COMPARISON OF FLOW-BASED VERSUS BLOCK-BASED PROGRAMMING FOR NAIVE PROGRAMMERS

by

Kruti Dave

Bachelor of Engineering in Information Technology, Mumbai University, 2012

A thesis

presented to Ryerson University

in partial fulfillment of the

requirements for the degree of

Master of Science

in the program of

Computer Science

Toronto, Ontario, Canada, 2018

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I authorize Ryerson University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Ryerson University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

I understand that my thesis may be made electronically available to the public for the purpose of scholarly research only.

Abstract

Comparison of Flow-based versus Block-based Programming for Naive Programmers

Kruti Dave

Master of Science, Computer Science

Ryerson University, 2018

There is general agreement that most people should have some programming ability, whether to investigate the vast amount of data around them or for professional purposes. Visual Programming Languages comprise two broad categories: Flow-based, functional programming or Block-based, imperative programming. However, there has been a lack of empirical studies in the visual programming domain to evaluate the relative benefits of the two categories. This research provides an empirical study to analyze the effects of the comparison between Flow-based and Block-based paradigm, to determine which of the two representations is easier for non-programmers or novice programmers. Each user is given a random, simple problem to program in a random environment. Both of the environments, Flow-based and Block-based are designed to be as similar as possible to make the comparison useful. The results indicate that Flow and Block are equivalent environments for non-programmers or novice programmers in terms of usability and effectiveness.

Acknowledgements

Firstly I would like to thank my supervisor Dr. Dave Mason, for granting me the opportunity to work in the area of my interest and for his continuous encouragement, help and guidance in completing my research. My biggest thanks goes to my family; my father Harish Dave, my mother Daksha Dave and my brother Nikunj Dave for being a constant pillar of support whenever I needed them, for showering me with ceaseless love and for their constant guidance and help. My family has always been my source of encouragement and hope and this accomplishment would not have been possible without them.

Contents

List of Figures

Chapter 1

Introduction

My thesis is that Flow-based visual programming environment and Block-based visual programming environment are statistically equivalent for naïve programmers in terms of usability and effectiveness.

1.1 Motivation

> “The tools we use have a profound (and devious!) influence on our thinking habits, and, therefore, on our thinking abilities” - Edsger Dijkstra, Turing Award Winner (1972).

In the context of this thesis, ‘the tools we use’ refer to the programming development environment, which have an enormous, and often unforeseen, impact on how and what one can think. Hence using the right environment is necessary while coding, be it for learning purposes or manipulating data. That is one of the motivations behind conducting this research to build the right environment for users, who are non-programmers by profession or have less experience in programming, to help them do their required data manipulation tasks.

There is a huge volume of data surrounding us, which if utilized and in the right manner, could prove to be beneficial to people. But one cannot generally get a sense of the relationships between things just by looking at the raw data. Either because there are too many connections between different data sources to analyze, or there is simply too much data available. Hence, it becomes important to have a programming environment that can help users to analyze data easily, so as to avoid a vast amount of data going to waste.

Numerous people have the analytical mind to utilize the bulk of data available, but do not have the required skillset to do so. Programming in a general Text-based environment can be exceptionally demanding and taxing, especially for users, who are not interested in learning to program but only have the intent to manipulate their data. Visual programming languages could be useful for such a task and are discussed in more detail in section 2.1.

1.2 Approach

The shortage of empirical studies in the visual programming languages domain has been very apparent historically, according to K. Whitley (1997). The present study provides an empirical study to analyze the effects of the comparison between two visual programming environments (Flow-based and Block-based), to establish which representation is more usable for non-programmers or novice programmers, who are not professionally trained. Although several studies have been conducted to determine the efficiency of a visual programming language in comparison to a Text-based programming language, this present study is the first one that will address the comparison between Flow-based and Block-based visual programming paradigms.

The main objectives of this research are to:

1. Evaluate which of the two programming paradigms (Flow or Block) is more usable for non-programmers or novice programmers.

2. Evaluate both of the environments' ability to help users in solving data manipulation tasks and highlight its usability and limitations.

1.3 Outline

Chapter 2 gives an overview of the background and classification of the visual programming languages, followed by examples of a few other existing visual programming languages in both domains (Flow-based and Block-based). Next a brief overview of usability along with different ways to measure and evaluate the usability of a programming environment is discussed. The last part of the chapter discuses comparisons of Flow-based with Text-based languages, and Block-based with Text-based languages.

Chapter 3 first presents a brief description of the experiment. This is followed by a detailed description of the two environments we constructed - Flow-based and Block-based. Next, various parts of the experiment such as problem questions, help documentation, participants involved, deployment of the two environments, types of analysis, data collection methods and methods to prevent biased data are described. Finally methods to analyze and determine the usability of the two environments are explained.

Chapter 4 describes the empirical evaluation used to verify the findings of this work. This includes graph visualizations that provide qualitative and quantitative evaluation on the comparison of the two programming paradigms.

Chapter 5 provides conclusions and possible future work.

Chapter 2

Related Work

A plethora of information is widely available to people in various ways, encouraging them to exploit the available data sources in order to solve some emerging problems. In order to access and manipulate this data, one will have to perform programming tasks. However, some people find it difficult to obtain a command over programming, mainly due to the strenuous efforts required to code on a general Text-based programming environment.

Flow-based and Block-based visual programming environments are developed to address these needs, where individuals without any programming background are able to perform tasks using these environments. Visual Programming Languages (VPLs) may or may not lack the power of Text-based programming languages, but they do not require the steep learning curve, hence they are better suited to the needs of users who have little or no training in computer programming.

2.1 Visual Programming Languages

Most visual programming languages fall into two categories:

Imperative Languages:

These are the Block-based languages and are known to keep the ‘state’ of a program. The state of where the program currently is and where the program should go next, continuously modifying the state. The mutation of state changes the flow of the program.

Functional Languages:

These are the Flow-based languages and are known for data flowing from one node to another. These languages are stateless and are reasoned about by just looking at a function’s input and output. Rather than assigning values which can then be mutated like what happens in imperative languages, the value returned by a function is only dependent on its input.

Discussions pertaining to the approach to be utilized towards programming between the two categories have been ongoing for some time. However, it is arduous to compare functional and imperative textually-based programming languages due to the structure and syntactic contrasts that tend to overwhelm any sort of direct examination. Since visual programming languages are basically syntax-free, it is more reasonable to make a comparison between the Flow and Block VPLs.

2.1.1 Background of VPLs

This sub-section gives a basic introduction to visual programming languages. The purpose is to lay a basic conceptual background which would develop a common understanding of how the VPLs function in general and how they are more efficient than the conventional programming languages.

Shu (1999) defines Visual Programming (VP) as a programming process which makes relevant use of graphical representations to create a program. In VP, the traditional method of textual coding is replaced with graphical representations such as figures, diagrams, etc. which simplifies the process of programming for users who do not have extensive computer training or knowledge of coding languages. The most focused users, while developing VPLs were either novice programmers or non-programmers by profession, who could gain advantage from programming.

Code represented graphically may be easier to understand as one is accustomed to the new representation. The advantage of writing a program in VPLs is that a program created using visual representations is analyzed and interpreted exactly the way it is visible on the computer screen.

K. Whitley (1997) mentions two principles that apply to both textual as well as visual notations. The first principle states that people respond better to information presented in a structured and a stable manner. Second, a competent representation would be the one that can make information explicit. However, a basic assumption by researchers while designing a VPL is that the visual notations enhance and improve human-computer interaction (HCI) while also catering to human cognitive abilities. Shu (1999) mentions that the use of graphics is more precise, self-explanatory, easy to comprehend, and allows for a longer memory retention period, which eventually makes programming more interesting and efficient.

In the investigations by Day (1988), instructions were presented in a matrix (visual) and list (textual) format for stroke patients directed to take certain medications. He was able to prove that the matrix representations were more efficient in comparison to the list because matrix allowed the subjects to comprehend information in two dimensions simultaneously (medicines to be taken as well as times of the day). Another similar research by Schwartz (1971) and Schwartz and Fattaleh (1972) reported that the matrix representation surpassed tree diagrams and sentence rewrites in cracking out problems on deductive reasoning. Polich and Schwartz (1974) also stated in their research that as the problem size became larger, the matrix representation proved to be more efficient than the textual one. Carroll, Thomas, and Malhotra (1980) ran an experiment where users were supposed to solve one of the two complex design problems that were isomorphic in nature: layout for a business office and scheduling problem. The authors were able to prove that there was a considerable alleviation in the performance of the two isomorphs, due to the usage of graphical representations in the second experiment as opposed to the first experiment that did not use the graphical representations.

Fix, Wiedenbeck, and Scholtz (1993) point out a considerable difference in the mental representation models between a novice and an expert programmer for a textual programming language. The authors also describe aspects of coding that are required to form an appropriate mental representation of a program:

1. The user should have the skill to recognize and link the basic recurring patterns.
2. The user should have the skill to comprehend the links connecting various modules of an application. These connections are important for a user to grasp, as they indicate the passing of data between different modules.
3. The user must also understand the hierarchy of the entire program as it is one of the essential attributes to the flow of the program.

For novices, such characteristics can only be obtained through extensive study and practice as described by Fix et al. (1993). According to an experiment by Ma, Ferguson, Roper, and Wood (2007) and Ma, Ferguson, Roper, Ross, and Wood (2009), novice programmers were known to retain non-feasible mental models for basic programming for an entire year using textual-based programming education. A small percentage of programmers who held feasible mental models had their performance graph higher than those with non-feasible mental models.

Such observations led to substantial research being conducted on the design and development of different types of VPLs which help users create program elements graphically rather than textually.

2.1.2 Classification of VPLs

This sub-section gives a general classification of the visual programming languages. It provides the different categories in which VPLs can be classified. Shu (1999) has classified VPLs into the following three types of categories:

1. Diagrammatic Programming Languages (DBL)

   This category makes use of graphical representations which produces machine-interpretable units on the execution of the program. These can also be used as extensions to the traditional programming languages.

2. Iconic Programming Languages (IPL)

   IPL is described to generate a flow of control by selection and connection of different icons that are used as representation for objects and actions.

3. Forms-oriented programming languages (FPL)

   This is recommended for users who have no aspirations to learn programming but need to create applications. Spreadsheets is an example for this approach. Users can perform a variety of data processing applications without the need to produce and develop algorithms or learn any programming concepts.

Boshernitsan and Downes (2004) present a more advanced classification scheme:

1. Purely visual languages

   In this category, a program can be created using graphical representations. The program is debugged and executed in the same visual environment and does not need any external textual code. Some of the examples include VIPR (Visual Imperative Programming), Prograph etc. VIPR is a pure visual programming language and makes use of concentric rings as means of representing a program. A state object is connected to the outermost ring and changes based on the actions in the innermost loop. Prograph is a pure VPL and object-oriented in nature. It is a combination of two powerful concepts of 'classes and objects' and 'dataflow mechanism'.

2. Hybrid text and visual systems

   This category is very similar to the 'Diagrammatic Programming Languages' category discussed above. This system is a combination of visual representations and textual underlying code. Programs here are translated into underlying textual code after they have been created graphically. An example is the Rehearsal World tool, a system which is trained by the user to solve problems by changing graphical elements. A Smalltalk program is then generated to implement the solution (Finzer & Gould, 1993). Another possible use includes visual element extensions on a textual language (Erwig & Meyer, 1995).

3. Programming-by-example systems

   In the paper proposed by Myers (1990), the author makes use of a new Visual Programming tool 'Peridot' (Myers, 1988, 1987) for creating graphical and interactive interfaces. This language makes use of the interesting concept of "programming by example". A picture of the required interface needs to be drawn by the user. This picture is then generalized by the system to generate a parameterized procedure. The user has to provide some example values for each parameter to enable the system in displaying a solid instance of the user interface. Another tool known as Pygmalion (Smith, 1975) is able to perform programming by example where the tool manipulates icons (concrete pictures) for data and programs to create a program.

4. Constraint-oriented systems

   These systems are used for simulation design. Physical objects are modelled by a programmer as objects in a visual programming environment. These objects mimic natural laws while being subjected to constraints. Thinglab (Borning, 1981) and ARK (Smith, 1986) are examples in this category. ARK (Alternate Reality Kit) enables users to create interactive simulations in a 2D animated environment. ARK allows users to control an on-screen hand that can interact with objects. These objects may either be interactors (representation of physical laws) or balls and blocks (possess speed and mass). This system attempts in simplifying the way laws interact with objects by connecting these abstract laws with physical reality. Thinglab is a system that revolves around objects that send and receive messages.

5. Form-based systems

   These systems are similar to the Forms-oriented programming languages discussed above and are comparable to spreadsheets. One of the examples of this system is Forms/3 (Burnett, 1995). Forms/3 is an object-oriented visual programming language. It is similar to how a spreadsheet works, with the cells and formulas representing data and computation respectively.

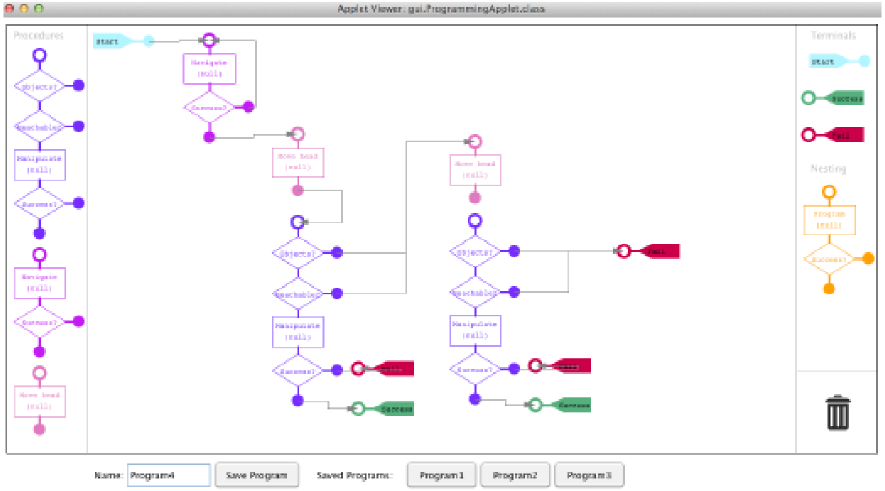
2.1.3 Examples of Flow-based VPLs

This section illustrates a few examples of the many Flow-based visual programming languages developed over the past years. They include RoboFlow (Alexandrova, Tatlock, & Cakmak, 2015), PureData ("Pd Community Site," 1990), Quadrigram ("Data visualization & presentation tool," 2005), and Flowgorithm ("Flowgorithm - Flowchart Programming Language," 2014). Each of these are briefly explained to better understand the concept, use and design of Flow-based representation.

RoboFlow

Alexandrova et al. (2015) propose and develop a Flow-based visual programming language - 'RoboFlow' designed specially for the end-users who will be able to program a robot for a particular environment only and configuring everyday objects within that environment. Simple tasks needed by humans including 'pick up', 'put away', 'organize' and many others are initiated distinctly in every home. Hence, RoboFlow offers a dense and extensible VPL enabling a programmer to tailor programs according to its environment. The programming paradigm here allows the user to deal with graphical interactions and physical demonstrations.

A box-line representation is used here. Boxes represent procedures or functions having inputs and outputs. Data flows from one box to another through a line. Figure 2.1 shows an overview of RoboFlow.



Name: Program4

Figure 2.1: Overview of the RoboFlow editor (Alexandrova, Tatlock, & Cakmak, 2015).

A user study conducted with three user groups using RoboFlow concluded the two groups ‘roboticists’ and ‘programming language experts’ stating that using a general-purpose programming language would be preferable rather than a VPL like Roboflow, whereas, the ‘non-programmers’ preferred coding in RoboFlow (Alexandrova et al., 2015).

PureData (Pd)

This is a Flow-based visual programming language originally developed by Miller Puckette in the 1990s. It is an open source language that can run on personal computers, Raspberry Pis and smart phones. Interactive computer music and multimedia works are the key weapons in its arsenal. Sound, video, 2D/3D graphics, interface sensors, input devices, and MIDI can be produced and generated using Pd. Musicians, performers, visual artists, developers and researchers primarily use this language to develop software graphically with no need of writing lines of code ("Pd Community Site," 1990).

The visual representations used here are visual boxes called objects that are placed in a 'canvas' and the entire program created is called a 'patch' which is shown in Figure 2.2. Data flows from one object to another using visual connectors called 'cords'. Users can change a behaviour of any object at any time by connecting them in a different way leading to the reusability feature. Though it is so efficiently used by musicians, artists etc. to create music, it is difficult to create massive parallel processes in puredata. The arrays and other entities here are also susceptible to name space collisions. However, Pd is still suited for complex systems for large scale projects ("Pd Community Site," 1990).

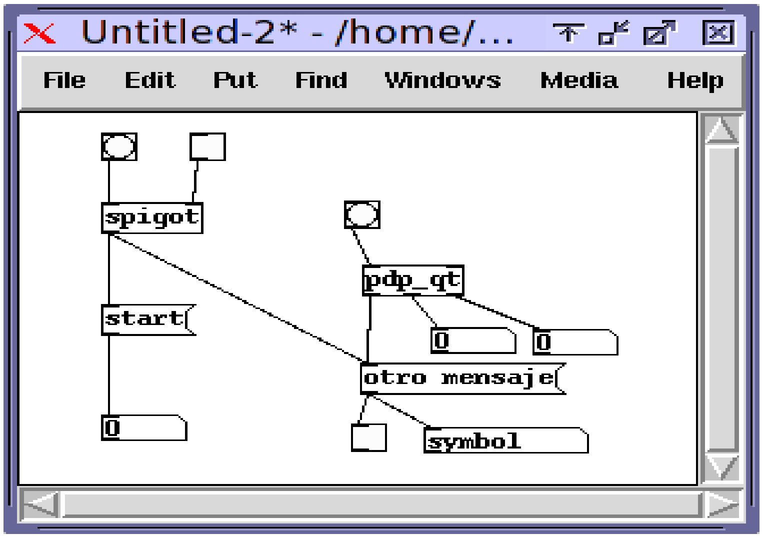


Figure 2.2: A sample program in PureData also known as 'Patch' ("Pd Community Site," 1990).

Figure 2.3 shows an example program of PureData in which the program generates music continuously, and according to the different beats of the music, the visualization is produced in real-time ("Pd Community Site," 1990).

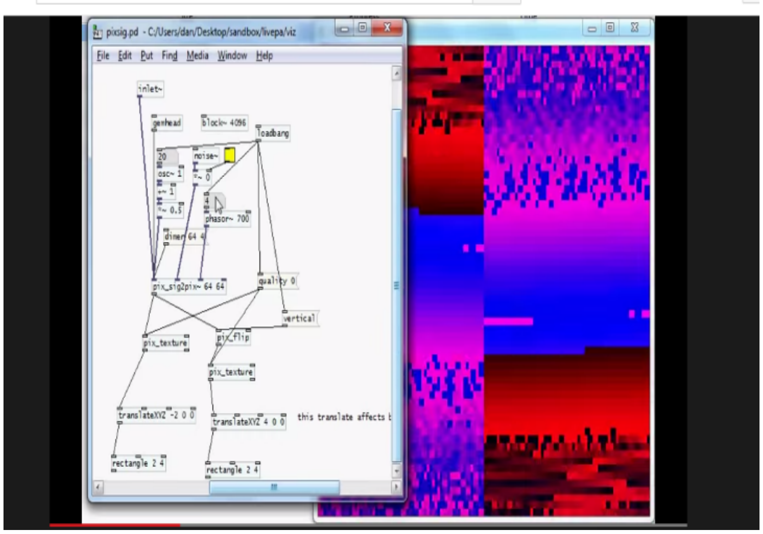


Figure 2.3: Example program in Pd with the output (on the right side) (“Pd Community Site,” 1990).

Interestingly Pd is utilized for live music performances. It has a unique feature of getting immediate feedback as soon as the user completes writing a program, as there is no distinction between writing a program and running a program. Hence, Pd programming tends to bring a user much closer to manipulation of objects in the real physical world. This also makes live performance a favourable situation for artists (“Pure Data,” 1990).

Quadrigram

This is another Flow-based visual programming language with a drag-and-drop programming paradigm. It is designed to make the practice of data analysis and data visualization more universal and helps in the creation of powerful customized visualizations. This environment basically lets the user create and share interactive visualizations. It eliminates the need for any programming skills in the development of data solutions. Figure 2.4 shows the development environment of Quadrigram, which has small connectors as visual representations. These connectors are connected to each other to form modules. Data is processed through these modules to create a final processed output which is a visualization graph also shown in

Figure 2.4 (“Data visualization & presentation tool,” 2005).

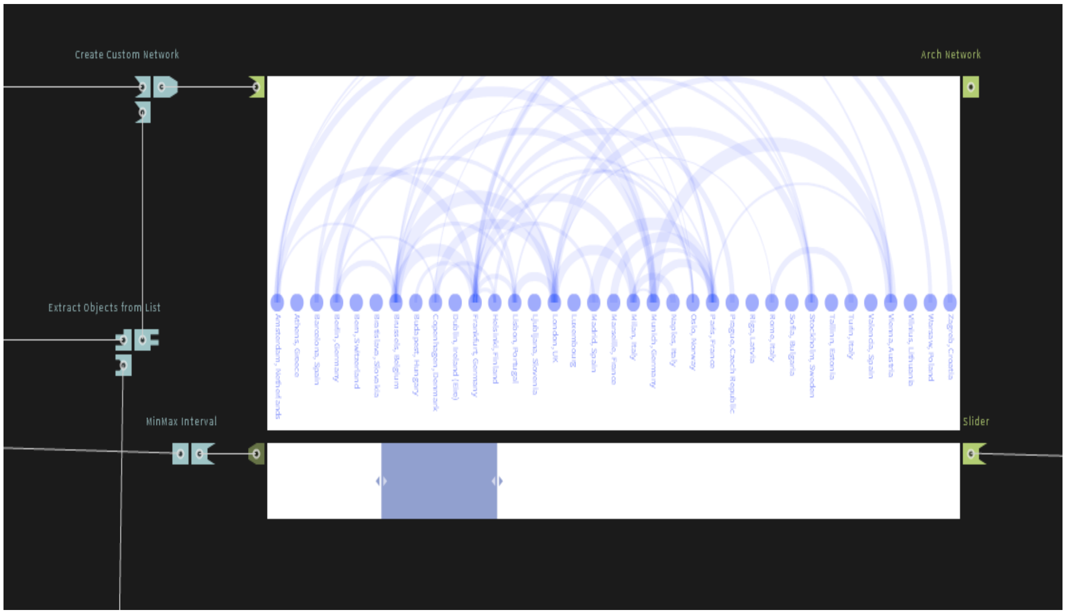


Figure 2.4: Development environment of Quadrigram (left hand side), Final output (right hand side) (“Data visualization & presentation tool,” 2005).

Another example of a visualization graph created by Quadrigram visual programming language is shown in Figure 2.5.

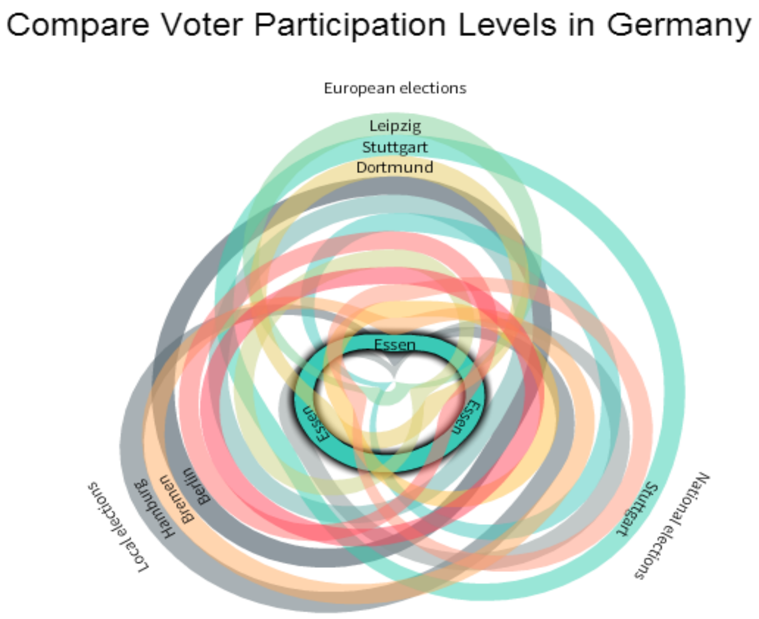


Figure 2.5: An example visualization produced by Quadrigram ("Data visualization & presentation tool," 2005).

Quadrigram has an extensive catalogue of visualizations that can be connected, interconnected and customized to visualize any dataset. It also enables users to share interactive data visualization projects ("Data visualization & presentation tool," 2005). Quadrigram provides an opportunity to manipulate data in real time that is constantly getting altered. This kind of data can come in from sensor feeds, social network, surveys etc. Quadrigram is a tool that is useful to anybody who has knowledge in the manipulation of data and is not just for data scientists.

Flowgorithm

Flowgorithm is a Flow-based visual programming language that helps in creating programs using flowcharts. One can make use of the given shapes to create a program. These shapes constitute the different actions that the program needs to execute, which removes the difficulties of a Text-based programming language. At a high level, a programmer can also export their flowcharts into different Text-based languages like Java, Python, Perl, C++ and many others. Flowgorithm has various useful features such as multilingual support, safe recursion, graphical variable watch window etc. A short example to better understand the working of Flowgorithm is explained below. It is the traditional "Hello, world" program ("Flowgorithm - Flowchart Programming Language," 2014).

Figure 2.6 below opens up on starting a new flowchart. It consists of two rounded rectangles called 'terminals' representing the start and end of a program. Since every program in Flowgorithm is like a flowchart and flowchart always consists of shapes, one can add different shapes with different functionalities to create a program. In Figure 2.7, to add a new shape between the terminals, mouse pointer has to be moved over the line in between them. A shape is permitted to be added if the line turns orange ("Flowgorithm - Flowchart Programming Language," 2014).

