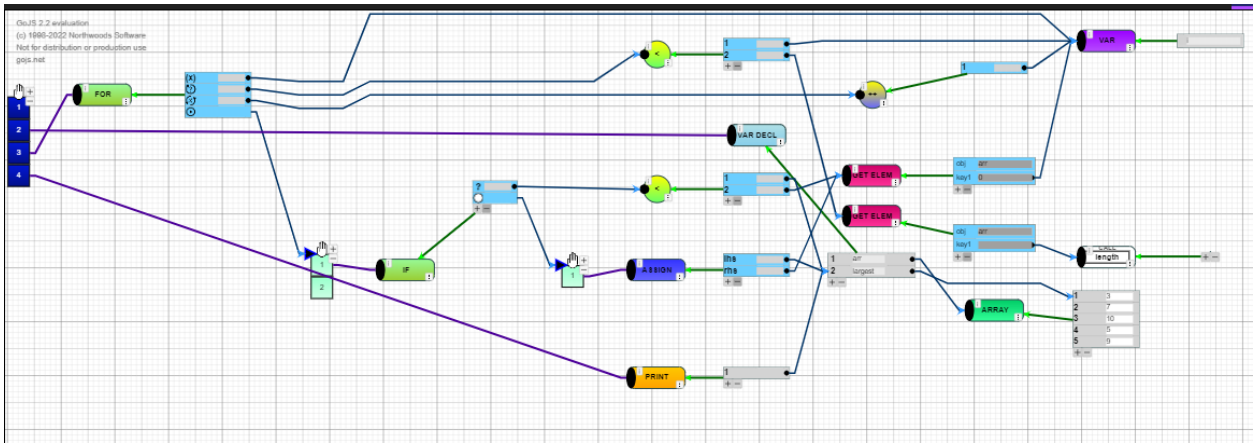
Future work will (or may) include the following. First, the implementation of the message-based debugger which is an essential part for the system to be complete. The programmers then will have access to more features like making good use of the breakpoints and executing code step-by-step. Second, but parallel to the first, the implementation and design of statements and features that allow the user to construct more complicated programs and with more ease.

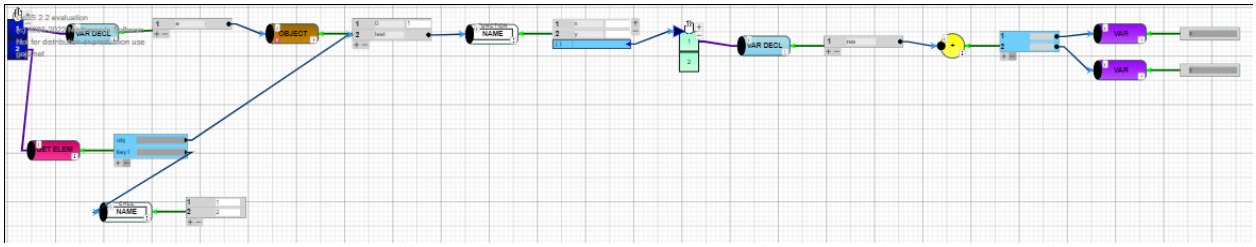
5. Conclusion

Visual Programming can be very close to the traditional text-based programming in case of the features. Each visual editor can be used in different scenarios, and this particularly may be a bridge to connect the novice programmers to experts.

Appendix


fibonacci
fibonacci
WHILE
VAR DECL
PRINT
ASSIGN
VAR
VAR


FOR
IF
VAR DECL
PRINT
ASSIGN
VAR
length
ARRAY


VAR DECL
OBJECT
NAME
VAR DECL
VAR
VAR
NAME

Bibliography

[1] Myers, Brad A. "Taxonomies of visual programming and program visualization." Journal of Visual Languages & Computing 1.1 (1989)
[2] Visual Programming Languages: A Perspective and a Dimensional Analysis
[3]I Know What You Did Last Summer An Investigation of How Developers Spend Their Time
[4] Visual Programming for Message-Passing Systems, Nenad Stankovic Kang Zhang
[5]Cedric Fournet, Georges Gonthier (1995). "The reflexive CHAM and the join-calculus"