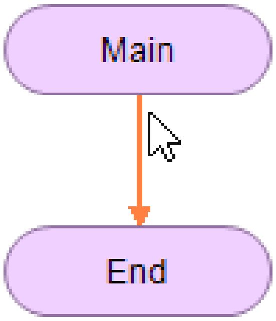


Figure 2.6: Start a new flowchart ("Flowgorithm - Flowchart Programming Language," 2014).

A pop-up menu opens up on either double-clicking or right-clicking on the line as shown in Figure 2.7.

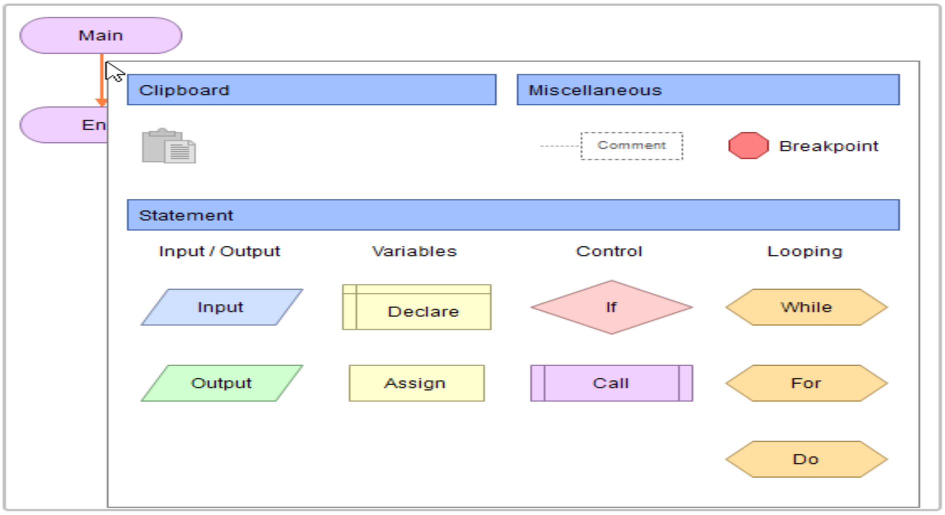


Figure 2.7: Add a new shape ("Flowgorithm - Flowchart Programming Language," 2014).

Every action performed by the computer is represented by a different shape. Adding the 'Output' shape will result in Figure 2.8.

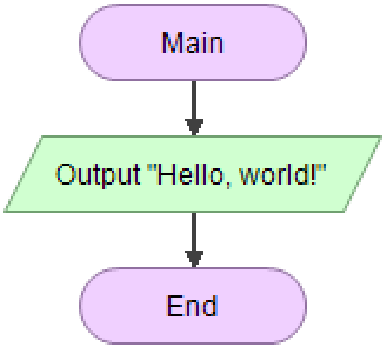


Figure 2.8: Add the Output shape ("Flowgorithm - Flowchart Programming Language," 2014).

Another window called 'Console Screen' is used for displaying textual information as shown in Figure 2.9.

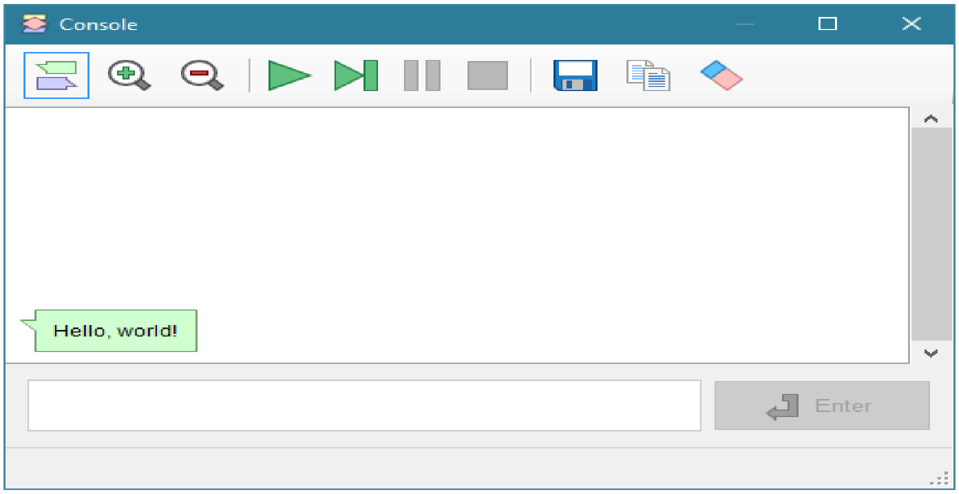


Figure 2.9: Console Screen ("Flowgorithm - Flowchart Programming Language," 2014).

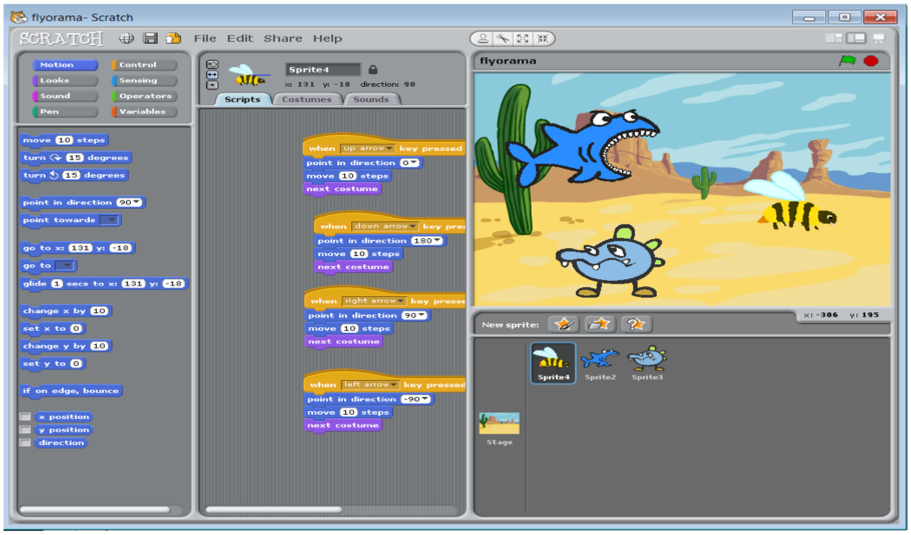
2.1.4 Examples of Block-based VPLs

There are many Block-based VPLs developed over the past years. Some of them are Scratch ("Scratch - Imagine, Program, Share," 2007), Blockly ("Blockly Google Developers," 2012), App Inventor ("MIT App Inventor," 2015), Alice ("Alice.org," 2007), Waterbear ("Waterbear," 2011), Snap ("Snap! (Build Your Own Blocks) 4.0," 2011) etc. Few of these are discussed in this section to better understand the concept of Block-based programming languages.

Scratch

This is a visual programming language created by Lifelong Kindergarten research group of the MIT Media Lab to help learn programming. Users in the age group of ages 8 to 16 are mostly an audience to this language (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008). Scratch helps in the development of media-rich programs using a mix of multimedia elements like audio, graphics, video etc. to create a new project. There have been a wide variety of projects created by users which includes greeting cards, music videos, tutorials, science projects etc. Drag-and-Drop is the programming paradigm used in Scratch. Figure 2.10

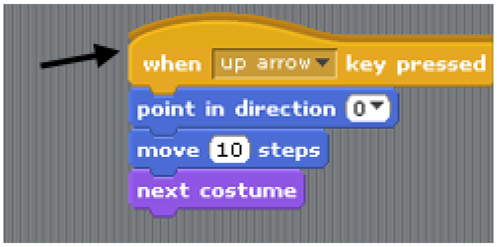
shows the development environment of Scratch.



```
when up arrow key pressed
point in direction 0
move 10 steps
next costume

when down arrow key pre
point in direction 180
move 10 steps
next costume

when right arrow key presse
point in direction 90
move 10 steps
next costume

when left arrow key pressed
point in direction -90
move 10 steps
next costume
```

Figure 2.10: Scratch Development Environment (“Scratch - Imagine, Program, Share,” 2007).

Each object here is called a ‘sprite’ (2-D graphical object) situated on a background called the ‘stage’, and each scenario is called a ‘script’. Each object can have one or more scenarios connected to it. The different scenarios/programs can be created by joining the required colorful blocks together like a puzzle as shown in Figure 2.10 . Users can drag the blocks they need from the panel on the left-hand side and drop it on the middle panel (called scripts area) to create a program (Papadakis, Kalogiannakis, Orfanakis, & Zaranis, 2014). The program/script starts running by either clicking on any block in the stack of blocks created (shown in Figure 2.11) or by clicking on the green flag in the display area (shown in Figure 2.12). Click on the red stop button next to green flag to stop the running script (“Scratch - Imagine, Program, Share,” 2007).



```
when up arrow key pressed
point in direction 0
move 10 steps
next costume
```

Figure 2.11: One way of running the program (“Scratch - Imagine, Program, Share,” 2007).

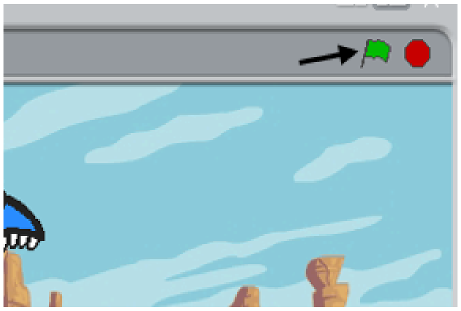
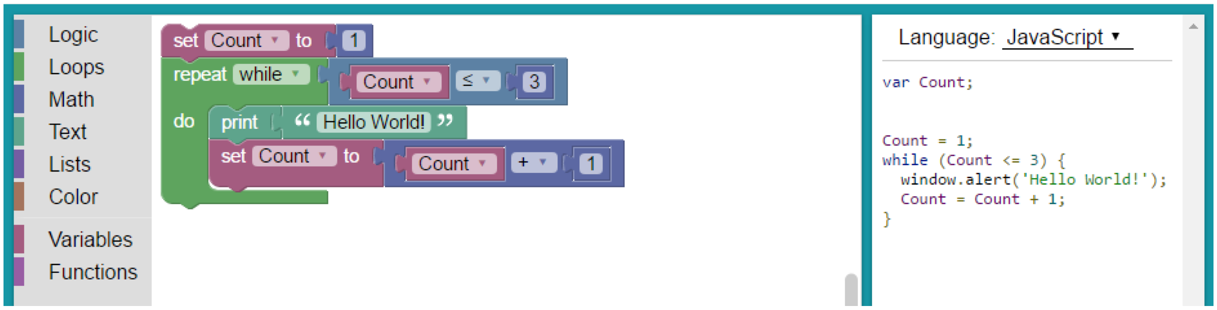


Figure 2.12: Other way of running the program (“Scratch - Imagine, Program, Share,” 2007).

Blockly by Google Developers

Blockly is another open source Block-based visual programming language, that can be customized. It is a pure JavaScript library that is compatible with major browsers: Chrome, Firefox, Safari, Opera, and IE. In a browser, Blockly supports inclusion of a visual code editor for many languages such as JavaScript, Python, PHP, Lua, Dart etc.

There are two perspectives to the Blockly app - user and developer. From the user’s perspective, Blockly helps to code in a simple manner using blocks. One can use the Blockly editor which consists of three parts - toolbox: that stores block types; workspace: the place where user can arrange blocks and create a program; and language: shows Text-based equivalent program (of the selected language) of the Block-based program coded in the workspace (“Blockly Google Developers,” 2012). This can be seen in Figure 2.13 below:



Language: JavaScript

```
var Count;


Count = 1;
while (Count <= 3) {
  window.alert('Hello World!');
  Count = Count + 1;
}
```

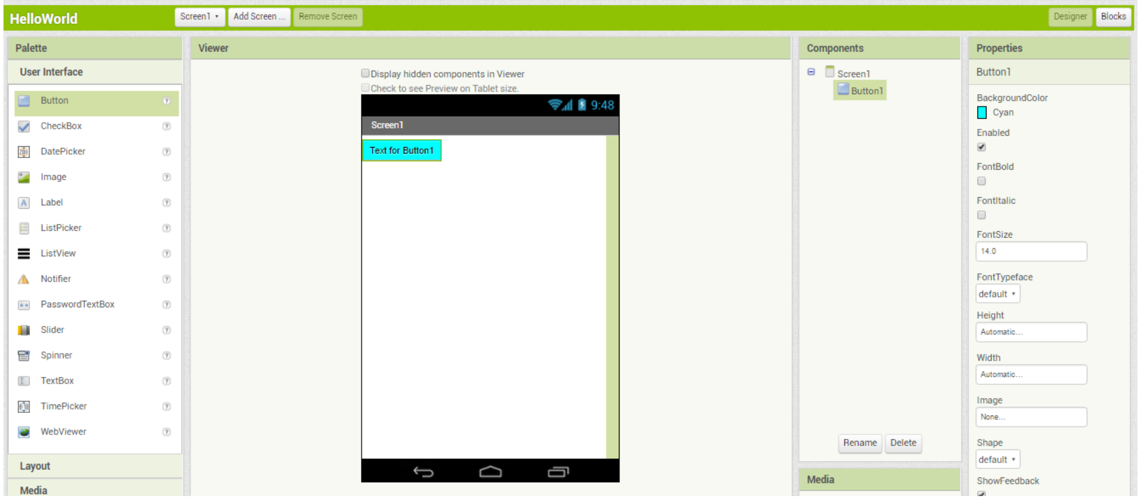
Figure 2.13: Blockly Development Environment (“Blockly Google Developers,” 2012).

From the developer's perspective, building a Blockly app can be divided into various steps: integrate the Blockly editor, create blocks for users and add them to the toolbox ("Blockly Google Developers," 2012).

One of the big advantages of using Blockly is that the user/developer can convert the Block-based code into conventional programming languages. This helps in the smooth transition to a Text-based programming language. Blockly can implement easy as well as complex programming tasks. It has been translated to 40+ languages, making it a diverse tool ("Blockly Google Developers," 2012).

App Inventor

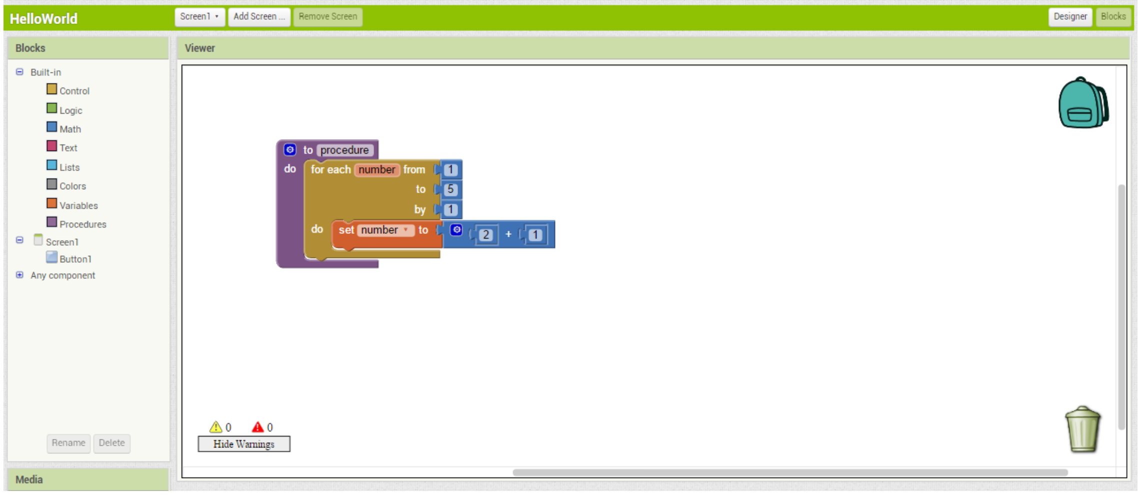
App Inventor (AI) is an open source web environment used to create mobile applications for Android devices (Papadakis et al., 2014). This is built by MIT. It leverages a Block-based visual programming language to create apps. AI uses Blockly visual programming editor (discussed in the above section) to construct the blocks (to be used by the user) in the browser. One does not have to be a professional developer to create applications using App Inventor. It uses a graphical interface similar to Scratch. AI consists of two main parts: 'Designer' for designing the user interface of the entire application which is portrayed in Figure 2.14 and 'Blocks Editor' where the user can create a program by connecting blocks or tiles showing the application's behaviour. This can be seen in Figure 2.15. The visual representations used here are very similar to Scratch.



Button1

BackgroundColor: Cyan
Enabled: ☑
FontBold: ☐
FontItalic: ☐
FontSize: 14.0
FontTypeface: default
Height: Automatic...
Width: Automatic...
Image: None...
Shape: default
ShowFeedback

Figure 2.14: Designing stage of AI ("MIT App Inventor," 2015).



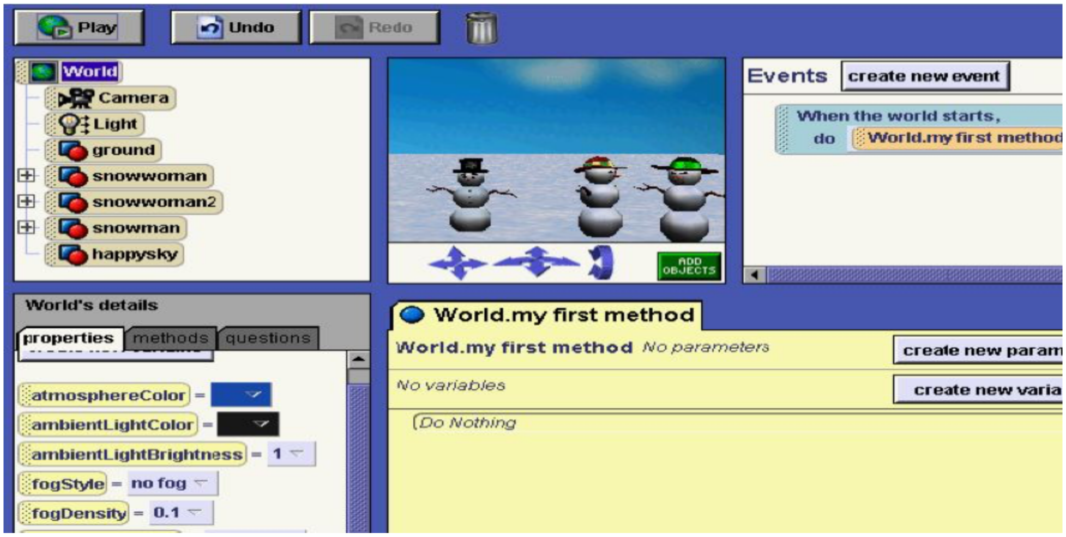
```
to procedure
do  for each number from 1
                     to 5
                     by 1
    do  set number to 2 + 1
```

Figure 2.15: Block Editor of AI ("MIT App Inventor," 2015).

AI has two main versions: App Inventor 1, also known as App Inventor Classic launched in 2009 and App Inventor 2 released in 2013. The former ran its blocks editor in a separate Java application. The latter runs its block editor in a web browser (Xie, Shabir, & Abelson, 2015; "MIT App Inventor," 2015). Since the users do not require any coding ability, AI has attracted thousands of non-programmers either for educational purposes or just to broaden their knowledge in the computing world (Wolber, Abelson, & Friedman, 2015).

Alice

Alice is a 3D interactive animation programming environment. It consists of 3D objects that populate a virtual world and users develop a program to animate those objects. The goal of this language is to make it easy for novices to develop interesting 3D environments. Figure 2.16 shows the Alice development environment.



When the world starts, do World.my first method

World.my first method No parameters

No variables

Do Nothing

atmosphereColor =
ambientLightColor =
ambientLightBrightness = 1
fogStyle = no fog
fogDensity = 0.1

Figure 2.16: Alice Programming Interface ("Alice.org," 2007).

An animation can easily be created such as presenting a story, playing an interactive game and many others without any programming knowledge. Users can drag and drop graphic tiles to create a program. Similar to other Block-based environments discussed above, users can immediately see the behaviour of their coded program thus enabling them to better understand the relationship between the code and the object behaviour. Alice is claimed to be a teaching tool to learn object-oriented programming for novice users ("Alice.org," 2007; Weerasinghe & Cohen, 2012).

2.1.5 Usability of VPLs

Green and Petre (1996) gave some useful insights on the HCI (Human-Computer Interaction) part of programming. They highlighted some major aspects of programming such as ways in which a programmer connects to the code and how important it is to have a successful and a practical way to measure the code management. Code management in visual environments would include comprehension of layout, control of visual elements, mapping of code to the right entities and reusability of pieces of code. They proposed new cognitive dimensions in their research that would help evaluate the usability of visual programming environments. Few of those dimensions (related to this research) are discussed below:

“Abstraction Gradient”

Abstraction captures the essence of what a particular function does. Considering abstraction as challenging and a high educational achievement in computer programming can make it difficult to implement for a novice programmer, as their mental models are not trained to immediately map a problem entity onto a program entity and create a suitable abstraction for it.

The advantage of a visual programming language in terms of abstraction as compared to a Text-based programming language is that the former has the facility to create and use visual data abstractions having low overhead and easy comprehension, whereas the latter would have to create and use libraries that would have a high overhead as the user needs to understand how to create or use those existing libraries. In VPL, users do not have to worry about programming entities like hierarchical rules, inheritance trees, pointers etc. thus reducing error, though every visual programming environment has a different level of abstraction.

“Closeness of Mapping”

Mapping a problem entity onto a program entity can be achieved easily, if the programming world is closer to the problem domain, and if the representation of the program operations is intuitive and simple to use.

VPLs are very efficient as users are able to map a problem entity onto a program entity which helps in the rapid solution of the problem.

“Consistency”

A programming language whose structure and syntax are consistently maintained fall under this category. For example, if a user has knowledge about parts of a language syntax and structure, it will be relatively easy to determine some other similar parts in the language successfully.

VPLs in this regard have been successful because VPLs do not require the user to remember any syntax, as compared to the primitive languages.

“Diffuseness/Terseness”

Different programming entities termed as lexemes are preferable to be fewer, to hold them in a user’s memory for a longer time. Higher number of lexemes would cause the user to perform recurrent searches.

VPLs have a good number of lexemes in terms of icons, connectors, etc. It is hard to conclude the terseness of VPLs in general, as every VPL has different number of lexemes.

“Error-proneness”

Green and Petre (1996) define ‘slip’ as doing something that is not meant to happen. These slips are not uncommon in a Text-based language with high syntactic design features. One of the frequent mistakes or slips that a user can make is attaching wrong identifiers to a variable.

This is highly unlikely in a visual programming language as most of them do not require for a user to remember or type any identifier. Also, the structure of a VPL is quite different from a primitive language, hence reducing the slips caused.

"Hard Mental Operations"

Green and Petre (1996) mention how a programming language should avoid mind-boggling operations. These include two properties:

The first property: For example, a conditional loop is in itself a tough concept for a user to comprehend and if its notation is not rightly denoted, it would be a hard-mental operation for the user. On the other hand, if the conditional notation is denoted in a decision table or some other format (Curtis, Sheppard, Kruesi-Bailey, Bailey, & Boehm-Davis, 1989; Wright & Reid, 1973), a huge amount of difficulty would be diminished as most part of the issues are at the notational level and not at the semantic level.

The second property: This states that if many of the offending objects are co-occurring, the user would face difficulty such as nested negotiations.

Green and Petre (1996) point out with a few examples of how hard mental operations were not so uncommon in VPLs, especially with the control constructs.

"Hidden Dependencies"

If the dependency between two entities that are dependent on each other, is not noticeable, it gives rise to a hidden dependency. This is an issue which persists in the conventional languages for a long time, in which cross-references and call-graphs are used to avoid the hidden dependency.

In VPLs, the visual elements count towards lessening the hidden dependencies. This is because most of the dependencies are visible to the user due to VPL's visual nature.

"Premature Commitment"

Premature Commitment takes place when the user is enforced in making decisions before the required information is made available. This could happen under several conditions:

- The environment has certain constraints that the user needs to follow.
- The order is improper.
- Internal dependencies surround the existing notations.

VPLs are extremely flexible and does not restrict the user to follow any order in coding.

“Progressive Evaluation”

A usual standard in most environments consists of naïve programmers to assess their own progress periodically in problem-solving.

Block-based and Flow-based programming environments have an option for the users to execute a program at any point to understand the progress of their program.

“Role-Expressiveness”

This feature mainly revolves around the question of a programming language being easy or hard to read. This can be determined by using meaningful identifiers, separation of related functions into modules, and appropriate abstraction. (Teasley, 1994; Green & Petre, 1996).

“Viscosity: Resistance to Local Change” A programming environment needs to have a facility to make changes at a global level which would automatically reflect at specific levels within the program. That is low viscosity and is known to be better.

In VPLs, viscosity is low as compared to the Text-based languages. Either the user needs to rearrange the blocks in a Block-based system or rewire the components and wires in a Flow-based system and the changes are reflected immediately, which can be seen via the visual feedback.

The above discussed dimensions will be analyzed in more detail with respect to the Flow-based and Block-based environments with regard to the usability results achieved in Chapter 4.

2.2 Flow-based Programming Concepts

Before discussing Flow-based programming concepts in detail, another conceptually similar language known as 'Communicating Sequential Process (CSP)' will be described.

2.2.1 Communicating Sequential Process

CSP was invented by Tony Hoare in 1977. Hoare describes CSP as a language that allows interaction between different processes through message-passing communication in concurrent systems ("Communicating sequential processes," 1977). Hoare introduces the concept of an input and an output by describing them as basic primitives of programming and calls synchronization as one of the main aspects of CSP. Multiple concurrent processes are allowed to communicate with each other by synchronizing their input and output (Hoare, 1978).

In CSP, communication takes place through channels via messages passed on to the channel (Hoare, 1985). As seen from figure 2.17, Let P1 and P2 be processes, and C be the channel between P1 and P2. Channel C will act as an output channel for P1 and as an input channel for P2. Communication takes place on channel C, when a message with some value is output from P1 to channel C and the input process P2 accepts a message with the same or a modified value from Channel C. One cannot see the internal working of the channel but they know what function the channel performs (Hoare, 1985).

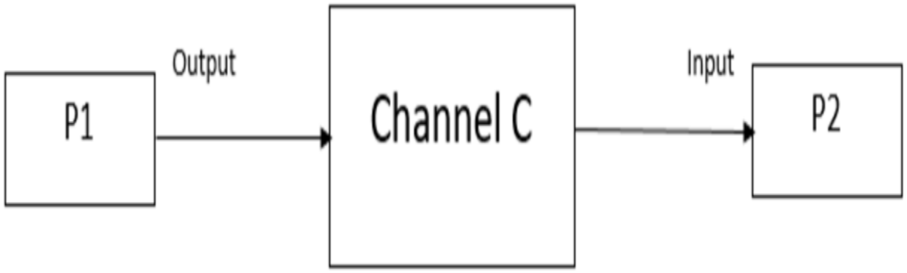


Figure 2.17: Communicating Sequential Process

2.2.2 Flow-based programming

Flow-based programming (FBP) is a visual programming paradigm invented by J. Paul Morrison in the early 1970s. He defined this framework to be a network of “black box processes”, that communicate with each other by passing data as messages. FBP is a methodology which aims at code reusability and asynchronous processing (Morrison, 1994).

A major advantage of this framework is that these black boxes can be rearranged in numerous ways, which gives rise to component reusability. Thus, instead of building everything from scratch, one can build using pre-existing components making FBP cost-efficient (Morrison, 1970, 2005, 2012).

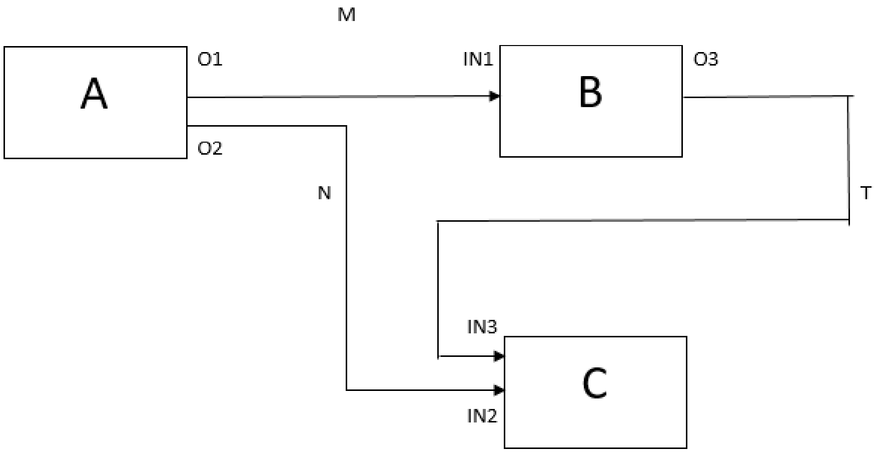


Figure 2.18: A Simple FBP Diagram.

Figure 2.18 shows a simple FBP diagram, that has processes (A, B, and C). Each process can have either input or output ports. In this case, Process A has two output ports O1 and O2, Process B has an input port IN1 and an output port O3, and Process C has two input ports IN2 and IN3. These processes are connected to each other via connectors (bounded buffers) - M, N and T.

Data is passed as Information Packets (IP). At any given time, each IP is either owned by a single process, or is being transferred from one process to another. When any process receives an IP, the code of the process is used to manipulate the IP. This code assigned to the process is not visible to the outside world, thus making a process similar to a black box.

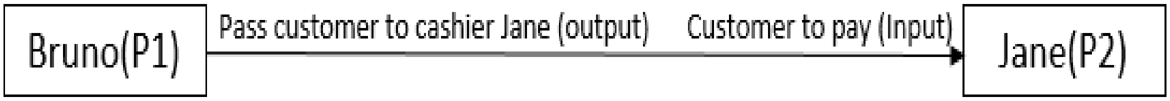
The current process outputs the IP through its output port to transfer it to the next process. The capacity of allowed IPs at a time for a given connection are fixed. Also, processes are provided with exclusive access to other resources such as their own storage space, parameters, control blocks, etc. and these cannot be shared (Morrison, 1970).

Morrison (1994) explains that FBP utilizes the designers' cognitive skills, while also supporting progressive stepwise development. He also states that applications developed using

FBP have run dependably for long stretches (nearly 40 years), while undergoing consistent alteration because of changes in equipment, programming, business and administrative areas (Morrison, 1994).

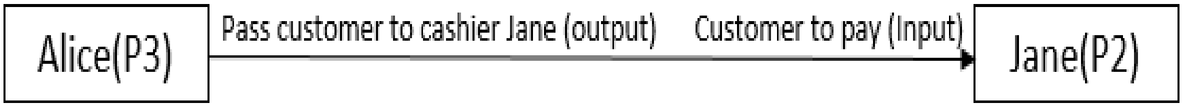
The relation between processes is cooperative rather than hierarchical. FBP gets the tasks done rapidly and easily by delegating tasks from one block to another. Consider a real world example to understand the FBP approach:

Assume 'Company A' owns a clothing outlet store, where they only sell winter clothes. Jane is the cashier and Bruno is the salesman, who helps people find the right winter outfits for them. Once Bruno is successful in helping the customer find the outfit, he will delegate the task of making the payment to Jane.



From our observations above, we notice that Bruno and Jane are processes. When the task was delegated from one person to the other, it formed a connection between the two. The efficiency of the FBP can be pinpointed by enhancing the example further:

Let's assume that the winter season got over, and the spring season started. Taking that into consideration, the company introduced spring clothing in their store. Assuming Bruno is not equipped with the right knowledge to handle spring clothing, the corresponding customers cannot be linked to Bruno. Hence Alice is recruited by the company to handle the spring clothing buyers. It can be noticed that the design can be changed, whenever required, to add or remove components without affecting the other unrelated components. In this example, while Bruno is replaced by Alice, the cashier Jane is unaffected in both the seasons.



FBP can form numerous patterns by simply changing connections of blocks without changing the code in the blocks. This characteristic of Configurable Modularity provides an advantage to engineer the code. FBP has been implemented successfully using programming languages like Java known as JavaFBP, C# known as C#FBP, and C++ known as CppFBP (Morrison, 1970). Several projects have been built using the FBP paradigm to introduce FBP concepts to people. Some of them are:

1. Quadrigram: http://www.quadrigram.com/.
2. Flowgorithm: http://www.flowgorithm.org/.
3. LabView: http://sine.ni.com/psp/app/doc/p/id/psp-357.
4. NoFlo: https://noflojs.org/.
5. Pure Data: https://puredata.info/.

2.3 Block-based Programming Concepts

Block-based programming environments have turned into a generally utilized instructive stage in an educational domain for novices to figure out how to program. The Block-based paradigm has been progressively grasped by domain specialists to create end-user programming (Techapalokul & Tilevich, 2015). This is especially evident with the development of another era of tools such as Scratch ("Scratch - Imagine, Program, Share," 2007), Snap! ("Snap! (Build Your Own Blocks) 4.0," 2011), and Blockly ("Blockly Google Developers," 2012).

In Block-based programming, a sequence of blocks grouped together form a lexical structure of a program. A block can either be a declaration block, a statement block, control structure block, math block etc. Block-based programming languages use a programming-primitive-as-puzzle-piece metaphor and drag-and-drop mechanism that gives visual signals to the user about how and where commands can be utilized. Learners can gather working programs utilizing just a mouse, by snapping together instructions and getting visual (and in some cases audio) feedback educating the user if a given development is legitimate. In the event that two blocks cannot be connected to shape a substantial syntactic articulation, the environment keeps them from snapping together, therefore restraining any syntax blunders, however, holding the act of constructing programs in a sequential manner. This component is particularly applicable in this review, as graphical programming advocates claim that the absence of language syntax is a key element that adds to its propriety for youthful learners (Resnick et al., 2009).

Alongside utilizing the shape of the block to signify use, there are other visual signals to help novice developers, including different coloured blocks or the use of natural language labels on the block to indicate its function. Blocks can either be coloured according to their respective functions or by its various categories. The categories are easily navigable by the users. These visual features make Block-based computer programs simpler to use and relatively convenient for the users.

LogoBlocks (Begel & Resnick, 2000) and BridgeTalk (Bonar & Liffick, 1987) programming environments were the early forms of the Block-based programming paradigm, with LogoBlocks being the first blocks language developed in 1995, which was then referred to as “building-blocks programming”. These two languages defined the Block-based programming approach that has since become utilized as a part of many well-known programming environments like Alice (“Alice.org,” 2007), Scratch (“Scratch - Imagine, Program, Share,” 2007) and many others that are broadly classified as a part of introductory programming courses. However, it was only after the development of Scratch that the blocks method or technique truly became well known (Vasek, 2012).

Block-based environments gave learners open-ended, exploratory spaces intended to bolster imaginative and creative exercises like story narration and game building. Notwithstanding to the common interfaces that the Block-based environments share, they also share qualities such as targeting novice programmers particularly students in their primary or secondary school, reflecting the syntax and structure of textual programming languages and focussing on programs that have relevance in a user's day-to-day life (Price & Barnes, 2015).

Not only has it been utilized as a part of more traditional software engineering settings, a developing number of situations have implemented the Block-based programming approach to lower the hindrance to programming over a variety of spaces incorporating modelling and simulation tools like StarLogo TNG (Begel & Klopfer, 2007), mobile application development with MIT App Inventor ("MIT App Inventor," 2015), artistic tools such as Turtle Art (Bontà, Papert, & Silverman, 2010), game-based programming environments such as RoboBuilder (Weintrop & Wilensky, 2012), commercial educational applications like Hopscotch, and Google's Made with Code and Code.org's Hour of Code activities.

All of these new Block-based environments additionally fortify the need to better comprehend the intellectual and effective affordances of the methodology (Weintrop, 2015b, 2016).

Nonetheless, as users pick up experience and begin making bigger programs, they experience two badly arranged properties of Block-based languages: blocks consume more screen space than textual languages and dragging blocks from a palette is slower than writing text (Brown, Mönig, Bau, & Weintrop, 2016).

Despite the fact that these environments are prevalent for having a "low floor" and are simple to begin with, programs written in Block-based languages frequently end up noticeably cumbersome as programs become more mind boggling (Techapalokul & Tilevich, 2015).

However, there are many reasons why instructors like using Block-based programming environments:

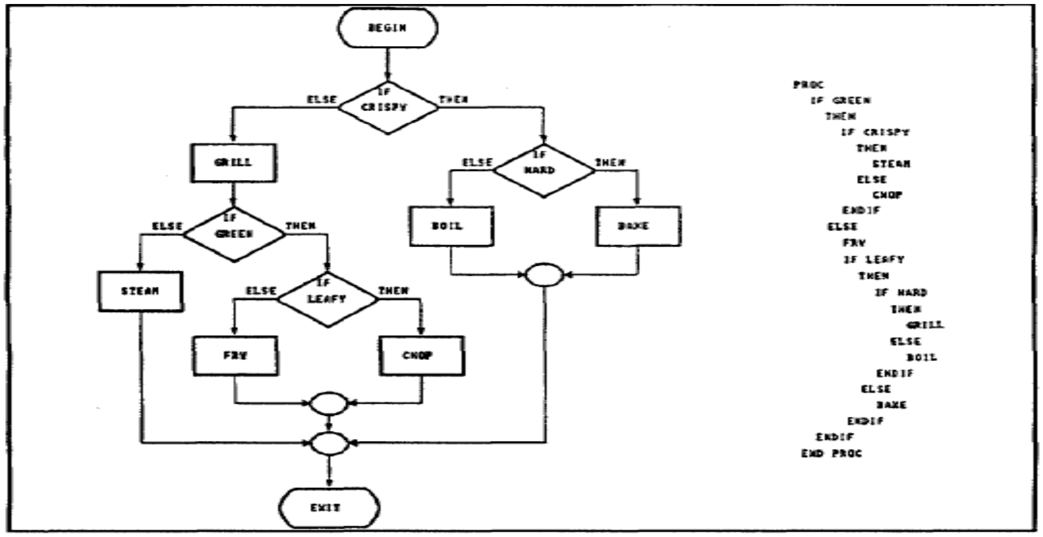
1. Low boundary of passage: One has to basically drag and drop blocks to create and execute their program. This helps both educators and students.
2. Low student disappointment: Absence of syntax blunders are dependably accompanied with Block-based programming. This also helps both instructors and students. Instructors, who do not have a major in Computer Science degree at school, can grasp and learn Block-based coding generally quickly.
3. Block-based environment helps in learning programming concepts such as control structures, operators, events etc. It also cultivates experimentation. Block-based coding is easy to the point that it asks its users to simply attempt a few blocks to see what happens. This encourages innovativeness (“What is Block Based Coding?” 2014).

2.4 Comparisons

2.4.1 Comparison between Text-based and Flow-based languages

This section starts with the work done by early researchers in comparing flowcharts with pseudocode. A flowchart makes use of a diagram to represent an algorithm or a process. This diagram has boxes of different shapes representing different functions connected by arrows (“Flowchart,” 1921). Pseudocode describes a computer program in an informal high-level manner and although it has conventions of a traditional programming language, it is designed for human reading rather than machine reading (“Pseudocode,” 2003). Basically, a flowchart has a Flow-based visual programming language like structure and pseudocode has a Text-based programming like structure.

A study conducted by Ramsey, Atwood, and Doren (1983) compares flowchart representation with a pseudocode representation (Program Design Languages (PDL)). The experiment in this paper consisted of 20 students in a computer science graduate course. It was indicated in the results that PDL proved to be better than flowcharts. PDL turned out to be more precise in terms of algorithmic detail and variable names were used with less abbreviation. An example of this comparison can be seen in Figure 2.19.



```
PROC
  IF GREEN
    THEN
      IF CRISPY
        THEN
          STEAM
        ELSE
          CHOP
      ENDIF
    ELSE
      FRY
      IF LEAFY
        THEN
          IF HARD
            THEN
              GRILL
            ELSE
              BOIL
          ENDIF
        ELSE
          BAKE
      ENDIF
  ENDIF
END PROC
```

Figure 2.19: Comparison between Flowcharts (left hand side) and PDL (right hand side) (Ramsey, Atwood, & Doren, 1983).

Another similar research by Scanlan (1989) compared flowcharts with pseudocode. However, the results indicated structured flowcharts to be more usable than pseudocode in terms of algorithmic comprehension. These algorithms were already designed as opposed to the research by Ramsey et al. (1983).

This early research gave rise to an evolution that led the later researchers in comparing a Flow-based VPL with a Text-based VPL. T. Green, Petre, and Bellamy (1991) claim that there have been studies which compare flow-charts or structure charts with a textual language, but this is the first study comparing a data-flow VPL with a textual language. This paper investigates two major hypotheses:

1. Hypothesis A: "the Superlativism hypothesis" - Since the 2D visual elements are easier to perceive, VPLS are good in a natural manner rather than the text format.

2. Hypothesis B: “information accessibility” - As the name suggests, the information is accessible more easily in a VPL than a text language with a particular way of designing notations. This will make working on some tasks easier while some other tasks may not gain an advantage from it.

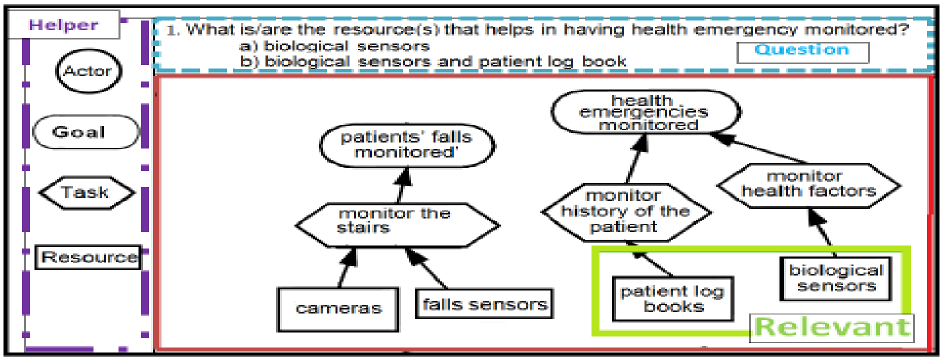
There were five users in this study, each having 6 or more months experience with LabView VPL and their overall programming experience from 5 to 15 years (included many different programming languages). The users were domain experts and not professional programmers. The results indicated that the hypothesis A and hypothesis B did not turn out to be so true and text was more favoured than graphics. However, the subject size was too small and better results would be generated with a larger sample size.

Another research was conducted to assess which of the two notations was better suited to estimate the output of a program, sequence of program statements in terms of execution, and identification of errors in the code. Whitley, Novick, and Fisher (2006) conducted an experiment to compare a Flow-based VPL (LabView) with a textual language (a self-developed textual equivalent to LabView). A total of 31 subjects participated in this experiment. The results indicated that the visual notations assisted in estimating execution sequence of program statements and in identifying errors in the code whereas the textual notations assisted in estimating the program output.

There was also research directed at which of the two notations was more comprehensible in terms of speed and accuracy. Cunniff and Taylor (1987) investigated the comparison between graphical representation (First Programming Language (FPL)) and textual representation (Pascal). FPL has a Flow-based programming paradigm. It consists of icons that can be connected to each other to form a program with each icon representing a specific programming action. This investigation was based on how fast and accurately the subject responded to the comprehension questions. It was observed from the findings that the graphical representation was better than the textual one in both - speed and accuracy.

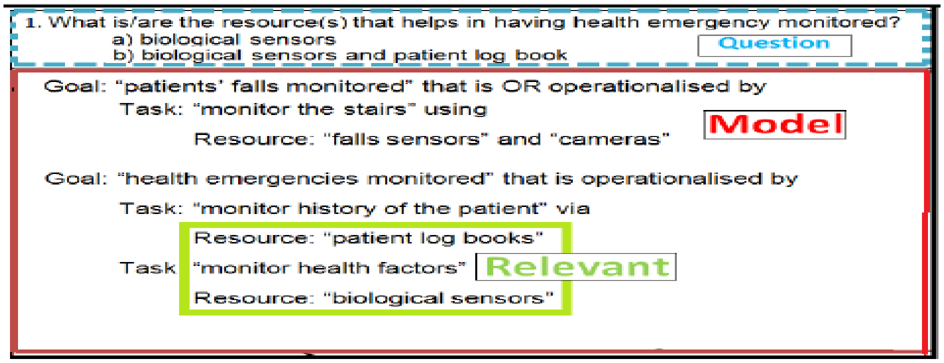
The empirical study conducted by Sharafi, Marchetto, Susi, Antoniol, and Guéhéneuc (2013) consisted of an eye-tracking experiment to analyze which of the two representations was more efficient - Flow-based graphical representation or the Text-based representation. A total of 28 subjects were a part of it, 12 female subjects and 16 male subjects. The environment chosen for this study was TROPOS based on its flexibility of including both graphical and textual notations. TROPOS uses the approach of 'actor' and 'goal' diagrams to visualize requirements. An actor diagram is like a graph consisting of two factors: nodes and vertices, nodes constituting actors and vertices indicating the dependencies between them. Eye-tracking system used here provides details of the subject's eye movements to understand their ongoing cognitive process. Graphical and textual representation of this environment is shown in Figure 2.20 and Figure 2.21 respectively.

The results from this experiment reported that the subjects consumed more time with TROPOS graphical representation as compared to the textual one. Though they preferred the graphical format more, they still performed better in comprehension tasks in the textual format. There were no significant differences between the two formats while assessing accuracy. The author implies on giving a training on graphical representation to the subjects in order for them to comprehend the benefits. Also, it may be harder for a non-native English speaker to understand the kind of graphical representation used in this study.



1. What is/are the resource(s) that helps in having health emergency monitored?
a) biological sensors
b) biological sensors and patient log book

Figure 2.20: Graphical Representation of TROPOS (Sharafi, Marchetto, Susi, Antoniol, & Guéhéneuc, 2013).


1. What is/are the resource(s) that helps in having health emergency monitored?
a) biological sensors
b) biological sensors and patient log book

Goal: "patients' falls monitored" that is OR operationalised by
Task: "monitor the stairs" using
Resource: "falls sensors" and "cameras"

Goal: "health emergencies monitored" that is operationalised by
Task: "monitor history of the patient" via
Resource: "patient log books"
Task "monitor health factors"
Resource: "biological sensors"

Figure 2.21: Textual Representation of TROPOS (Sharafi, Marchetto, Susi, Antoniol, & Guéhéneuc, 2013).

Stein and Hanenberg (2011) address the question of which of the two representations (Flow-based or Text-based) is more comprehensible. 35 students took part in this experiment. The results generated from the conducted experiment in the paper showed varying results with each of the two representations being good at certain tasks. The Flow-based notation proved to be better at helping students in identifying objects involved in more than two methods whereas the Text-based notation helped students in identifying methods that involved more than one dependent object.

2.4.2 Comparison between Text-based and Block-based languages

This section will compare Block-based programming with Text-based programming according to previous research studies, which were trying to establish which of the two representations have been more useful for users in terms of usability, ease of learning, helped better in understanding programming concepts and the programming practices engendered while using the language. Most of the work has been directed for students who need to be better prepared in atleast one programming language before they go to a university or a college where programming gets tougher. There has been less work targeted for adults. Since Text-based languages are generally on the difficult side for the learners to start with, most of the comparisons conducted previously introduced Block-based languages first to the learners. There have also been hybrid environments developed which incorporates the features of both Block and Text-based languages that help in an easy transition from Block to Text. This section is divided into two parts - first part talks about the research studies that compare Block and Text languages, and the second part portrays some of the hybrid environments built.

1. Comparison of Block and Textual Languages

   The study conducted by Weintrop (2015b) took place at an urban public high school. It was an 'Introduction to Programming' class and had a timeline of 10 weeks. Snap! ("Snap! (Build Your Own Blocks) 4.0," 2011) and Java were the two programming languages used for this study. The former was introduced in the first five weeks and the latter was introduced in the next five weeks. Similar to Scratch, Snap! is also a Block-based programming environment, which is implemented in JavaScript. There were 90 students in total across three sections which took this study including 67 male students and 23 female students (Weintrop, 2015a; Weintrop & Wilensky, 2015a, 2015b; Weintrop, 2016).

   The results consisted of 92% of students feeling Block-based programming to be easier in comparison to Text-based programming. This was supported by the views from students which included the ease of use of the drag-and-drop feature of the Block-based programming, easy to read labels on blocks, utility of the organization and ease of browsing the blocks (blocks divided into categories) and finally, a lack of the need to memorize syntax. Also, the shape and visual layout of the blocks served as cues to help realize where the blocks could be used while also comprehending the sequence of commands concept. Students pointed out disadvantages to the Block-based as well, as not being scalable enough to build large programs and issues of authenticity. Authenticity would mean to determine the closeness between 'programming tools and practices' and 'non-educational programming' situations. Even though it has pedagogical advantages, it could be harmful for older learners who would want to develop skills beyond academics, probably in a job or somewhere else. Weintrop and Wilensky (2015b) mentioned in their paper “There is a potential drawback to the programming-primitives-as-puzzle-pieces metaphor stemming from the fact that in a puzzle, each piece has one specific place that it belongs”. The students almost concluded that every block has a particular place that it will fit in, which opposes the fact of a general programming concept of commands allowing a great amount of diversity in the ways that it can be connected and the places it can be used. Students also pointed out another drawback of the Block-based programming language as being less powerful. The lack of the existence of a block for every operation renders it less workable than a Text-based language. Another problem encountered was the creation of longer programs with blocks compared to a textual code which was difficult to manage according to the participants.

4% of students felt Text-based was easier and the remaining 4% with an opinion that both the environments were similar to use.

Students performed well on questions related to conditional logic, procedures, function call and iterative logic in Block-based programming in comparison to Text-based programming. They also did better on the former in the concept of variables, though by a small margin with no difference in results on comprehension questions. In terms of the variables, the performance was better in Block-based where the variable values could be reset after being set. But when it came to a simple variable assignment, they surprisingly did well on the Text-based one. The results indicate that the Block-based environment did help students to understand the working of a construct and the output it would produce, but it does not necessarily simplify where and how to use that construct (Weintrop & Wilensky, 2015a, 2015b).

Armoni, Meerbaum-Salant, and Ben-Ari (2015) discussed how easy or hard the learning of computer science (CS) concepts would be, after using the Scratch environment. Users in this study were middle school students who decided to choose CS in secondary schools based on their experience with Scratch. In Scratch as we have seen before, one can program by dragging and dropping blocks. These blocks depict program components, such as expressions, conditions, statements, and variables. The secondary school class was divided into students experienced with Scratch, and students who had no programming knowledge. The students who had learned Scratch had an idea of some of the concepts such as variables, conditional statements, repeated execution etc. Hence, their understanding was faster in easy or a difficult concept as opposed to the students without any programming knowledge.

The disadvantage was that some of these similar concepts were recognizable to them only in the way that they had come across them in Scratch. Nevertheless, their learning was still faster and shorter amount of time was dedicated to learning a concept. An observation was that students experienced with Scratch were more enthusiastic and motivated in solving programming exercises. Finally, there were tests taken at the end in order to compare the learning between both the groups. The results indicated that there were no significant differences between both the groups - those who studied Scratch and those who did not, in most of the concepts except the "Repeated Execution" one, which is supposed to be one of the tougher concepts to understand in programming.

Another comparison was made using 5th grade students as part of a computing camp over the course of 6 days. There were two groups - one using Scratch and the other using Logo, a Text-based language. The course mainly concentrated on teaching media-rich lessons of how to make music, movies or games using computers. The author's hypothesis stated that Scratch being a Block-based language would make learning programming easier due to its lack of syntax errors. However, this did not turn out to be so true as students found problem questions equally difficult in both the languages. It seemed like both the languages were better suited to teaching specific constructs. Students learning Logo had a better comprehension of loops and those learning Scratch had a better comprehension of conditionals.

Booth and Stumpf (2013) took a small sample size of adults to perform their study. Participant exercises included them to create programs for Arduino for 20 minutes. A comparison was made between a Block-based editor known as Modkit and a Java-based textual editor. The results of this study indicated that the former was perceived to be more usable than the latter. In the exercise where participants were supposed to modify a program rather than implementing a new program, Modkit had better completion rates. In general, Modkit was more user friendly, less work load and a higher observed success. No statistical analysis was performed by the author and conclusions were also supported with quotations from participants.

Predictive modeling was used to compare a variety of Text-based and Block-based representations of programming. These included environments like Alice, Scratch, Greenfoot, Python and LEGO Mindstorms NXT. Booth and Stumpf (2013) modeled the time that was needed to execute the different programming tasks in each of the environment. This modeling was done using a prototyping tool called CogTool. The takeaway from the results was that some tasks, such as insertion and replacement were better suited by textual languages, while other tasks like deletion and movement were better suited by block languages. Even though the Block-based languages look similar in terms of their user interface (drag-and-drop), they could be diversified in terms of many aspects such as interactions, features etc. One of the differences identified by the authors was in terms of handling instantiating literals (text and numeric). Depending on how they are used in a particular Block environment, task time increases significantly. They also specify various drawbacks of their model that does not account for the time utilized by users in thinking or designing a program, and that some languages facilitate this better than others (McKay & Kölling, 2013).

Wagner, Gray, Corley, and Wolber (2013) investigated the importance of teaching a Block-based language (MIT App Inventor discussed in section 2.1.4) before transitioning to an actual Text-based language (Java Bridge, a Java implementation of the App Inventor API). This study took place at a K-12 summer camp in the span of five days, first two days with App Inventor, next two days with Java and the last day on student project. The teaching of Visual programming language followed by Java turned out to be a fruitful approach according to the authors. This not only gave students the confidence to construct their own applications but it also helped in fortifying programming concepts such as methods, objects and decision statements.

2. Hybrid Environments (Block and Text):

   The hybrid environments include both Block and Text elements. The reason why a hybrid environment came into existence is because neither Text-based nor Block-based programming environments fit all demographics for an intuitive and an easy programming experience.

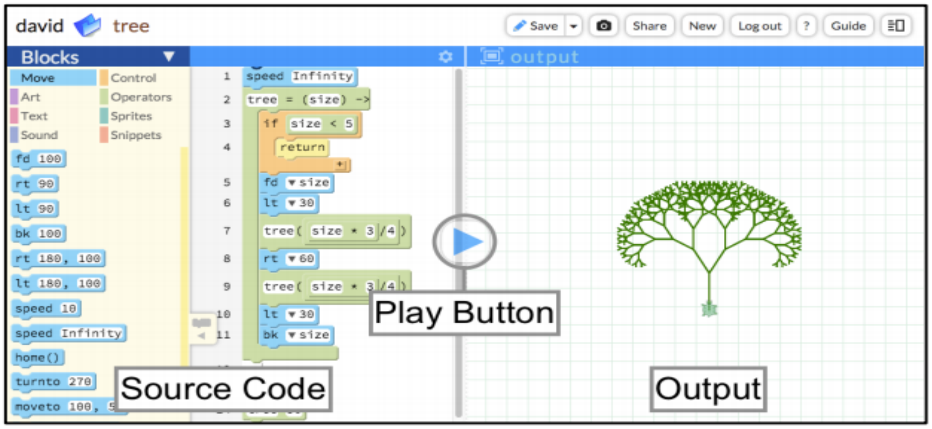
   In the case of paper by Armoni et al. (2015), the results indicated a performance improvement of the students who had learnt Scratch earlier, with additional advantages of increased motivation and higher learning speed of the Text-based version. Whereas, another paper by Powers et al. (2007) showed motivation reduction due to Block-based environments often perceived as not "real" programming or having a toy-like appearance.

   Weintrop (2015b) suggests that even though a Block-based programming environment with few instructions or none (Maloney et al., 2008; Malan & Leitner, 2007) has been successful in teaching programming concepts to learners, there was still a certain amount of struggle found in the changeover from a graphical environment to a Text-based language. There have been environments built which blend both styles of programming - Block-based and Text-based to minimize the issues associated with both programming techniques.

   Dann, Cosgrove, Slater, Culyba, and Cooper (2012) conducted a study in an introductory undergraduate Computer Science course where there was a transition from Block-based interface of Alice to a Java implementation of Alice. Java code was used in the test given to the students at the end of the course. To estimate the results, the authors compared students' scores with the ones they had received on a previous all-Java version of the course. The results included Alice classes to perform atleast one grade higher than the previous all-Java classes on each section of the test.

   Another programming environment called Pencil Code was developed for the education domain as well (Bau & Bau, 2014). It is a dual mode editor with Block and Text modes.

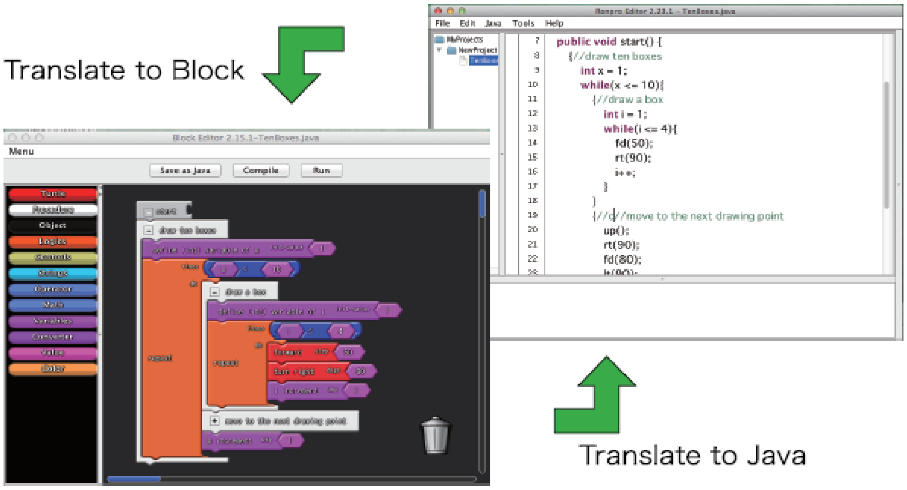
Figure 2.22 shows the Pencil Code interface.



```
speed Infinity
tree = (size) ->
  if size < 5
    return
  fd size
  lt 30
  tree( size * 3/4)
  rt 60
  tree( size * 3/4)
  lt 30
  bk size
```

Figure 2.22: Pencil Code Interface (Bau & Bau, 2014).

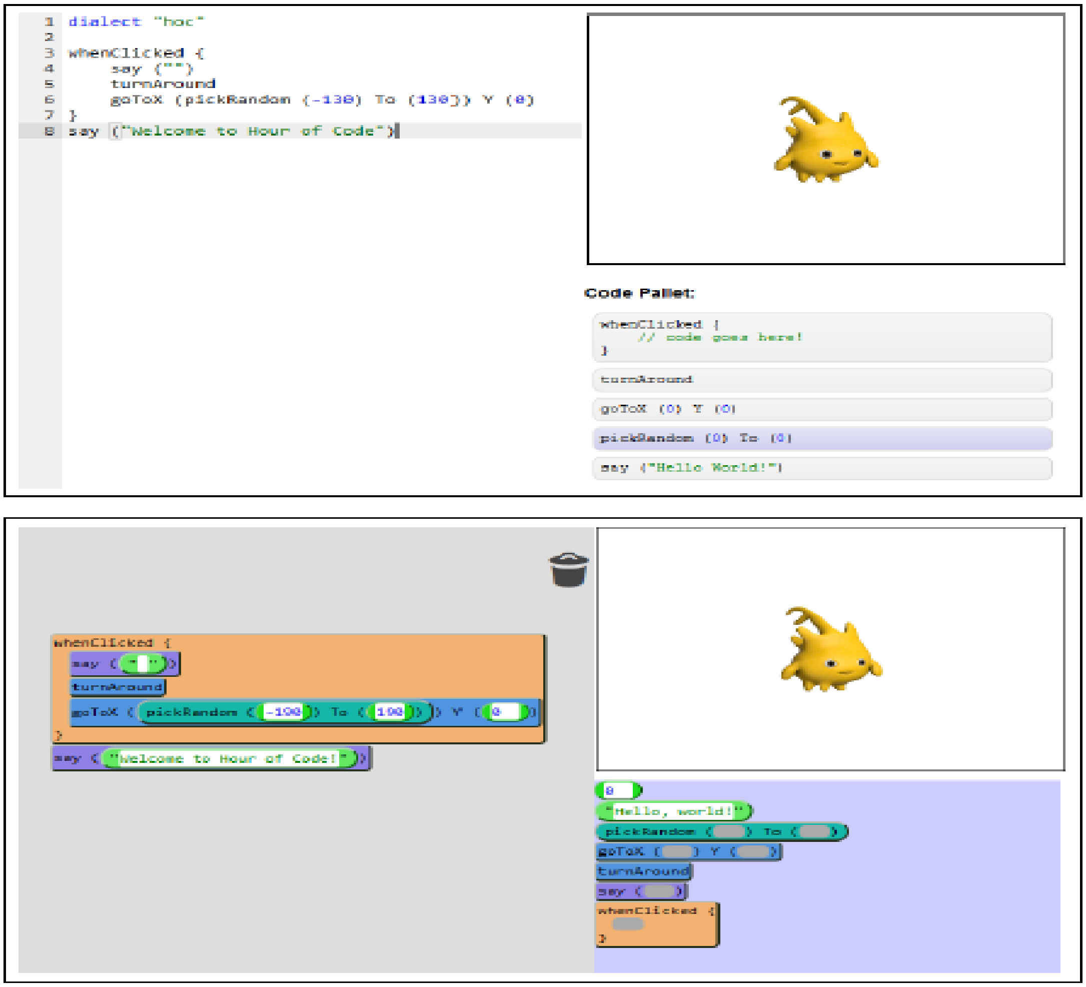
BlockEditor is a system that can be interchanged bidirectionally between Block and Text based on OpenBlocks framework developed at MIT. The textual language used here is Java. A study to ease the migration between Block and Text was conducted targeting 100 university students. The BlockEditor interface can be seen in Figure 2.23. Users can drag and drop blocks in the center pane to construct a program, the textual version of which can be obtained using the "save as Java" option. Since the environment is bidirectional, the Java code can be edited, saved and converted back into a Block program (Matsuzawa, Ohata, Sugiura, & Sakai, 2015).



```
public void start() {
  {//draw ten boxes
    int x = 1;
    while(x <= 10){
      {//draw a box
        int i = 1;
        while(i <= 4){
          fd(50);
          rt(90);
          i++;
        }
      }
      {//c//move to the next drawing point
        up();
        rt(90);
        fd(80);
```

Figure 2.23: BlockEditor (Matsuzawa, Ohata, Sugiura, & Sakai, 2015).

Price and Barnes (2015) chose Tiled Grace programming environment (M Homer, 2014) for their study which occurred as a part of a middle school STEM outreach program called SPARCS (Cateté, Wassell, & Barnes, 2014). It is a web-based environment that consists of both block and textual programming interfaces (Figure 2.24).



```
dialect "hoc"

whenClicked {
      say ("")
      turnAround
      goToX (pickRandom (-130) To (130)) Y (0)
}
say ("Welcome to Hour of Code")
```

Code Pallet:

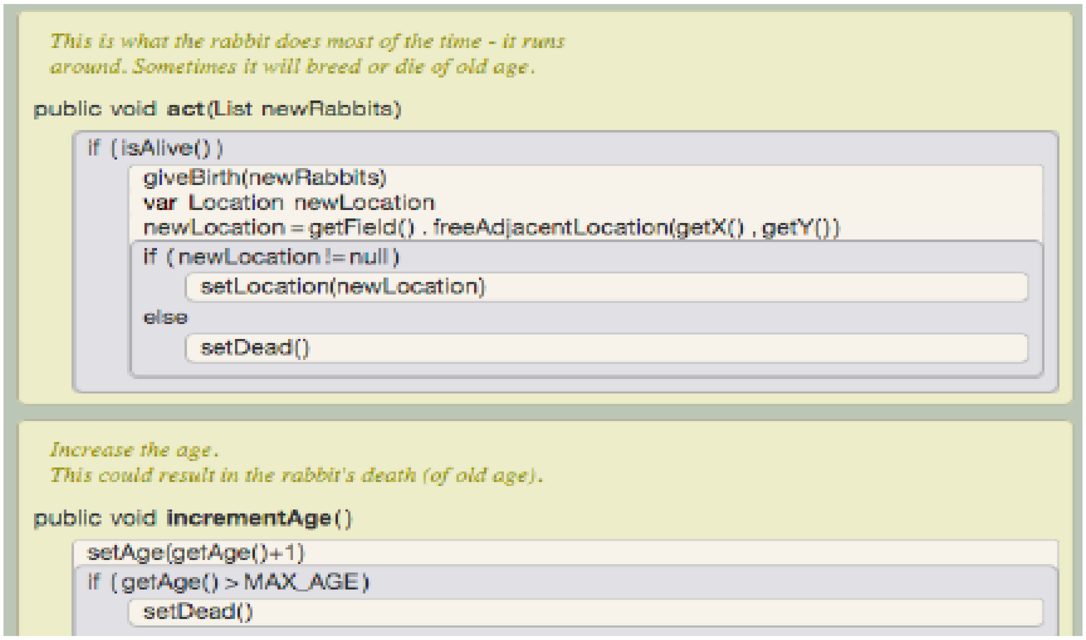
```
whenClicked {
    // code goes here!
}
turnAround
goToX (0) Y (0)
pickRandom (0) To (0)
say ("Hello World!")
```

```
whenClicked {
  say ("")
  turnAround
  goToX (pickRandom (-100) To (100)) Y (0)
}
say ("Welcome to Hour of Code!")
```

Figure 2.24: Text-based interface (top) and Block-based interface (bottom) (Price & Barnes, 2015).

The user can interchange between the two modes. There were two groups in this study - sixth and seventh graders who were both randomly chosen for the experiment. The analysis of the amount of time taken using the interface showed that the idle time of a Block-based version was less than the idle time for the textual one. An analysis of the completion rate showed the Block group performing better than the Text group in completing a particular goal of the given activities. The authors conclude by claiming Block-based programming can help novices enhance their programming performance.

Another research concentrated on a hybrid frame-based editor, that would provide an easy transition from Block-based to a Text-based version (M Kölling, 2015). Figure 2.25 below provides an example program in Stride (Java-like language), in a frame-based editor.



```
This is what the rabbit does most of the time - it runs
around. Sometimes it will breed or die of old age.
public void act(List newRabbits)
    if (isAlive())
        giveBirth(newRabbits)
        var Location newLocation
        newLocation = getField() . freeAdjacentLocation(getX() , getY())
        if (newLocation != null)
            setLocation(newLocation)
        else
            setDead()

Increase the age.
This could result in the rabbit's death (of old age).
public void incrementAge()
    setAge(getAge()+1)
    if (getAge() > MAX_AGE)
        setDead()
```

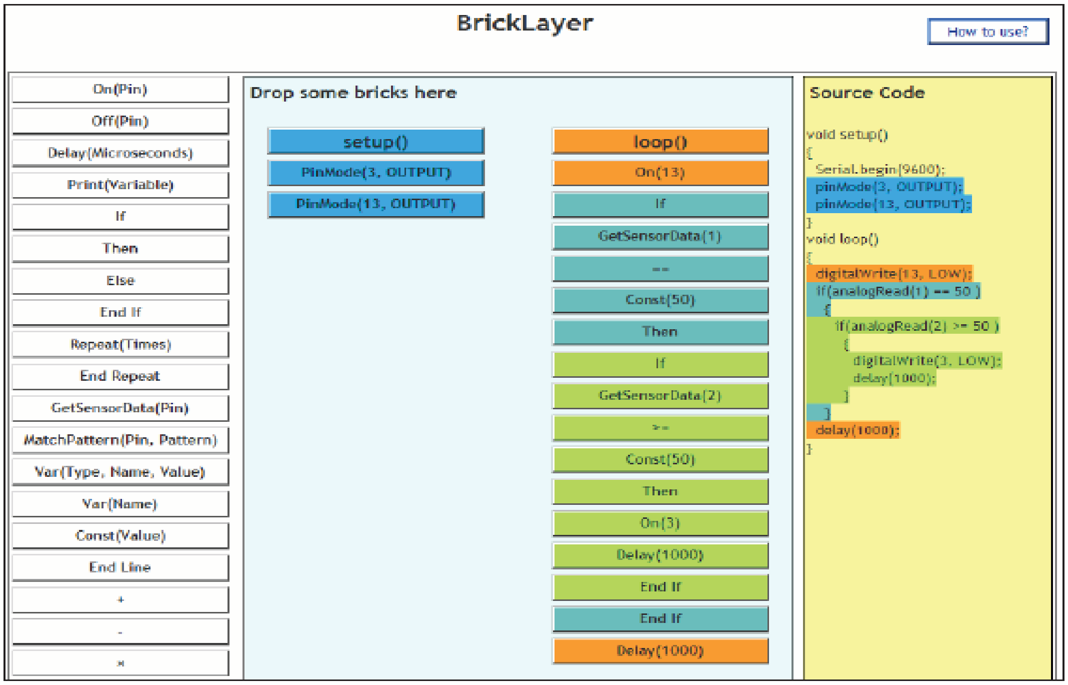
Figure 2.25: Program representation in a frame-based editor (M Kölling, 2015).

An advantage of frame-based editor is the usage of slots. There are two types of slots - frame slots and text slots. The former has a structure similar to a Block-based language (seen at the statement level). The latter has a structure that works like a Text-based language (seen at the expression level). This saves the user from dragging in multiple blocks at the block level to form a multi-operator expression for example and permits him to write the code in text to do the same, yet preventing the user from making any parenthesis error.

The authors encountered some issues such as readability, number of method calls (increased due to many methods being available from the Java library), understanding types, two types of transitions involved (transition from block to frame, and frame to text) and one of the most important problem is the restricted availability of frame editors (M Kölling, 2015).

There was also another language developed called Patch which incorporates the features of both Scratch and Python programming language. Patch was designed to be used by students before and during their middle school years (Robinson, 2016).

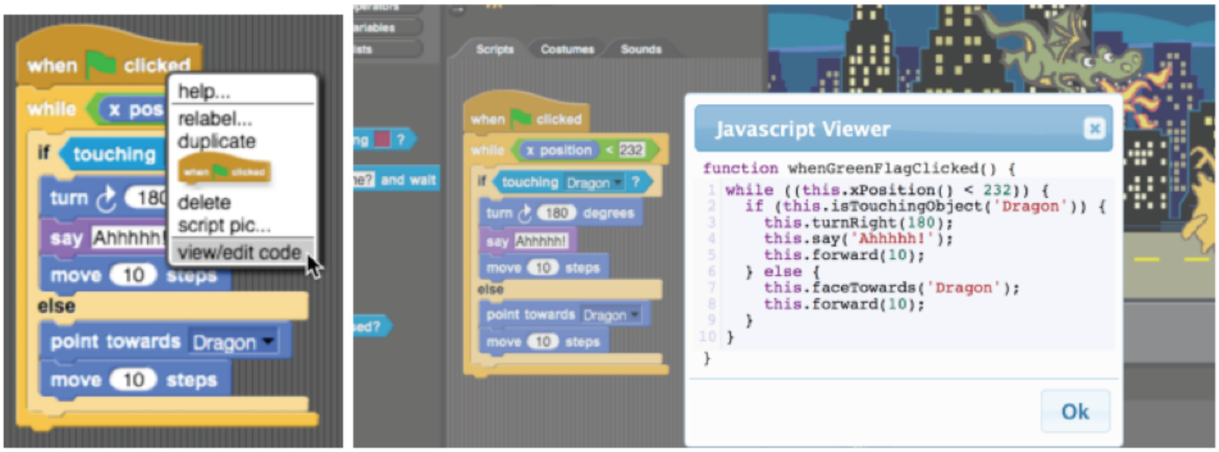
Another environment developed by Cheung, Ngai, Chan, and Lau (2009) called BrickLayer shown in Figure 2.26 is a text-enhanced graphical programming environment designed for junior high school students. Students would have to drag and drop blocks to create a program, while a text code is immediately produced and shows on an on-screen code area helping them to pick up the Text-based programming syntaxes though experimentation and observation.



```
void setup()
{
  Serial.begin(9600);
  pinMode(3, OUTPUT);
  pinMode(13, OUTPUT);
}
void loop()
{
  digitalWrite(13, LOW);
  if(analogRead(1) == 50 )
   {
     if(analogRead(2) >= 50 )
      {
        digitalWrite(3, LOW);
        delay(1000);
      }
   }
  delay(1000);
}
```

Figure 2.26: BrickLayer Interface Cheung, Ngai, Chan, and Lau (2009).

Finally, Snap! ("Snap! (Build Your Own Blocks) 4.0," 2011) is also a hybrid programming environment implemented in JavaScript as shown in Figure 2.27, with a few different functionalities as compared to the other environments described above.



```
function whenGreenFlagClicked() {
while ((this.xPosition() < 232)) {
  if (this.isTouchingObject('Dragon')) {
    this.turnRight(180);
    this.say('Ahhhhh!');
    this.forward(10);
  } else {
    this.faceTowards('Dragon');
    this.forward(10);
  }
}
}
```

Figure 2.27: Hybrid version of Snap with Blocks and Text side by side ("Snap! (Build Your Own Blocks) 4.0," 2011).

2.4.3 Comparison between Flow-based and Block-based languages

There is currently no previous work, which compares Flow-based and Block-based programming environments. Therefore, the purpose of this research is to analyze and compare the Flow-based and Block-based methodologies to determine the usability of each with respect to the conditions described in the next chapter.

Chapter 3

Experimental Design

3.1 Description of the experiment

There is enough evidence in the 'Related Work Chapter' about the benefits of Visual Programming Languages (Block and Flow) over traditional Text-based languages. However, we do not have any knowledge of a comparison between the two visual programming paradigms - Flow and Block. Our working hypothesis is that Flow-based programming and Block-based programming are equivalent in terms of performance for individuals with little or no programming background.

In order to prove this hypothesis, we conducted an experiment to compare Flow-based and Block-based representations to determine which is more usable for naïve programmers or individuals with basic programming skills. This comparison is conducted in terms of various factors such as speed, accuracy, ease of use, easy notations on the blocks, and other factors that are discussed in more detail in this chapter.

3.2 Detailed Description of Flow-based and Block-based Environments

The different categories and different types of blocks are explained in this section for each of the two environments.

3.2.1 Similarities of Flow and Block Environments

To make a useful comparison, Flow and Block environments are developed as similarly as possible.

Both the environments are built using Pharo and PharoJS. Pharo is an Integrated Development Environment (IDE) which is an implementation of the Smalltalk programming language and PharoJS is an open source framework that allows Pharo applications to run on a JavaScript interpreter in a browser.

The user interface for both the environments is designed to pop-up a bubble menu with a mouse click, which is the initiation point for any user. The bubble menus for the Flow-based and Block-based environment are displayed in Figure 3.1 and Figure 3.2, respectively. A sub-menu is displayed on choosing any of the categories displayed in the bubble menu except for the ‘Done’ category.

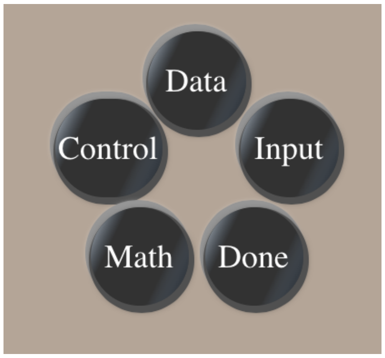


Figure 3.1: Bubble Menu Flow Environment.

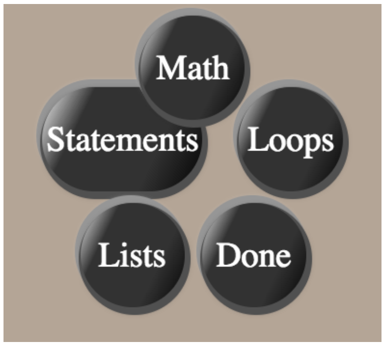


Figure 3.2: Bubble Menu Block Environment

This bubble menu is divided into various categories, out of which 'Math' and 'Done' are the two common categories, and they are identical in the two environments.

MATH

The different kinds of blocks in this menu can be seen in Figure 3.3.

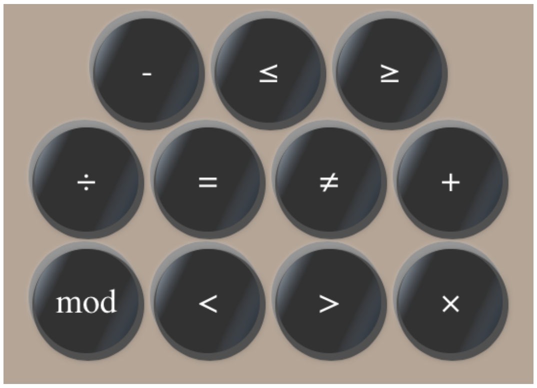


Figure 3.3: Math Menu.

DONE

Users click on the 'Done' sub-menu at the end, when they have finished solving the problem question and have generated program code as an output. On clicking Done, the user is directed to a feedback form.

3.2.2 Flow-based environment

The Flow-based environment is illustrated in the Figure 3.4.

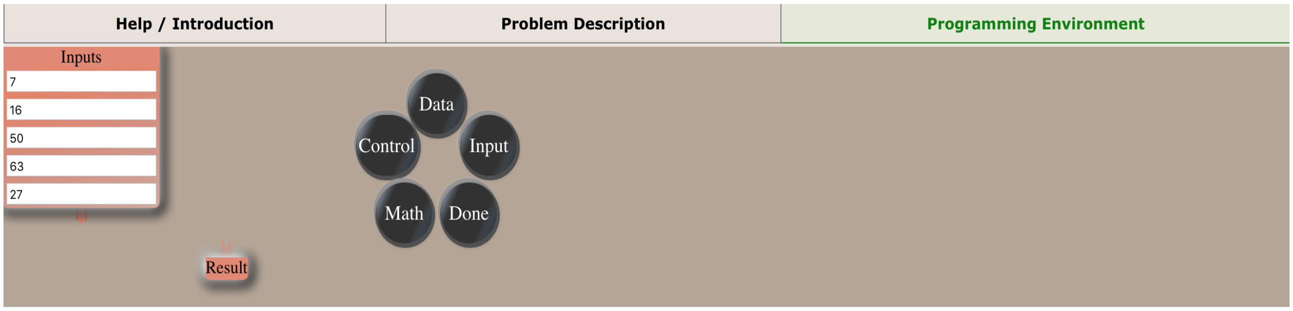


Figure 3.4: Flow-based environment

Flow-based environment uses pre-defined blocks that are displayed within the bubble menu, providing options for the user to select from depending on their requirement. Apart from Math and Done categories discussed above, the Flow-based bubble menu consists of the following categories: Input, Data, and Control. The user can build a program using the different blocks in each category. Each of these blocks have a 2D rectangular shape that consists of inTabs and outTabs to receive inputs and produce outputs, respectively (Mason, 2017).

Input Category

There are two types of blocks in the Input category: Simple Value Block and Input Table Block (Mason, 2017). These blocks with values in it were given in the environment according to the problem question requirements. These blocks can be seen in the Figure 3.5 and Figure 3.6 respectively.

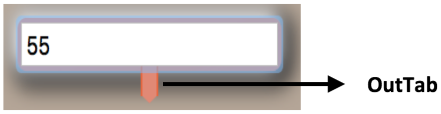
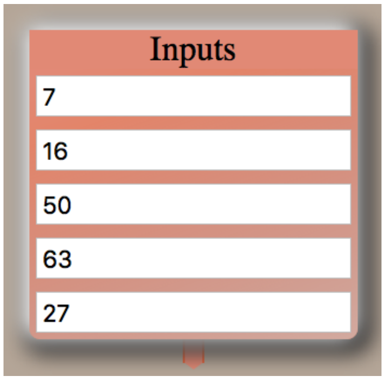


Figure 3.5: Simple Value Block



7
16
50
63
27

Figure 3.6: Input Table Block

Input blocks have outTabs only. Output values that come out of an executed Input block are transferred to another block through the ‘outTabs’. Hence, outTabs are always found at the bottom of an Input block shown in Figure 3.5.

Output Category

There are two types of blocks in the Output category: Result and Result Table Block (Mason, 2017). An empty ‘Result Block’ or ‘Result Table Block’ is given in the environment according to the problem question requirements and is shown in Figure 3.7.

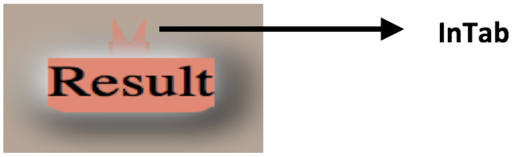


Figure 3.7: Result Table Block

Output blocks have inTabs only. Input values come into the recipient Output block through the ‘inTabs’. Hence, inTabs are always found at the top of an Output block shown in Figure 3.7.

Each block in the sub-categories Data and Control have a pre-defined function associated to it. Hence, the data for any block is processed according to the block's function. A block is executed when all of its inputs are received, and the execution is complete when outputs are produced from the block in a sequential order. Users can dynamically change the input value at any given time according to their needs, and the associated blocks will change the output with respect to the new value.

Data Category

This category consists of five blocks: Total, Count, Copy, Merge, and Average as shown in the Figure 3.8.

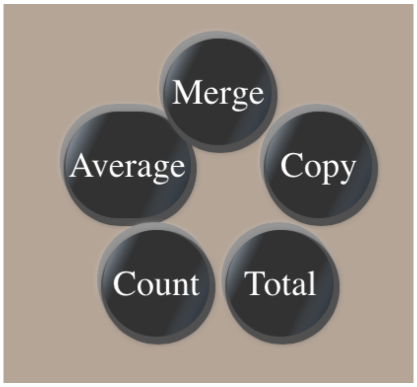


Figure 3.8: Data Category.

Control Category

This category consists of four blocks: When, Unless, Filter, and WhenUnless as shown in the Figure 3.9.

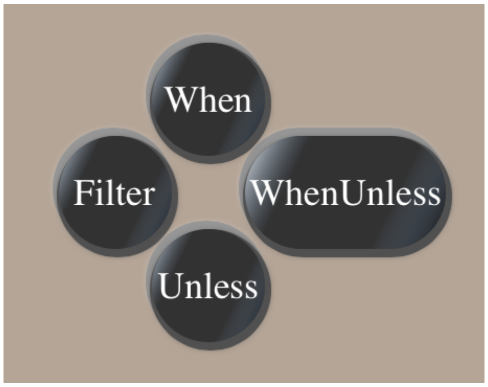


Figure 3.9: Control Category.

More details on the working of the blocks mentioned in the Data and Control categories above can be found in the ‘Flow-based Environment Help Documentation’ attached in appendix A.

Pipes

Flow-based environment contains pipes that perform the task of linking blocks together by transferring data from one block to another. Data flows through the pipe based on the sequence of the connected blocks. Users can generate a pipe by clicking on a block’s outTab and dragging it to the next block’s inTab.

An Example Problem

Figure 3.10 shows an illustration of how to multiply two numbers by means of Flow-based blocks and pipes. The figure displays two Simple Value Blocks, where the user can fill in any number manually. Both the blocks are connected to a multiply block with a pipe. The user can then view the output in a Result Block connected to the multiply block.

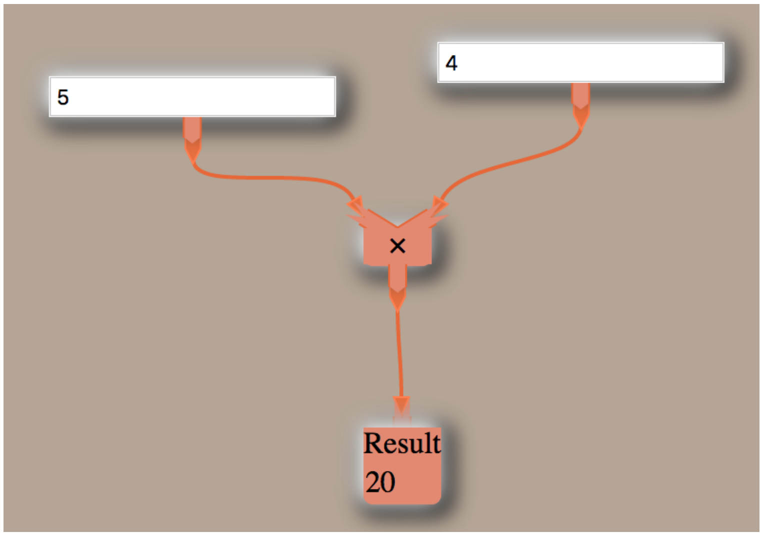


Figure 3.10: An example to multiply two numbers.

As shown in the example in the Figure 3.10, users could likely create a program with an understanding of the blocks and data flow, without prior programming knowledge.

3.2.3 Block-based environment

While developing the Block-based environment, I divided its user interface into two sections as shown in the Figure 3.11: 'Program Area' and 'Display Area'.

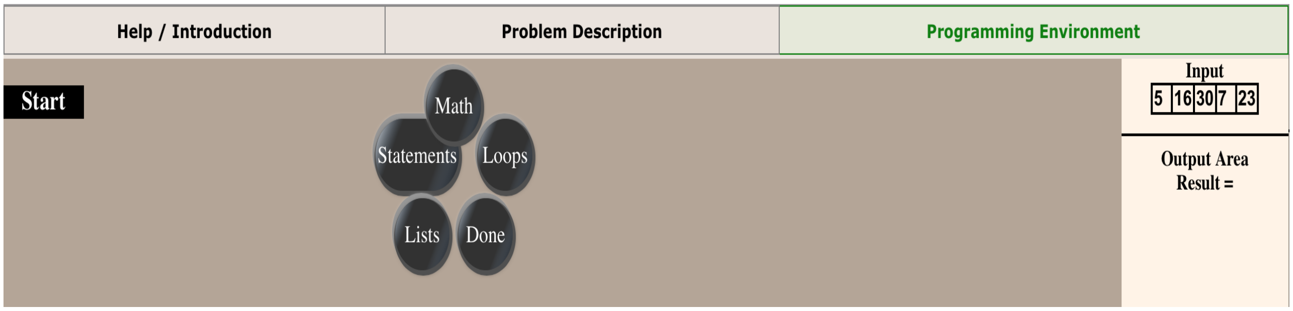


Figure 3.11: Block-based environment

Program Area: This area produces a 'bubble menu' on clicking anywhere on the page. Users can select appropriate blocks from the different categories displayed within the bubble menu and organize the blocks in this area. They can then drag and drop their created program below the ‘Start’ button to produce the desired output. The user can either create their program directly below the ‘Start’ button or they can create it outside and then drop the entire program below the ‘Start’ button. Users have the flexibility to code in the style that they find comfortable.

Display Area: The output of the program created in the program area is shown in the display area.

Apart from Math and Done categories discussed above, the Block-based bubble menu consists of the following categories: Statements, Loops, and Lists.

STATEMENTS

Figure 3.12 illustrates the Statements Menu. This category consists of two kinds of blocks: An Assignment Block and Conditional Statement Blocks.

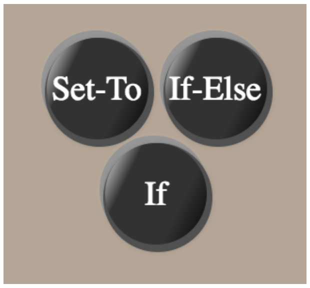


Figure 3.12: Statements Menu.

Assignment Block has one ‘Set-To’ Block shown in Figure 3.13. It sets a variable (‘x’ in this case) to a particular required value. Variables can be thought of as a location that holds distinct values at distinct times.

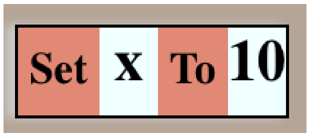


Figure 3.13: Set-To Block

There are two kinds of Conditional Statement Blocks: ‘If-Then’ and ‘If-Then-Else’. These blocks consist of a ‘condition area’ and a ‘body’. ‘If-Then’ block consists of one ‘THEN body’ and If-Then-Else’ block consists of ‘THEN body’ as well as an ‘ELSE body’. The code in their bodies execute according to the condition specified in their condition area. Two small programs demonstrating ‘If-Then’ and ‘If-Then-Else’ Blocks are shown in Figure 3.14 and Figure 3.15 respectively.

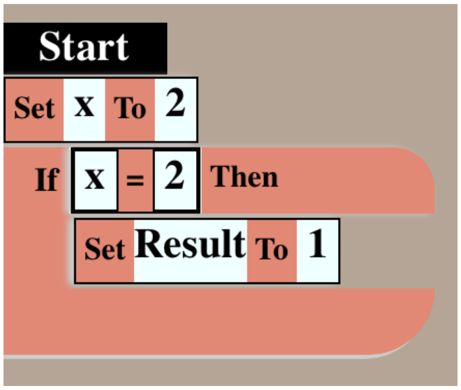
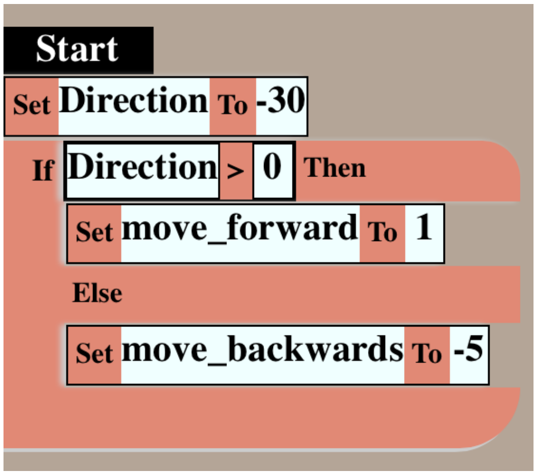


Figure 3.14: An example program using If-Then Block



```
Start
Set Direction To -30
If Direction > 0 Then
    Set move_forward To 1
Else
    Set move_backwards To -5
```

Figure 3.15: An example program using If-Then-Else Block

LOOPS

Figure 3.16 illustrates the Loops Menu showing two types of blocks namely: ‘Repeat-Until’ and ‘Repeat-While’.

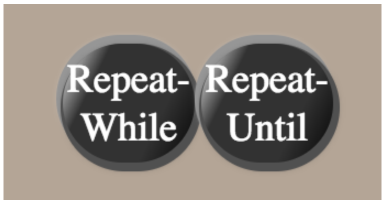


Figure 3.16: Loops Menu.

In computer programming, conditional loops are a way for computer programs to repeat one operation multiple number of times until a specific condition is fulfilled. In this case, the two types of blocks repeatedly execute themselves until the condition specified in their condition area is satisfied. Two small programs demonstrating ‘Repeat-Until’ and ‘Repeat-While’ Blocks can be seen in the Figure 3.17 and Figure 3.18.

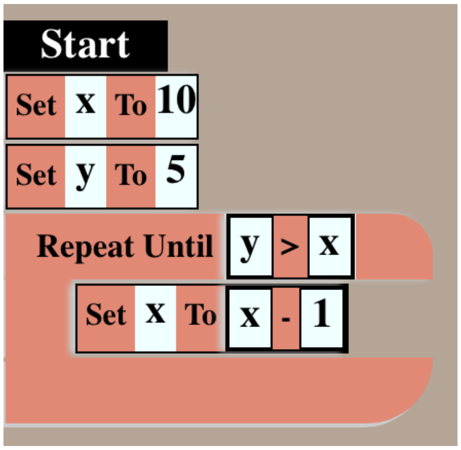


Figure 3.17: An example program using Repeat-Until Block.

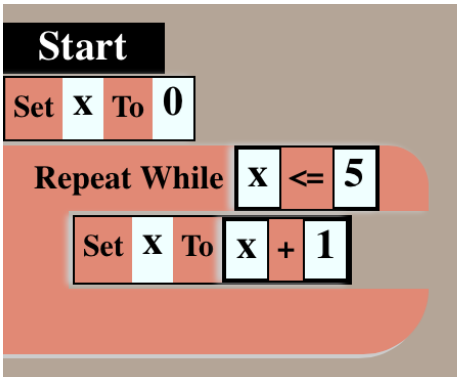


Figure 3.18: An example program using Repeat-While Block.

LISTS

In general, as well as in computer programming, Lists assist in collecting pieces of information in one place. At a single time, they can store multiple items. There are three types of List Blocks in this category: ‘Length of List’ Block, ‘Item Position of List’ Block, and ‘Add-to List’ Block as shown in the Figure 3.19. Two other types of List Blocks namely ‘Input List’ Block and ‘Result List’ block were already given in the environment according to the problem question requirements. Users did not have to create them.

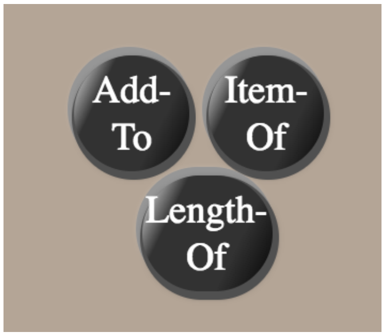
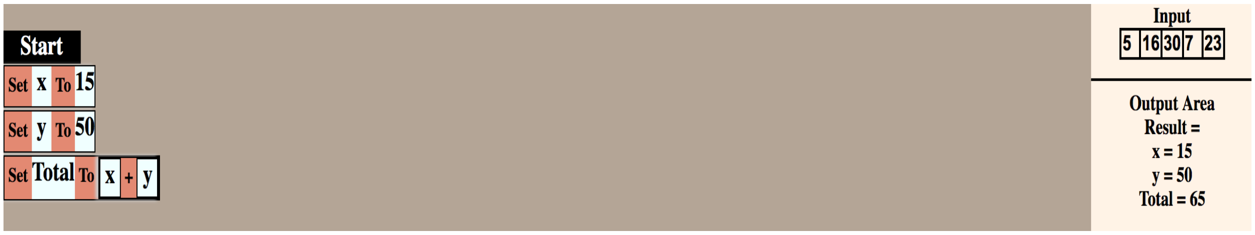


Figure 3.19: List Menu.

More details on the working of the blocks mentioned in the four categories above can be found in the ‘Block-based Environment Help Documentation’ attached in appendix B.

Example program: An example to add two numbers in this environment is shown in the Figure 3.20.



```
Start
Set x To 15
Set y To 50
Set Total To x + y
```

```
Input
5 16 30 7 23

Output Area
Result =
x = 15
y = 50
Total = 65
```

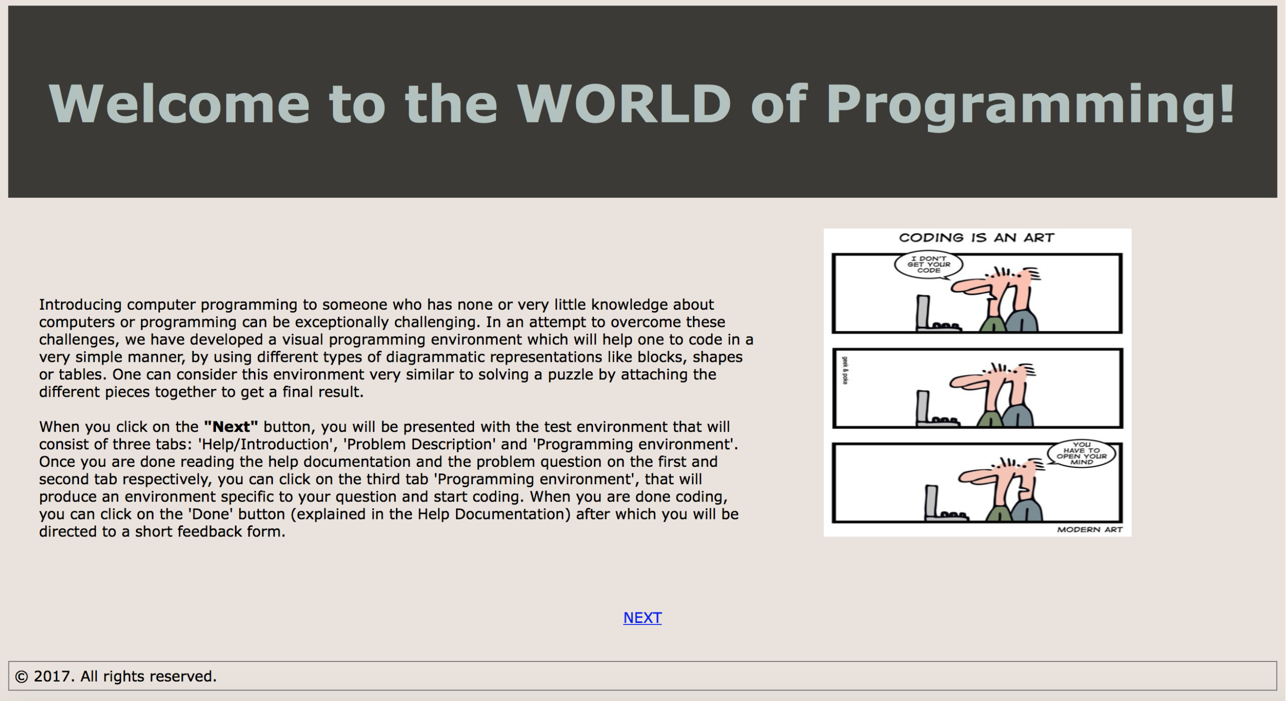
Figure 3.20: An example to add two numbers in the Block-based environment.

From the description of the Block-based programming environment above, it is evident that a user who has understood the Block-based paradigm and the working of the environment and its blocks can construct different programs, without any knowledge and background of programming languages.

3.3 Organization of the experiment

3.3.1 Test Environment

The first page of the test environment that is presented to the user is shown in Figure 3.21.



Welcome to the WORLD of Programming!

Introducing computer programming to someone who has none or very little knowledge about computers or programming can be exceptionally challenging. In an attempt to overcome these challenges, we have developed a visual programming environment which will help one to code in a very simple manner, by using different types of diagrammatic representations like blocks, shapes or tables. One can consider this environment very similar to solving a puzzle by attaching the different pieces together to get a final result.

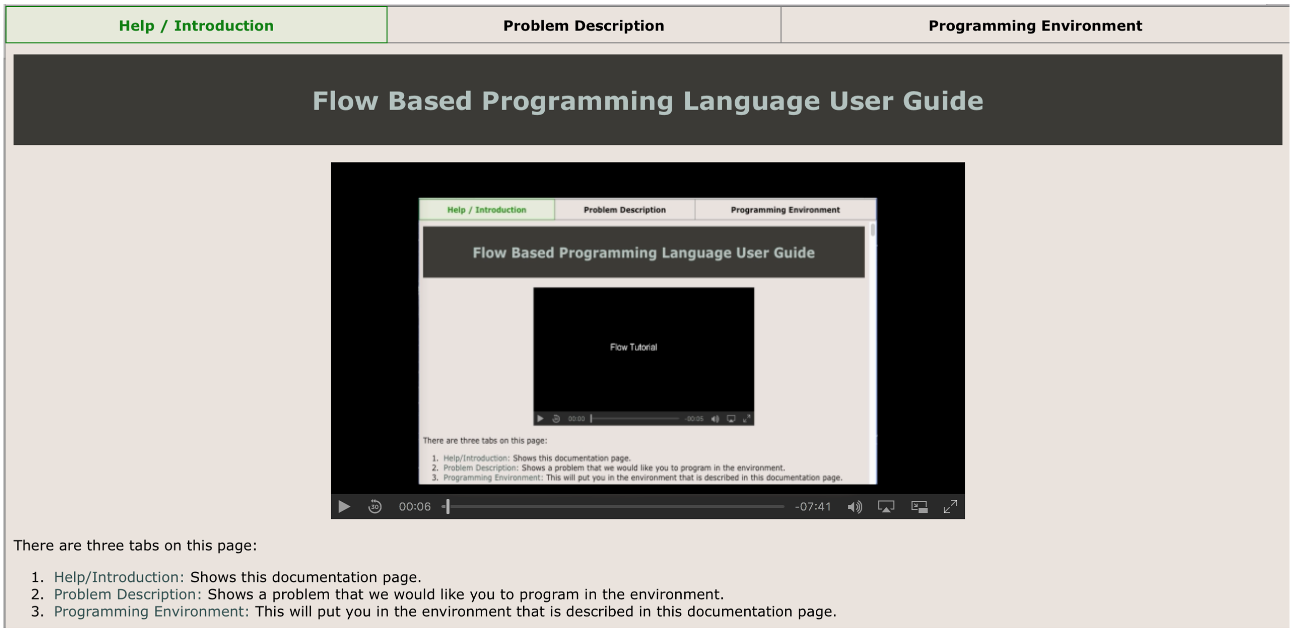
When you click on the **"Next"** button, you will be presented with the test environment that will consist of three tabs: 'Help/Introduction', 'Problem Description' and 'Programming environment'. Once you are done reading the help documentation and the problem question on the first and second tab respectively, you can click on the third tab 'Programming environment', that will produce an environment specific to your question and start coding. When you are done coding, you can click on the 'Done' button (explained in the Help Documentation) after which you will be directed to a short feedback form.

© 2017. All rights reserved.

Figure 3.21: Welcome Page.

The "Next" button shown in Figure 3.21 opens either Flow-based or Block-based environment. If the user loses the page either by closing it or in any other way, the environment they are working on will not be lost. Every time the user opens the environment page, the same one will be displayed consistently. This is handled by incorporating cookies in the webpage, that maintains the user's session information.

The Flow-based environment is illustrated in Figure 3.22.



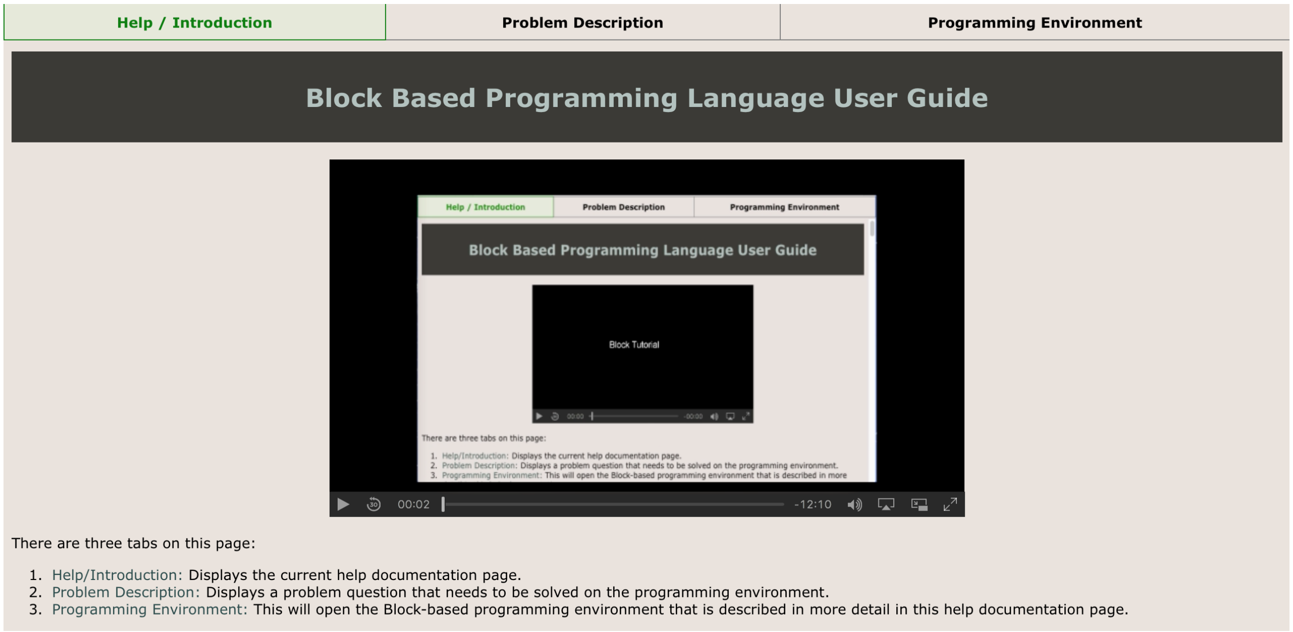
Flow Based Programming Language User Guide

There are three tabs on this page:

1. Help/Introduction: Shows this documentation page.
2. Problem Description: Shows a problem that we would like you to program in the environment.
3. Programming Environment: This will put you in the environment that is described in this documentation page.

Figure 3.22: Flow-based environment

The Block-based environment is illustrated in the Figure 3.23.



Block Based Programming Language User Guide

There are three tabs on this page:

1. Help/Introduction: Displays the current help documentation page.
2. Problem Description: Displays a problem question that needs to be solved on the programming environment.
3. Programming Environment: This will open the Block-based programming environment that is described in more detail in this help documentation page.

Figure 3.23: Block-based environment

The experiment runs on a webpage with three tabs: **Help/Introduction**, **Problem**

Description, and **Programming Environment** as seen in Figure 3.22 and Figure 3.23. At any point, a user can switch between these tabs.

The **Help/Introduction** tab presents the help documentation for that particular environment.

The **Problem Description** tab displays a problem question for that particular environment.

The **Programming Environment** tab is where the user is required to develop the code.

3.3.2 Problem Questions

We selected 6 problem questions for both the Flow-based and Block-based environments. The questions are formulated in a simple manner with each user provided a randomly selected question using one of the environments. There are 12 combinations possible in this experiment.

Every question has the result provided in its description. The question does not have the actual code or the method to produce the answer, but has hints if the question requires it. The user could refer to the hints to obtain formulas for certain questions such as calculating average, odd numbers and squares of numbers. These hints were given to mitigate the pressure on the user's thinking abilities in remembering formulas or concepts and put more effort and time on the programming process. There are minor differences in the way questions are stated for both the environments, such as Flow-based uses the term 'table' and Block-based uses the term 'list'. The descriptions below are for Block-based environment.

The six questions are described below:

1. Find the Odd Numbers in a List

   Given below is an Input list consisting of different numbers. Find all the odd numbers in the Input list and display them in the Result list. The following Input list will be provided in the environment. Perform the necessary calculations on the given data in the Input list and compute the answer.

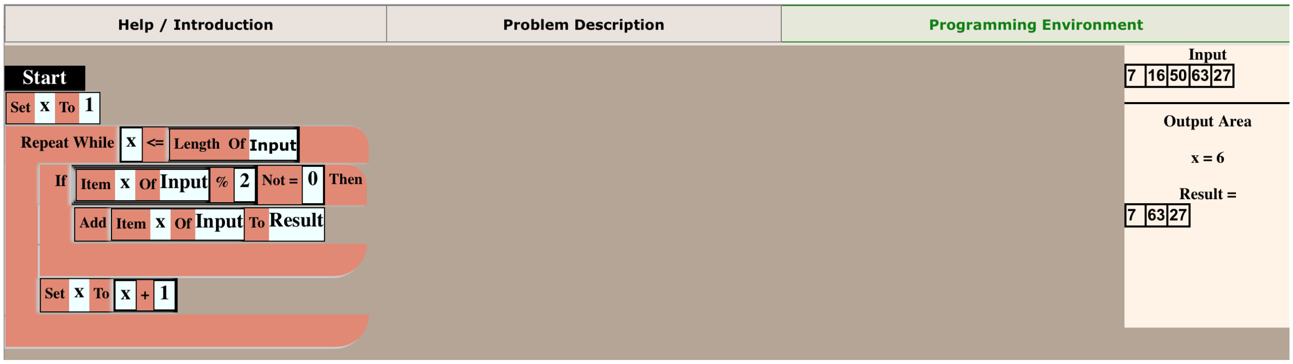
7	16	50	63	27	Input

   [Hint: The number is an odd number, if it leaves a remainder 1, when divided by 2. Use the mathematical operator - ‘mod’ to solve this question. Refer to the Help documentation to understand the working of ‘mod’ operator. Produce the answer (list of odd numbers) in another list].

   An empty Result list is created and provided in the environment as well. Add the answer (list of odd numbers) to the provided Result list. The answer to this question is:

7	63	27	Result

Figure 3.24 and Figure 3.25 shows a possible solution for this particular question in both Flow-based and Block-based environments:



```
Start
Set x To 1
Repeat While x <= Length Of Input
    If Item x Of Input % 2 Not = 0 Then
        Add Item x Of Input To Result
    Set x To x + 1
```

Input

7	16	50	63	27

Output Area

x = 6

Result =

7	63	27

Figure 3.24: Coded solution for The First Problem Question in Block-based environment

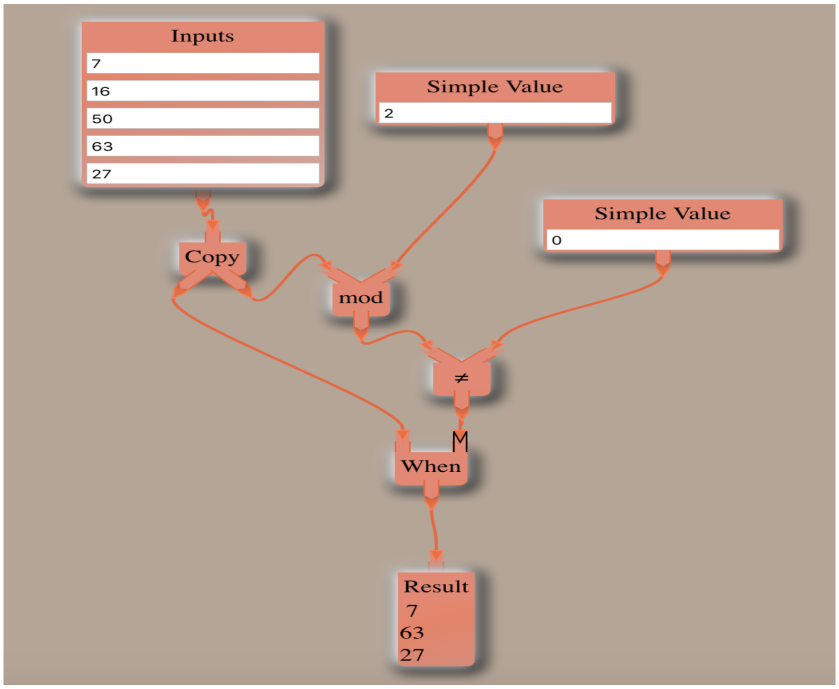


Figure 3.25: Coded solution for The First Problem Question in Flow-based environment

2. Square of numbers in a List

Gerald made deposits into 5 different banks. The values given in the Input list below are not the exact values that Gerald deposited. To find the exact values, calculate the square of each value in the Input list and display them in the Result list. The following Input list will be provided in the environment. Perform the necessary calculations on the given data in the Input list and compute the answer. [Hint: To calculate squares of numbers simply calculate the product: $4 \times 4 = 16$.]

5	16	30	7	23	Input

An empty Result list is created and provided in the environment as well. Add the answer (list of odd numbers) to the provided Result list. The answer to this question is:

25	256	900	49	529	Result

3. Calculate average of a list of numbers

You want to estimate your average daily household expense. So, you decide to note down the daily expenses for one week. This is how it looked:

50	70	30	55	76	81	46

Input

Calculate the average daily household expense using the above data. The above Input list will be provided in the environment. Perform the necessary calculations on the given data in the Input list and compute the answer.

[Hint: “Average” is the sum of all the values divided by the total number of those values. For example, the average of three numbers 5, 77, 18 is equal to $\frac{5+77+18}{3}$.]

Store the answer in the ‘Result’ variable, created and provided in the environment.

The answer to this question is:

Result = 58.28

4. Find the larger of two given numbers

Two numbers x = 455 and y = 811 are provided. Find which number is greater of the two and double the larger number. Perform the necessary calculations on the given data and compute the answer.

Store the answer in the ‘Result’ variable, created and provided in the environment.

The answer to this question is:

Result = 1622

5. Find Repetitions

A range of numbers is given in the Input list below, where the number ‘5’ is repeated many times. Count the number of times the number ‘5’ has been repeated. The Input list will be provided in the environment. Perform the necessary calculations on the given data in the Input list and compute the answer.

5	6	5	12	27	5	7	5	5	13	8	5	Input

Store the answer in the ‘Result’ variable, created and provided in the environment. The answer to this question is:

Result = 6

6. Find the greater of two lists corresponding to their positions

Two input lists are provided in the environment. Find the numbers in the first list ‘Input1’ that are greater than the numbers in the second list ‘Input2’ with respect to their positions. For example, the number at position 1 of Input1 list is greater than the number at position 1 of Input2 list. Hence this number should be added to the Result list. The same check should be done for all the other positions as well. Perform the necessary calculations on the given data in the Input1 and Input2 lists and compute the answer.

8	66	13	87	9	76	Input1

5	80	12	11	78	41	Input2

An empty Result list is created and provided in the environment as well. Add the answer to the provided Result list. The answer to this question is:

8	13	87	76	Result

3.3.3 Help Documentation

Help documentation can be used to understand the Flow and Block aspect of the environments and how to use them. This contains both video tutorials, that could help the user understand the basics of the environment faster and a detailed text, that gives them a detailed explanation of the environment and its different blocks. This documentation can be referred while executing the code. Refer to appendix A and appendix B for a review of the Flow and Block Help documentation provided to the users.

3.3.4 Test subjects/Participants

An invitation to participate was broadcast via emails and social media. Participants could be of different genders, ages, and levels of education. The goal behind having an extensive variety of participants is to observe distinctive methodologies of approaching problems. The different approaches and solutions produced by the participants are analyzed in section 4.2 and section 4.3.

3.3.5 Deployment of the Two Environments

The environments were deployed on a web browser and are compatible with three of the widely used browsers - Google Chrome, Safari and Mozilla Firefox. Our attempt is to help users have no issues using the browser of their choice. When the participants started their task, they were directed to a page that displayed a random choice of both the environment and the problem question. There is no restriction on the type, age or education of participants involved in this study.

3.3.6 Data Collection Methods

We collected data about the user's performance on the environments in two primary approaches. The initial approach consisted of adding three types of tracking features to the environments and in the secondary approach, users were provided with a feedback form with specific questions.

Tracking Features

Three types of Tracking features added to the environments were:

(a) User Mouse Click:

Every mouse click by the user is recorded:

- Number of clicks on the Help Documentation tab.
- Number of clicks on the Problem Question tab.
- Number of clicks on the Programming Environment tab.
- Time spent on Help Documentation.
- Time spent on Problem Question.
- Time spent on Programming Environment.

(b) Creation and Destruction of Nodes:

Every block created or destroyed is recorded for both the environments. For example, if the user pulled out an 'add' block from the 'Math' category, a message is recorded: 'created as addBlock'. If the user decides to delete the add Block, another message is recorded: 'removed as addBlock'. This has been referred to as **steps** in the entire document.

(c) Answer/Solution produced by the user:

The answer (program code) produced by the user is recorded, to analyze the accuracy of the result.

Feedback Form

At the end of the coding process, we presented users with a one page feedback form divided into two parts. The first part 'Background Information' collected Demographics/Personal data such as age, gender, country of residence, education, programming experience. The second part 'Experience with our Programming Environment' asked users their experiences and suggestions on the environment. This helped in analyzing users' perspectives on the environments. The feedback form can be viewed in appendix C.

3.3.7 Four Types of Analysis

Four different types of Analysis were performed on this experiment.

Analysis 1: Comparison between Flow and Block with different users

The initial analysis is to compare different users' experiences and performances on Flow and Block as their first environments. Each user is given a single combination from the 12 possible combinations (Question 1 with Flow, Question 1 with Block and so on).

Participants who completed programming on their initial environment were also given an opportunity to continue their coding process with either of the three combinations (Analysis 2, Analysis 3 or Analysis 4) given to them at random.

Analysis 2: Comparison between Flow and Block with the same user

This analysis compares same user's performance on the same problem question on both the environments. This consisted of two cases:

(a) Case1: Flow -> Block

In this case, if the initial environment is Flow-based, the following environment would be Block based.

(b) Case2: Block -> Flow

In this case, if the initial environment is Block-based, the following environment would be Flow based.

This is a comparison between users' who knew the problem question (by using Flow first) and then use the Block environment versus users' who knew the problem question (by using Block first) and then use the Flow environment. An analysis is made to determine which of the two cases improved (i.e. did block->flow improve more than flow->block or vice-versa).

Analysis 3: Comparison between Flow and Flow with the same user

In this analysis, if the initial environment is Flow-based, the following environment would be Flow-based as well. This analysis compared the same user's performance on Flow-based environment with two different problem questions.

Analysis 4: Comparison between Block and Block with the same user

In this analysis, if the initial environment is Block-based, the following environment would be Block-based as well. This analysis compared the same user's performance on Block-based environment with two different problem questions.

3.4 Threats to Validity of Results

Some external factors or threats could influence the data collected, which could turn the data into false or biased data. Certain preventive measures were taken to avoid such conditions:

- Design and Development

  If the environments were developed with a completely different approach and a huge difference existed in their layout, organization or method, results could have been affected in a significant manner. To avoid such a circumstance, both of the environments (Flow and Block) are developed similarly. The design of both the environments were customized to give a similar look and feel. They have similar layouts for the organization of different categories. The categories are organized in a bubble menu for both the environments. The overall colours of the blocks and the environment are the same as well. This is done in order to keep more focus on the coding process as opposed to focusing on differences in layout between the two environments.

- Common Code Base

  Both environments are developed within the same code base. They utilize common code for menus, calculations, widgets, etc.

- Set of Problem Questions

  The problem questions used for both environments (Flow and Block) are exactly the same.

- Feedback Form

  The feedback form used for both environments (Flow and Block) are exactly the same.

- Feasibility of Problem Questions for both Environments

  Problem questions are prepared in a way that they are not biased to either of the environments. If the problem question is feasible to solve in Flow environment, it is also feasible to solve the question in a Block environment.

- Help Documentation

  Help documentation for both environments have a similar pattern. The documentation starts with a short video tutorial giving an overview of the environment. The documentation then gives a detailed text explanation of the environment in terms of appearance, its different components and description of the blocks in each category with an example when required.

- Not biased to a particular user.

  Users were randomly chosen for this task. There was no special session conducted for the users by calling them at a particular place to do the task. Users did the task at their own setting.

- Four Types of Analysis

  The set-up of the 4 different types of analyses ensures integrity of the data without introducing bias in terms of the working of the solution between the two environments.

3.5 Determine usability

Section 3.3.6 describes the different methods to collect information. This section describes how the information is captured and analyzed followed by the method to determine usability of the two environments based on the results recorded. For analyzing the results recorded, an experimental design known as ‘Factorial Experimental Design’ is used in this research which will be described below.

3.5.1 Analysis of Gathered Information

The information collected through the tracking features (user mouse click, creation/destruction of nodes, feedback received and program code produced by the user) were captured through the generation of log files. For example, whenever a user changed tabs from help documentation to problem description, a time-stamped message is logged. This message indicates the time and date at which the user started reading help or problem pages and also records the clicks made on the respective tabs. Consider that the user clicked on the help documentation 5 times and the problem question 2 times. This would produce 5 click entries of the help documentation tab with time/date indicating when they started reading the help documentation and 2 click entries of the problem question tab with time/date indicating when they started reading the problem question.

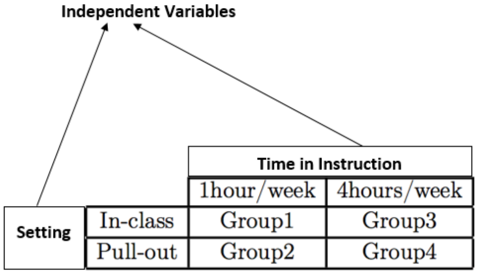
Similarly the solution or code produced by the users and the answers given on the feedback form by the users were also logged in the log files.

One log file gets generated for each user. This log file contains raw data that is converted into analytical data with the help of a script, which is written using the programming language 'Python' ("Python Software Foundation," 2001). The analytical data produced by the script is then converted into graph visualizations that helped in analyzing the usability of the two environments (Flow and Block) to the users.

Different types of dependent variables were collected by the script. The next section discusses these variables using the 'Factorial Experimental Design'.

3.5.2 Factorial Experimental Design

Before discussing the different types of data collected in detail, a clear understanding of the 'Factorial Experimental Design' is necessary. Consider a simple case: Let's assume a plan which includes an instructive program, where a look at the diversity of program varieties is needed to find out the best out of all. For example, one might want to change the measure of time the students are provided instruction: one gathering of students provided 1 hour per week and another gathering of students provided 4 hours per week. Also, they might want to fluctuate the setting with one gathering getting instruction in-class (presumably off into a side of the classroom) and the other gathering being pulled out of the classroom for instruction in another room. An obvious thing to do would be to have four separate groups to do this, yet when there is a need to change the measure of time in instruction, what setting should be used: in-class or pull out? Furthermore, when the setting needs to be considered, what measure of time should be utilized: 60 minutes, 4 hours, or something else? With factorial designs, we don't need to trade off while noting these inquiries ("Factorial Designs," 2006). Both ways can be possible on the off chance of crossing each of the two times in instruction conditions with each of the two settings as shown in Figure 3.26.



		Time in Instruction	
		1hour/week	4hours/week
Setting	In-class	Group1	Group3
	Pull-out	Group2	Group4

Figure 3.26: An example of Factorial Design.

Factorial Designs include two types of variables: Independent variables and Dependent variables. In the example above, 'Time in instruction' and 'Settings' are called factors or independent variables. A level is considered to be a subdivision of a factor/independent variable. In this case, there are two levels for 'time in instruction' and two levels for 'setting'.

Dependent variables in this example could be 'comprehension of instructions', how easy was it for the students to comprehend the instructions based on the two independent variables. Hence to generalize factorial designs, they can be understood as an experimental design that explores the impact of two or more independent variables on one or more dependent variables

(“Experimental Design Definition and Examples,” 2017).

Advantages of a Factorial Design (“Factorial Designs,” 2006):

1. Factorial designs are productive and efficient.
2. Rather than running a series of autonomous experiments, one can successfully join the different experiments into a single study.
3. Factorial designs can be useful when there includes a need to make decisions in investigating different group variations.
4. Lastly, according to “Factorial Designs” (2006), “factorial designs are the only effective way to examine interaction effect”.

3.5.3 Independent and Dependent variables

(a) Independent Variables

There are two types of independent variables:

- **Type of Environment (Flow or Block)** This has two types: Flow-based environment and Block-based environment.
- **Problem Question**

(b) Dependent Variables

The dependent variables identified for this research are:

1. Speed of Solution

This is the total time a user took to solve the problem question and produce a solution. This data is collected in the log files by the user click tracking feature.

2. Correctness of Solution

   This dependent variable determines the accuracy of the program code produced by the user. It establishes how close the user produced program code was to the optimum solution produced by the researcher. The solution is logged in the log files, once the user completed coding the answer and clicked on the 'Done' menu.

3. Average User Clicks

   This dependent variable records the average number of clicks by users on each of the three tabs: Help Documentation, Problem Description and Programming environment. This also records the average total number of clicks by users for solving the entire problem question. This data is collected in the log files by the user click tracking feature.

4. Average Time Spent

   This dependent variable records the average time spent by users on each of the three tabs: Help Documentation, Problem Description and Programming environment. This and the dependent variable 'Average User Clicks' together helped in establishing the difficulty level of the Help documentation, Problem Description and Programming environment as well as the intuitive understanding on the environments. This data is collected in the log files by the user click tracking feature.

5. Difficulty Level of Programming Environment

   This dependent variable determines how easy or difficult was it to get acquainted with the overall programming environment and code on it. It also analyses the utility of the organization of blocks in different categories, which means if the categories defined were comprehensible and not confusing. This data is collected in the log files through the feedback received from the users.

6. Complexity Level of Problem Question

   This helped in determining the complexity level of the question (easy or difficult to understand). This data is collected in the log files through the feedback received from the users.

7. Usefulness of Help Documentation

   As already mentioned, there were two releases of this experiment, one without video tutorial in the Help Documentation and one with video tutorial added to the Help Documentation. We wanted to determine if the help documentation was easy or difficult to understand in both releases and if it was easily accessible. This data is collected in the log files through the feedback received from the users.

8. Consistency in terms of Syntax and Structure

   Consider a user just started using the environment, by pulling out a few blocks from different categories and tried forming a simple program. The question here is was it easy for the user to understand similar blocks and categories of the environment? Was the environment consistent in that aspect? This dependent variable measures the consistency of the environment from not at all consistent to being extremely consistent. This data is collected in the log files through the feedback received from the users.

9. Easy to Grasp

   This dependent variable helps in determining if the environment was easy enough to be grasped by its users. This data is collected in the log files through the feedback received from the users.

10. Difficulty in Remembering Lexemes

    As mentioned in section 2.1.5 under “Diffuseness/Terseness”, “VPLs have a good number of lexemes in terms of icons, connectors, etc.” For Flow-based environment: pipes, input blocks, output blocks etc. are the lexemes used. For Block-based environment: assignment blocks, statement blocks etc. are the lexemes used. Since there are many blocks used in different categories for both Flow and Block environments, it would be significant to determine if the user finds it hard to remember so many of the blocks or they were easy enough to be conserved by one's memory. This data is collected in the log files through the feedback received from the users.

11. Ease of Notations

    An example for this is presented with the help of 'conditional loops' in programming in the section 2.1.5 under "Hard Mental Operations". Most of the times the issue lies at the notational level rather than the semantic level, as indicated by Green and Petre (1996). Therefore, it is necessary that the notations be intuitive and easy to use. This dependent variable checks if labels on the blocks for Flow and Block environments are easy enough for a non-programmer or a novice-programmer to understand, and if they are closer to the natural languages. This data is collected in the log files through the feedback received from the users.

12. Visual Layout of the Blocks

    This dependent variable helped in determining if the visual layout of the blocks was comfortable and easy enough to form a program (attaching blocks like a puzzle in the Block-based environment and connecting blocks with pipes in the Flow-based environment). This data is collected in the log files through the feedback received from the users.

13. Measure of Viscosity

    This dependent variable is measured in terms of resistance to local change, on how easy was it to change a part of the program such that it would reflect at other places too. An explanation for this has already been mentioned in section 2.1.5 under "Viscosity: Resistance to Local Change". Viscosity was measured for Flow and Block environments to determine which one has a lower one. This data is collected in the log files through the feedback received from the users.

14. Error Tolerance

    This dependent variable determines if the environments (Flow and Block) had good error notifications that were easy to understand and helped in rectifying the errors. This data is collected in the log files through the feedback received from the users.

15. Level of Engagement

    This dependent variable determines how engaging the environments were to them. This data is collected in the log files through the feedback received from the users.

Chapter 4

Results

There were two Releases of the environments that took place for this experiment. The first Release did not have a video tutorial included in the Flow and Block Help Documentations, while the second Release had a video tutorial included in both Flow and Block Help Documentations. The first release had 97 users and the second release had 192 users. The results are separated into four parts: 'Different Types of Analysis', 'User Clicks and Time Spent', 'Results from Survey', and 'Demographics of Participants' for both the Releases. The results for both the releases are not statistically different as determined by the p-value calculation in most of the analyses and hence are combined, except for the 'User Clicks and Time Spent' part, in which the data will be represented in two ways - with and without video tutorial (where the difference in results was high). Overall, it was observed that there were no statistically significant differences in terms of usability and effectiveness between the two environments.

4.1 Mann-Whitney U Test

I used the Mann-Whitney U test to perform a statistical analysis on the results collected. The Mann-Whitney U test is used to compare differences between two independent groups when the dependent variable is either ordinal or continuous (“Mann-Whitney U test,” 2017). It is used to test whether the two independent groups are equal or not. For example, one could use the Mann-Whitney U test to understand what type of vaccine is better suited to treat a particular type of the disease (i.e., the dependent variable would be ‘treatment of the disease’ and the independent variable would be ‘vaccines’, which has two types: ‘Vaccine A’ and ‘Vaccine B’.

The test was initially designed in 1945 by Wilcoxon for two samples of the same size and was further developed in 1947 by Mann and Whitney to cover different sample sizes. Thus, the test is also called Mann-Whitney-Wilcoxon (MWW), Wilcoxon rank-sum test, Wilcoxon-Mann-Whitney test, or Wilcoxon two-sample test.

Assumptions of the Mann-Whitney: Mann-Whitney U test is a non-parametric test, so it does not assume any assumptions related to the distribution of scores. However, there are some assumptions that are assumed:

1. The sample drawn from the population is random.
2. Independence within the samples and mutual independence is assumed.
3. Ordinal measurement scale is assumed.

For determining whether the difference between the two populations is statistically significant or not, the p-value calculated is compared to the significance level. Usually, a significance level (denoted as α or alpha) of 0.05 is mostly used and works well (“Mann-Whitney U test,” 2017; “Hypothesis Testing (P-value approach),” 2017).

The below conditions are used throughout this chapter wherever related calculations are performed.

p-value $> \alpha$ (0.05): The difference between the two populations is not statistically significant (Accept null hypothesis H0).

If the p-value is greater than the significance level, the decision is to accept the null hypothesis. There is not enough evidence to conclude that the difference between the two populations is statistically significant.

p-value $\leq \alpha$ (0.05): The difference between the two populations is statistically significant (Reject H0).

If the p-value is less than or equal to the significance level, the decision is to reject the null hypothesis. One can conclude that the difference between the populations is statistically significant.

4.2 Comparison between Flow and Block with different users

This analysis was to compare users' performance on Flow and Block as their first environments. A total of 289 users participated in this study distributed over six problem questions with 146 users in Flow and 143 users in Block. Each user was provided a randomly selected question using one of the environments, randomly selected as well. Figure 4.1 illustrates the distribution of total number of users in Flow and Block environments for each problem question. The blue bar indicates Flow environment and green bar indicates the Block environment for this entire chapter.

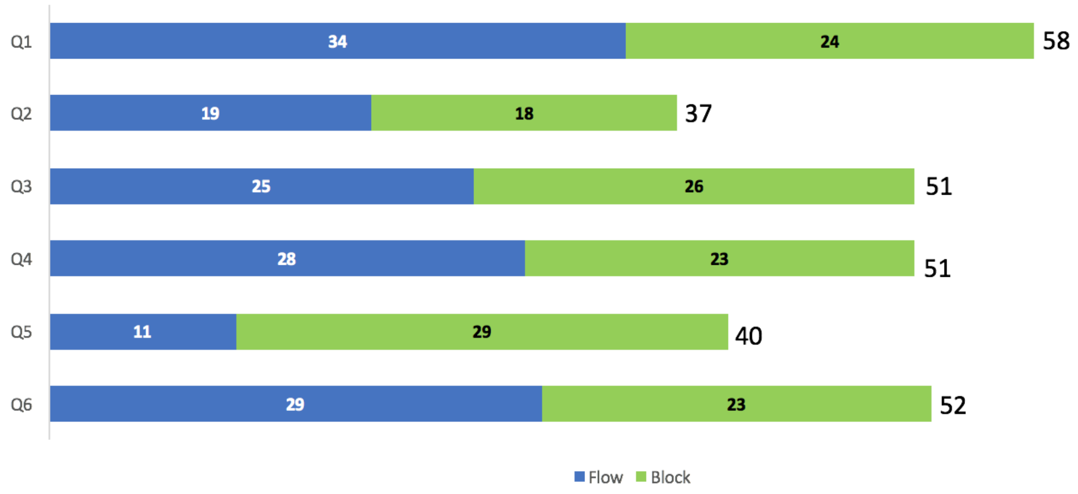


Figure 4.1: Distribution of the Total number of users for each question in Flow and Block.

In this section, the independent variable is the ‘type of environment’ which has two parts: Flow and Block and the dependent variable is the ‘correctness of solution’.

The null hypothesis (H0) is: There are no statistically significant differences between Flow and Block in terms of producing the correct solution.

The alternative hypothesis (Ha) is: There are statistically significant differences between Flow and Block in terms of producing the correct solution.

The dependent variable ‘Correctness of Solution’ was analyzed with respect to its division in four parts as discussed below.

(1) Steps and Correct Solution:

In this case, users took the right approach and coded the optimal program.

(2) Steps and Incorrect Solution:

In this case, users produced a program code, but an incorrect solution.

(3) Steps and No Solution:

In this case, users took an initiative to code but could not produce any solution.

(4) No Steps and No Solution:

In this case, users abandoned the environment without doing anything on it.

Users were evaluated with a range of values from 0-10, where 10/10 was given to users who coded the optimal solution, 7/10 was given to users who produced an incorrect solution, 3/10 was given to users who could not produce any solution and 0/10 was given to users who abandoned the environment without doing anything on it.

Using the above grading scale, a weighted mean to determine the average performance of users for each question in both Flow and Block is calculated. This weighted mean determines how close the users came to the optimal solution on the scale of 0-10, with 0 being the lowest and 10 the highest. This weighted mean of the dependent variable 'Correctness of Solution' is shown in Figure 4.2

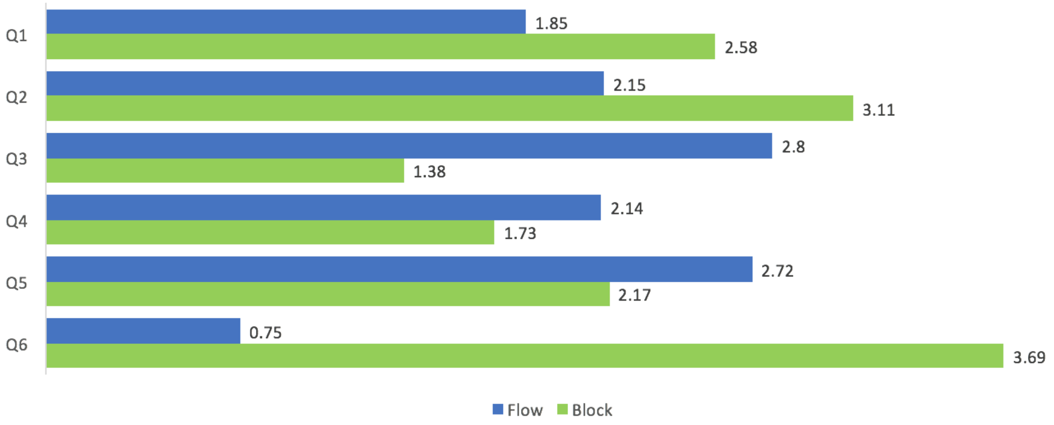


Figure 4.2: Weighted mean to determine 'Correctness of Solution' for each question for Flow and Block.

Using the Mann-Whitney U test, p-values were calculated to determine the average performance of users for each question with respect to Flow and Block being the two populations.

This was the sampling distribution used in all the analyses in order to perform the Mann-

Whitney U test.

This analysis of the dependent variable 'Correctness of Solution' for Analysis 1 is shown in Figure 4.3.

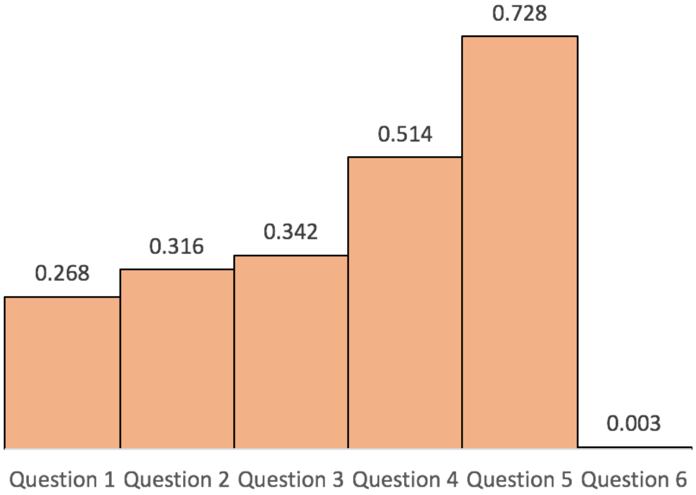


Figure 4.3: p-values for each question with Flow and Block as two populations.

For questions 1 to 5, the p-values are higher than the significance level, which supports the null hypothesis. Hence, there are no statistically significant differences between Flow and Block with regard to questions 1 to 5. However, an outlier can be seen with regard to question 6, where the p-value is less than the significance value, which indicates a significant difference in the result only with the question 6. After analyzing the individual solutions of people in both Flow and Block for question 6, it was concluded that users performed better in Block environment as compared to Flow environment. An overall calculation including all the questions was also made, which indicated the p-value to be 0.195, which also supported the null hypothesis.

4.3 Comparison between the same user working on two different or same environments

We concluded certain results from the comparisons made between the same user working on two different or same environments. From the p-value calculations and after checking the user's individual performances on their first and the second environment, a conclusion was made indicating users performing better on their first environment as compared to their second environment. We had anticipated that users would show learning effect on their second environment, instead we saw exhaustion effect from them while coding on the second environment. The reasons of poor performance on the second environment could be because of tiredness, not being paid to do the task, or user interfaces not being engaging enough.

4.3.1 Comparison between Flow and Block with the same user

This analysis was to compare same user's performance on the same problem question on both the environments. Since the problem question was the same, the users did not have to analyze or understand the question again, all they needed to do was to get familiar with the second environment's features.

The two cases analyzed are:

(a) Case1: Flow -> Block

A total of 11 users participated in this case. Mann-Whitney U test was used to compare the success in Flow and success in Block and the resulting p-value was 0.04. This indicated that there was a significant difference in the performance of users on both environments.

(b) Case2: Block -> Flow

A total of 9 users participated in this case. Mann-Whitney U test was used to compare the success in Flow and success in Block and the resulting p-value was 0.07. This indicated there was a significant difference in the performance of users on both environments.

However, when I compared these users' (who performed on both different environments) tasks on their first environments only, a p-value of 0.418 was calculated, which indicated no significant differences between users' performances on their first environment be it either Flow or Block as shown in Figure 4.4. Also, when I compared these users' tasks on their second environments only, in terms of already being acquainted with the problem question, a p-value of 0.397 was calculated which again indicated no significant differences between users' performance on their second environments be it either Flow or Block as shown in Figure 4.4.

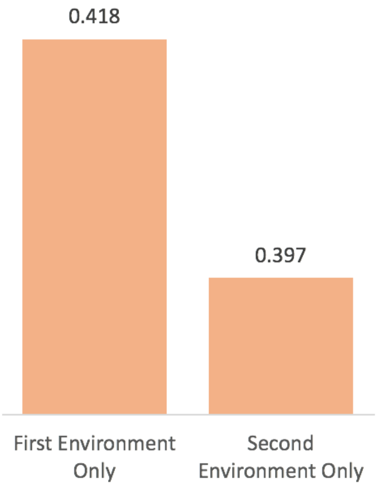


Figure 4.4: P-values indicating performance of users on their first or second environments only.

4.3.2 Comparison between Flow and Flow with the same user

This analysis compared the same user's performance on Flow-based environment with two different problem questions. A total of 17 users performed on both Flow environments. A p-value of 0.005 was calculated in this analysis which indicated a significant performance difference in the performance of users between both environments.

4.3.3 Comparison between Block and Block with the same user

This analysis compared the same user's performance on Block-based environment with two different problem questions. A total of 18 users performed on both Block environments. A p-value of 0.07 was calculated in this analysis which indicated a significant performance difference in the performance of users between both environments.

However, after observing both the comparisons in section 4.3.2 and section 4.3.3 respectively, when I compared these users' (who performed on both same environments) tasks on their first environments only, a p-value of 1.0 was calculated, which indicated no significant differences between users' performances on their first environments (Flow and Flow) as shown in Figure 4.5. Also, when I compared these users' tasks on their second environments only, a p-value of 0.640 was calculated which again indicated no significant differences between users' performance on their second environments (Block and Block) as shown in Figure 4.5.

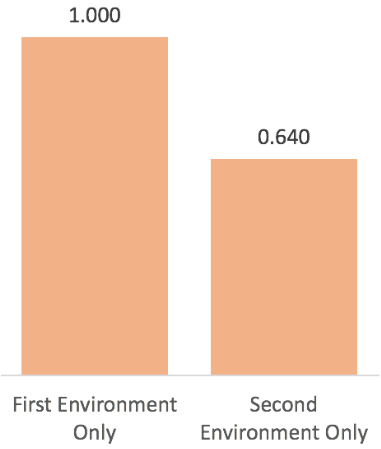


Figure 4.5: P-values indicating performance of users on their first or second environments only.

4.4 User Clicks and Time Spent

This section shows the mean and standard deviation of the number of times users clicked on each of the three tabs - Help Documentation Tab, Problem Description Tab and Programming Environment Tab. This data is shown both for Release 1 and Release 2 due to a considerable difference in their performance with and without video tutorial in the Help Documentation.

4.4.1 User Clicks

Figure 4.6 and Figure 4.7 show the mean and standard deviation of the number of user clicks on each of the three tabs for Release 1 and Release 2 respectively.

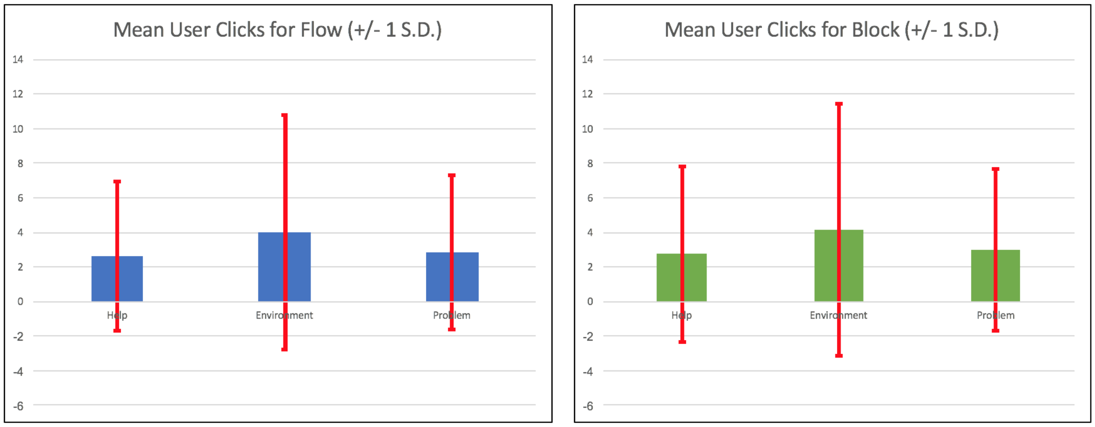


Figure 4.6: Number of user clicks on each tab for Release 1 - Flow and Block

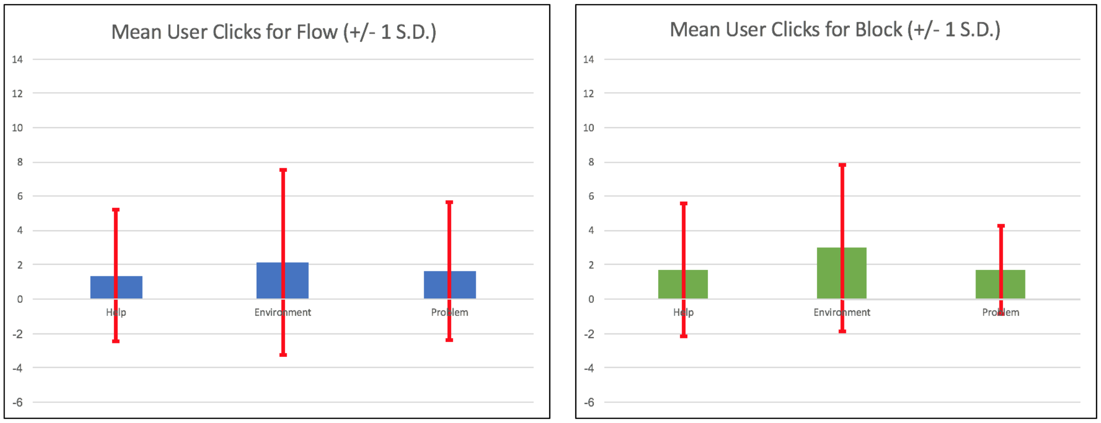


Figure 4.7: Number of user clicks on each tab for Release 2 - Flow and Block

In the release without video tutorial, higher number of clicks can be seen on each of the three tabs as compared to the release with video tutorial. From this observation and the user's comments, it was concluded that users' found the Help Documentation significantly easier after the video tutorial was added to it. The video tutorial also helped them in understanding the environment clearly and faster.

The standard deviation is higher than the mean for all the three tabs in both the releases, which indicates that the data was not a normal distribution.

4.4.2 Time Spent

Figure 4.8 and Figure 4.9 show the mean time spent and standard deviation on each of the three tabs for Release 1 and Release 2 respectively. The mean time shown in the graphs is in seconds.

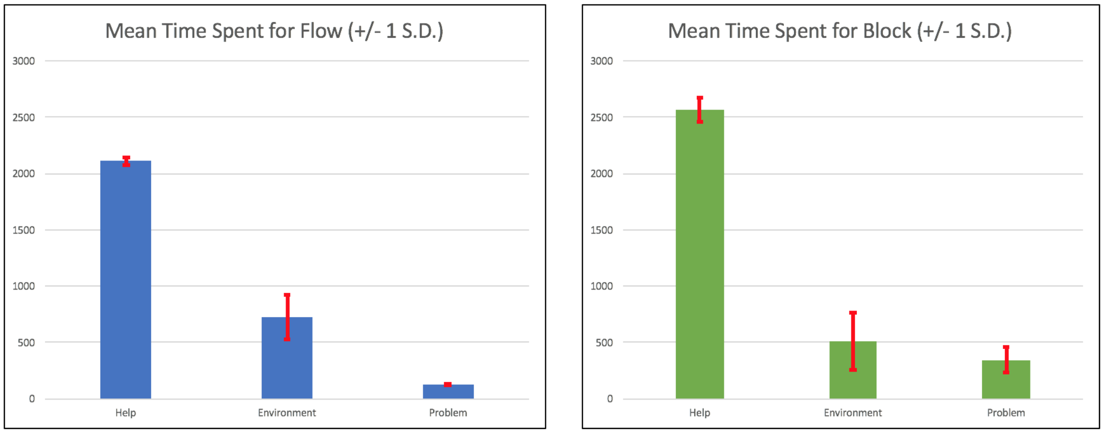


Figure 4.8: Time spent on each tab for Release 1 - Flow and Block

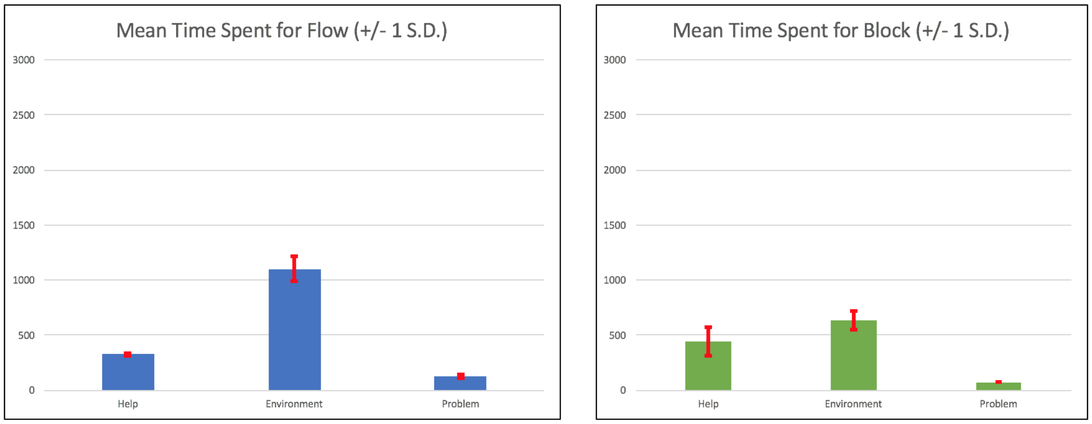


Figure 4.9: Time spent on each tab for Release 2 - Flow and Block

In the release without video tutorial, users spent more time on the Help Documentation rather than on Programming Environment, whereas in the release with video tutorial, users spent more time on the Programming Environment rather than on Help Documentation. The difference is by a large margin.

As observed from the mean time in both the releases, users' spent more time on the Flow Programming Environment than Block Programming Environment. This could be possible because of the difficulty users' faced in connecting and disconnecting pipes on the Flow user interface, which was evident from the user comments. In both the releases, users spent slightly less time on the Flow Help Documentation as compared to the Block Help Documentation. This could be possible due to the Help for Flow being slightly easier than Help for Block, which was also indicated in the users' comments.

For the release without video tutorial, in the case of the Help tab, the standard deviation is low compared to the mean which indicates that most users spent nearly the average amount of time on Help for both Flow and Block. In the case of environment, the standard deviation is high for both Flow and Block, which indicates some users have spent more time on the environment while some others have spent less time on the environment. In the case of problem description for Flow, the standard deviation is low, which indicates that most users spent nearly the average amount of time reading the problem question, whereas for Block, the standard deviation is more, which indicates some users have spent less time and some more time on problem question.

For the release with video tutorial, in the case of the Help tab for Flow, the standard deviation is low, which indicates that most users spent nearly the average amount of time on Help, whereas for Block, the standard deviation is high, which indicates some users have spent less time and some more time on Help. In the case of environment and problem description, the standard deviation is low for both Flow and Block as compared to mean, which indicates that most users spent nearly the average amount of time on both Flow and Block environments.

4.5 Results from Survey

A total of 92 surveys were received from the users. This section describes the user's experiences on the environments and are the dependent variables for this experiment.

4.5.1 Difficulty Level of the Problem Question

Figure 4.10 illustrates the distribution of users in Flow and Block environments, according to the user's feedback on how easy or difficult the problem questions were to them. The results are nearly similar for both Flow and Block environments.

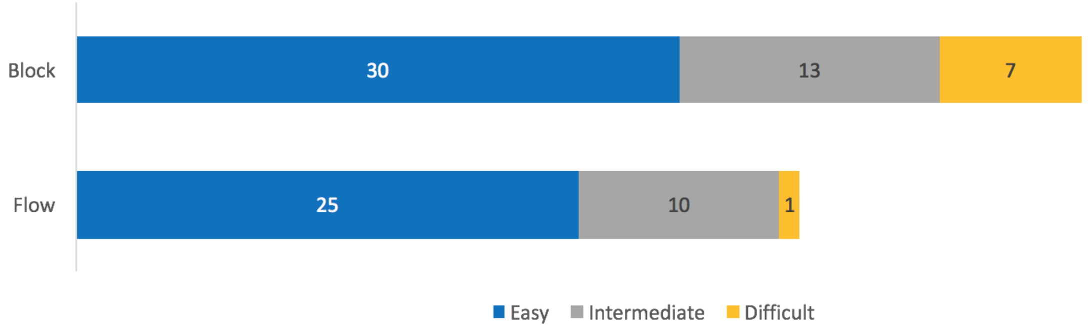


Figure 4.10: Difficulty Level of the Problem Question Distribution of Flow and Block users.

4.5.2 Difficulty Level of the Programming Environment

Figure 4.11 illustrates the distribution of users in Flow and Block environments, according to their feedback on how easy or difficult the overall programming environment was to them. The results are nearly similar for both Flow and Block environments.

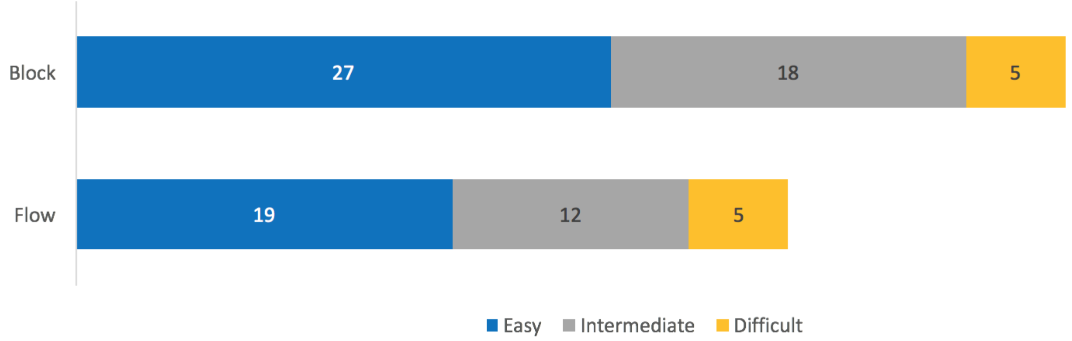


Figure 4.11: Difficulty Level of the Programming Environment Distribution of Flow and Block users.

4.5.3 Difficulty Level of the Help Documentation

Figure 4.12 and Figure 4.13 illustrate the distribution of users for Release 1 and Release 2 respectively in Flow and Block environments, according to their feedback on how easy or difficult the help documentation was to them. The Help documentation became easy to users by a large margin after the video tutorials were added to it. Flow and Block Help are nearly similar to users in their difficulty level.

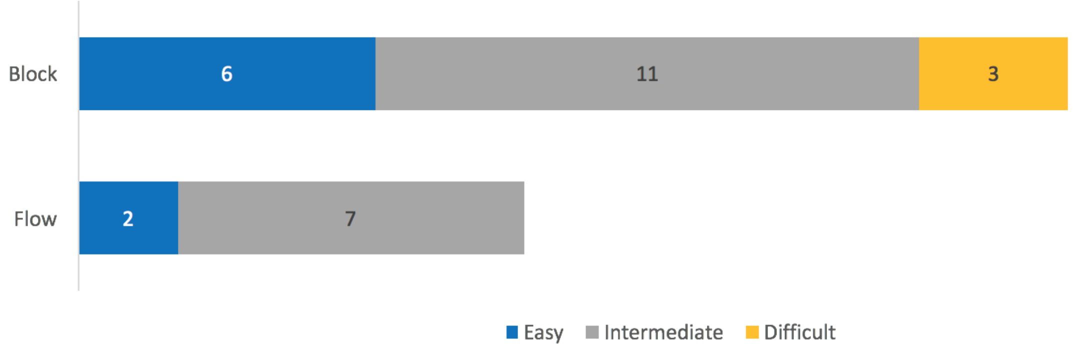


Figure 4.12: Difficulty Level of the Help Documentation Distribution of Flow and Block users for Release 1

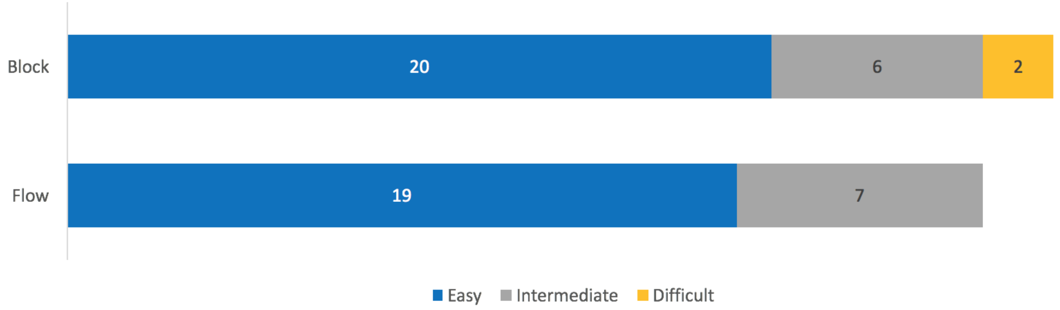


Figure 4.13: Difficulty Level of the Help Documentation Distribution of Flow and Block users for Release 2

The next few dependent variables from ‘Consistency’ until ‘Engaging’ measured the user’s feedback on their experiences with the programming environment, in terms of four factors: Not at all, Slightly, Very and Extremely. The answer varies depending on the type of question.

These dependent variables are described in the Chapter 3.

For this section, the null hypothesis (H0) is: There are no statistically significant differences between Flow and Block in terms of users' experiences on the environment.

The alternative hypothesis (Ha) is: There are statistically significant differences between Flow and Block in terms of users' experiences on the environment.

Using the Mann-Whitney U test, p-values for each of these dependent variables were calculated with Flow and Block being the two populations and can be seen in the Figure 4.14.

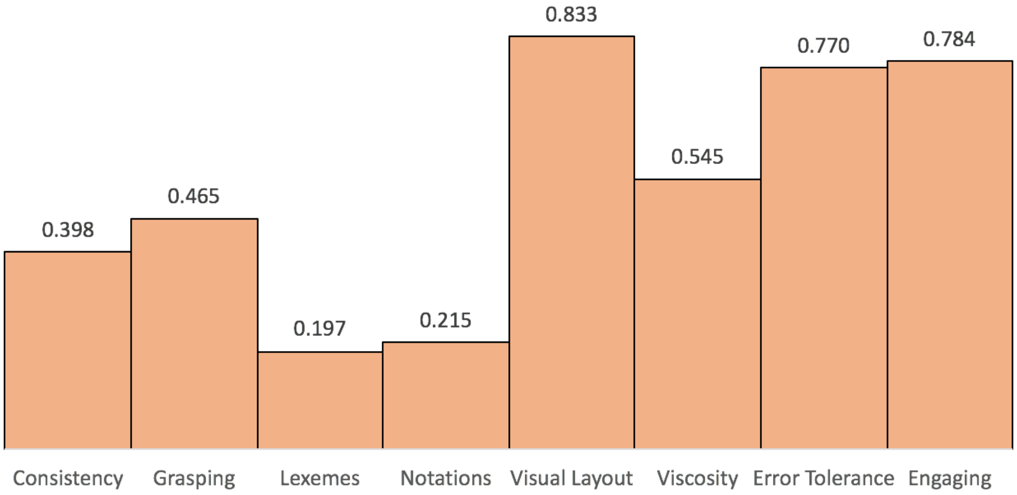


Figure 4.14: P-Value for dependent variables from 'consistency' to 'engaging'.

The p-values for all the dependent variables are higher than the significance level of 0.05, which supports the null hypothesis. Hence there are no statistically significant differences between Flow and Block with regards to experiences of users on the two environments.

4.6 Demographics of Participants

This section includes the background information collected about the users from the survey that are potentially independent variables.

4.6.1 Age

Figure 4.15 illustrates the distribution of users in Flow and Block environments according to their ages. There were six age groups in total. From the graph shown, it is clear that the environments had been distributed for the ages 10 - 60plus years.

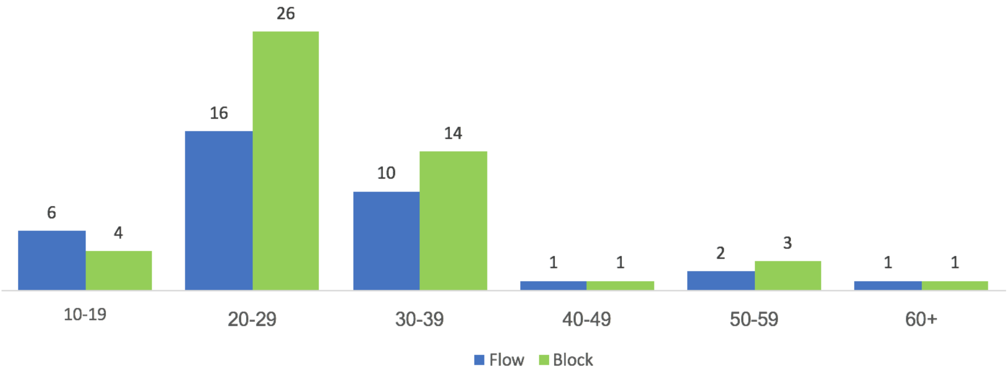


Figure 4.15: Age Distribution

4.6.2 Gender

Figure 4.16 illustrates the distribution of users in Flow and Block environments according to their genders. From the graph shown, it is clear that there were more males doing the task than females, a large difference in both the environments.

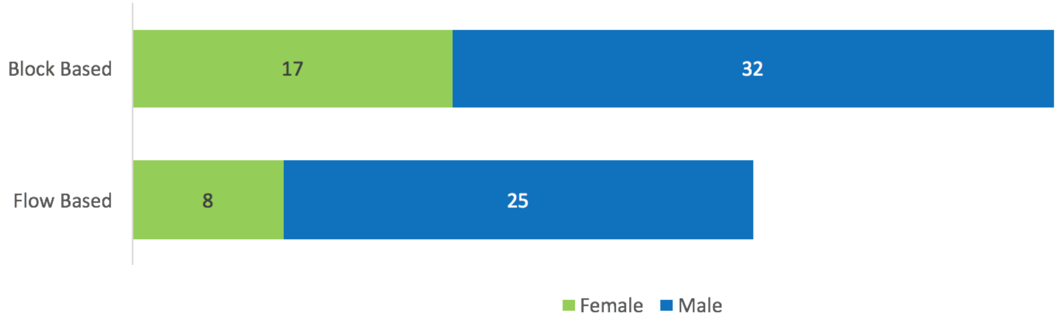


Figure 4.16: Gender Distribution

4.6.3 Country of Residence

Figure 4.17 illustrates the distribution of users in Flow and Block environments according to their country of residence. Users have done tasks from countries such as Canada, India, United States of America, Australia, Oman and Germany.

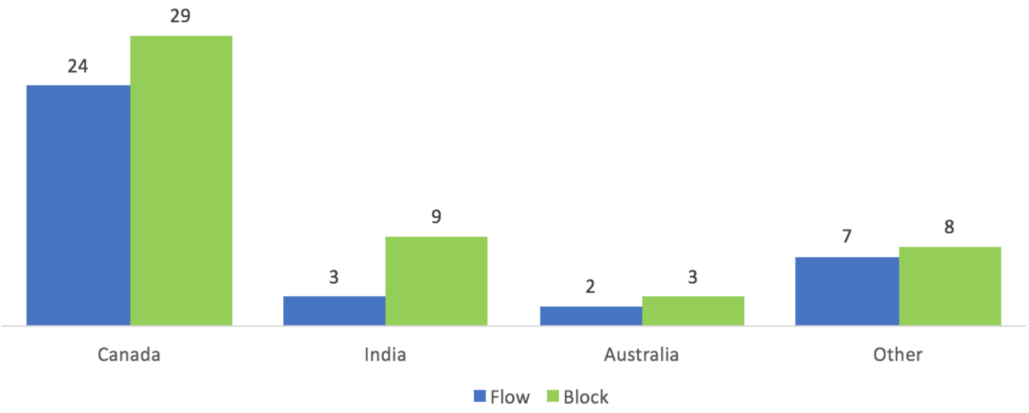


Figure 4.17: Country Distribution

4.6.4 Educational Background

Figure 4.18 illustrates the distribution of users in Flow and Block environments according to their educational background.

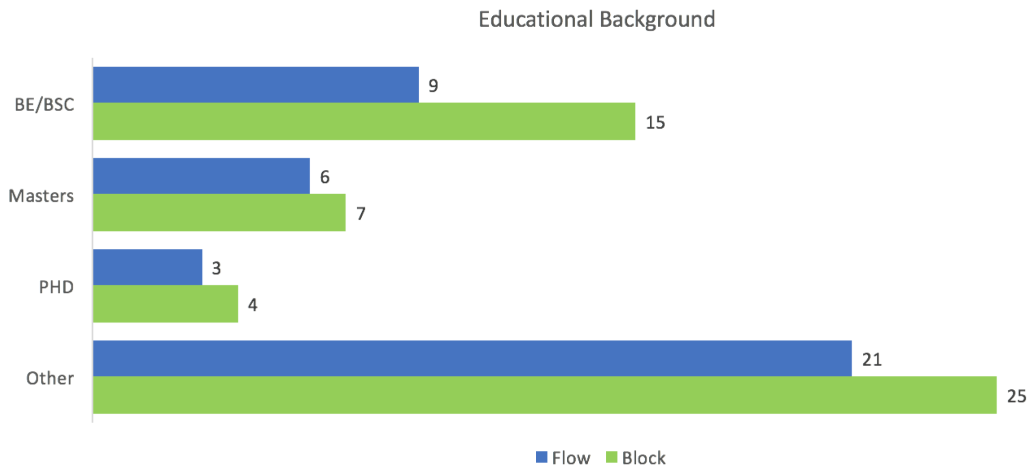


Figure 4.18: Educational Background Distribution.

Figure 4.19 illustrates the weighted mean and standard deviation of users' performance in computer science and non-computer science background for both Flow and Block. Computer science background includes users from Bachelors in Computer Science, Masters in Computer Science, PHD in Computer Science or related fields. This weighted mean evaluates how close the user came to the optimal solution and was calculated using an evaluation grading scale of 0-10, where 10/10 was given to users who coded the optimal solution, 7/10 was given to users who produced an incorrect solution, 3/10 was given to users who could not produce any solution and 0/10 was given to users who abandoned the environment without doing anything on it. The same evaluation scheme was used for the section 4.6.5 and section 4.6.6 as well.

The standard deviation is low compared to the mean for both Flow and Block in all the cases, which indicates that most of the users got a grade near to the mean grade.

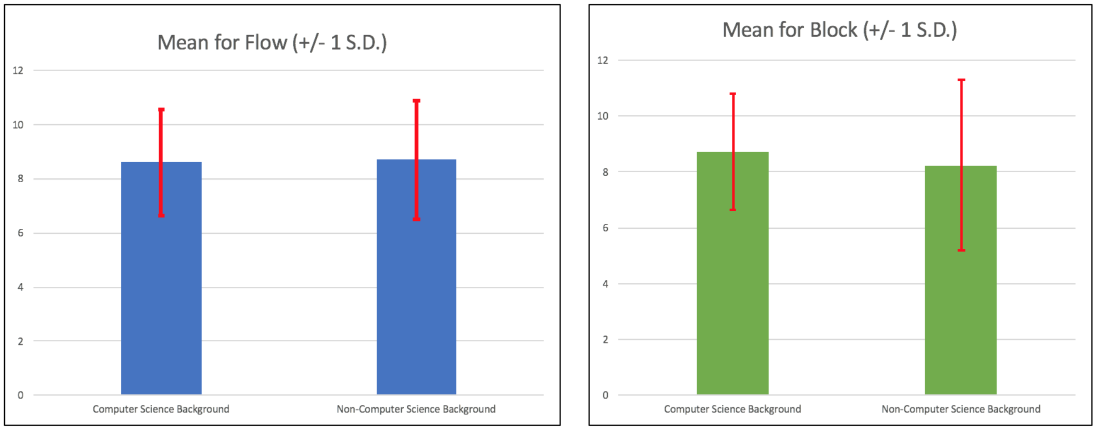


Figure 4.19: User performance for Flow and Block according to their educational background.

4.6.5 Programming Background

Programming Background of users was divided in three parts:

1. Users who had Text-based programming experience.
2. Users who had VPL programming experience.
3. Users who had no programming experience.

Figure 4.20 illustrates the distribution of users in Flow and Block environments according to their years of experience in programming.

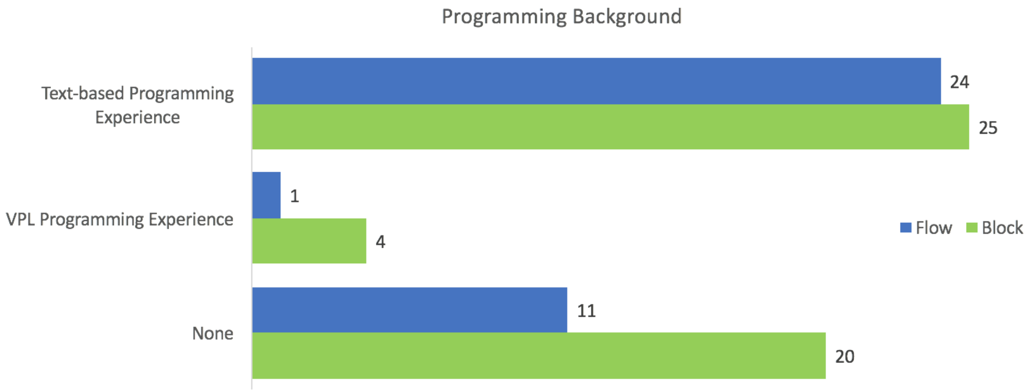


Figure 4.20: Distribution of users according to their Programming Background.

Figure 4.19 illustrates the weighted mean and standard deviation of users' performance according to their programming background for both Flow and Block.

A large number of users were not from any programming background according to both the graphs for Flow and Block. The standard deviation is low compared to the mean for both Flow and Block in all the cases, which indicates that most of the users got a grade near to the mean grade.

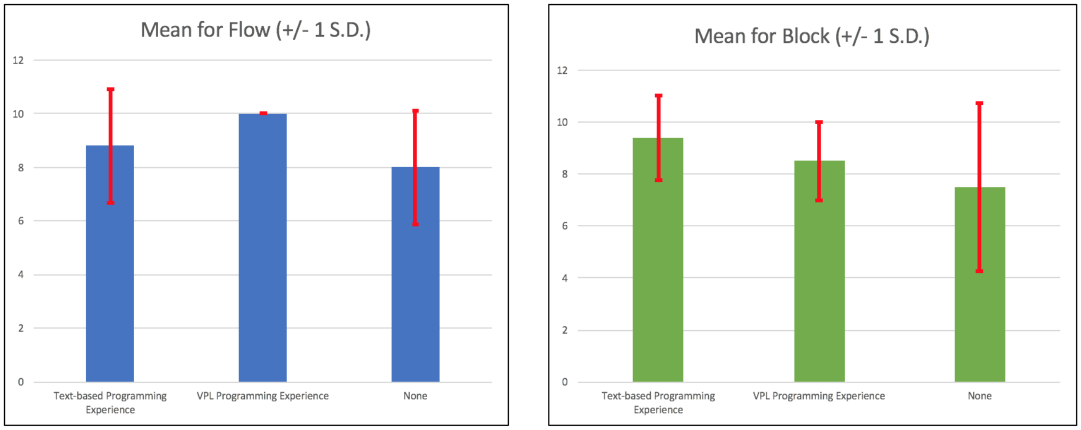


Figure 4.21: User performance for Flow and Block according to their Programming Background.

4.6.6 Years of Experience in Programming

Figure 4.22 illustrates the distribution of users in Flow and Block environments according to their years of experience in programming.

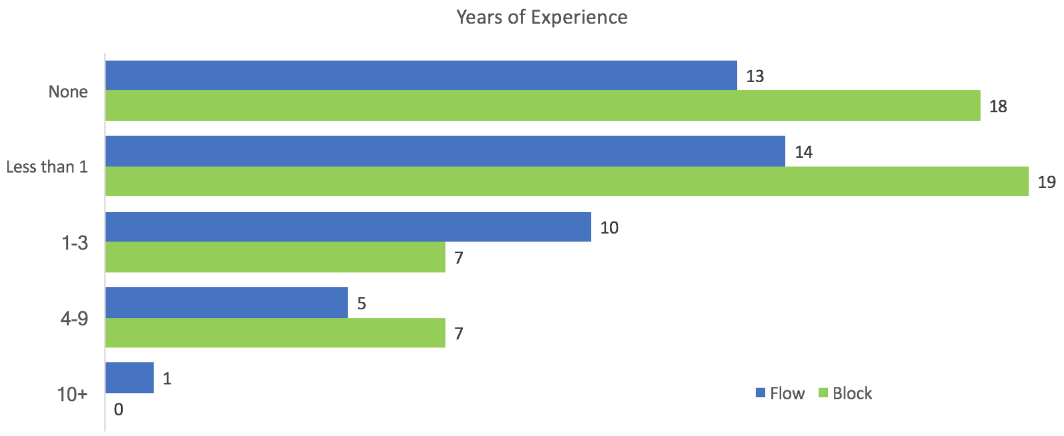


Figure 4.22: Distribution of users according to their Years of Experience in Programming.

Figure 4.19 illustrates the weighted mean and standard deviation of users' performance according to their years of experience in programming for both Flow and Block.

In the graph shown below, a high percentage of users in both Block-based and Flow-based had less than one or no experience in programming. The standard deviation is low compared to the mean for both Flow and Block in all the cases, which indicates that most of the users got a grade near to the mean grade.

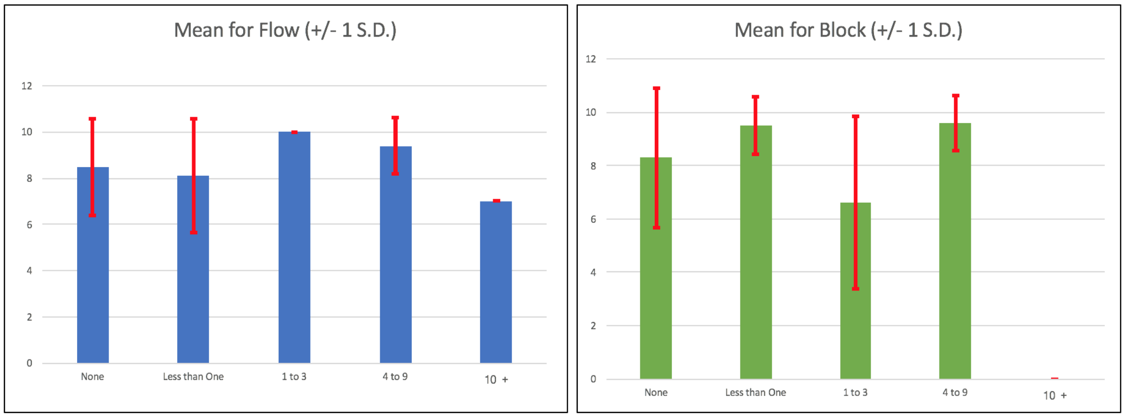


Figure 4.23: User performance for Flow and Block according to their Years of Experience in Programming.

4.6.7 Preference to VPLs or Text-based Programming

Figure 4.24 illustrates the distribution of users in Flow and Block environments, according to the user's feedback on if the Flow-based and Block-based environments were more preferable as compared to the general purpose Text-based programming.

A high number of users for both Flow and Block have given a higher preference to VPL paradigm compared to the Text-based paradigm.

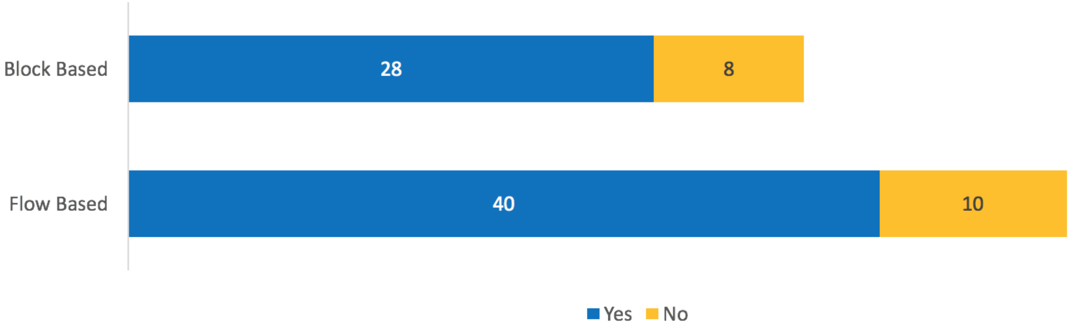


Figure 4.24: Preference to VPLs or Text-based Programming Distribution of Flow and Block users.

We were initially planning to run this experiment on Amazon Mechanical Turk ("Amazon Mechanical Turk," 2005). However, due to logistical problems, we were not able to run that part of the experiment.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The conclusions are divided into four parts in accordance with the four parts in the Results chapter.

5.1.1 Types of Analysis

Overall, Flow and Block have been observed to have no statistically significant differences between them in producing the optimal solution.

5.1.2 User Clicks and Time Spent

After the addition of video tutorials for Release 2, the results were considerably improved. There were no significant differences in the mean and standard deviation between Flow and Block in terms of User Clicks and Time Spent.

5.1.3 Results from Survey

From the feedback questions asking users if they found the dependent variables - Programming Environment, Problem Question and Help Documentation easy, intermediate or difficult, a high number of users found the Problem Questions and Programming Environment ‘easy’ and Help Documentation ‘easy’ (especially with video tutorials) in both Flow and Block.

In other feedback questions asking their experiences on the overall programming experiment, there were no statistically significant differences between the two environments in terms of dependent variables such as consistency, gasping, lexemes, notations, visual layout, viscosity, error notifications, engaging, and preference.

5.1.4 Demographics of Participants

A large pool of users had a non-computer science educational background such as Mechanical Engineering, Law, Bachelor of Arts, Masters in Surgery, Dropouts, Data Analyst, High School students, Graphics Communication, Finance and Accounting etc. A large pool of users did not have any programming background as well and had less than one year or no experience in programming. A small number of users had taken online coding courses or worked on VPLs like Scratch before.

5.1.5 Overall Summary

The diverse qualities and backgrounds of the users in this experiment and a large user pool of around 289 users have contributed in uncovering strategies that novice programmers or non-programmers may utilize or realize when they analyze data and perform programming tasks. The different approaches and methods utilized by the users to reach the solution discussed in section 4.2, section 4.3 and section 4.4 have provided useful insights on how the users strategize their approach while coding on a visual programming environment. Also the feedback and suggestions collected from the users will further enhance the design and development of the Flow and Block environments.

While some features of Flow have been liked by some users, some other features of Block have been liked by some other users. It is a very difficult comparison to make in terms of concluding one paradigm to be better than the other overall. Some users have commented on how the environment was very good for beginner programmers and would encourage them to go to advanced levels of programming like the Text-based programming referring to Block-based, while some other users commented how the environment could help a lot of people in analyzing data or do programming, if coding can be made this easy referring to Flow-based. Users took some time to get familiar with both environments and had to get accustomed to how the drag and drop system works in case of Block and how connecting and disconnecting of pipes work in Flow. Difficulties arose for some users in understanding the dropping of blocks one inside the other for Block and connecting pipes smoothly in Flow. These users would have preferred if the right method to do both of these things would have been described in the video tutorial. From the results, it seems like some users are more comfortable with Block and some other are more comfortable with Flow.

This research has the potential to directly benefit people by making use of the extensive data around them. However, there is also a perspective aspect to this study from the views of a developer and a researcher. A developer who needs to comprehend a programming environment to perform development and maintenance tasks and a VPL could make their tasks faster. A researcher who could utilize the discoveries made in this thesis (methodologies and results) to plan and design strategies, techniques, and tools to further analyze the results.

Development of a new programming language which will satisfy non-programmers or novice programmers completely can be very challenging. A non-programmer will not take the effort of learning syntax or the structure of a programming language like one needs to do in case of a Text-based programming language. They would want something to make their task simpler, data accessible and also reduce the learning constraints to the minimum.

This allows for language evolution that is extremely unlikely in the traditional Text-based languages.

For now, the Flow and Block environments support small scale approaches and will be developed further taking into consideration user's experiences and suggestions on the existing environments. We are following agile methodology for constructing the environments from a user's perspective.

5.2 Contributions

1. The development of the Block-based environment by me will help non-programmers and novice programmers to analyze data and perform programming tasks.
2. This research was the first one that did comparison of two most common visual programming paradigms - Flow and Block.
3. The results indicated no significant differences between Flow and Block in terms of usability and effectiveness.

5.3 Future Work

This section will first describe methods to improve and expand the existing research, and then outline some potential improvements on the Flow and Block environments.

After making improvements to both of the environments taking user's feedback and experiences, it would be valuable to run this experiment on Amazon Mechanical Turk ("Amazon Mechanical Turk," 2005) and analyze the result again.

It would also be interesting to compare Flow-based and Block-based visual programming languages versus textual programming languages for non-programmers or novice programmers. Most of the VPLs have been tested on school students either primary, secondary, high school or even university students. There has been no research showing empirical data comparing VPLs with Text-based Languages with no age or educational background user requirement.

It was noticed while analyzing the results that there has been a considerable amount of idle time by users while coding. Hence, it would be beneficial to explore and produce research on how the Flow and Block interfaces could minimize the idle time.

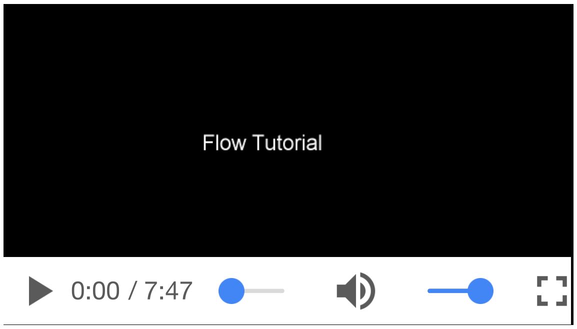
The results and observations were accompanied with a list of recommendations to be considered for future improvisations of the two environments. The Flow-based and Block-based environments developed are constantly evolving, and have many features planned for their future implementations. One of them is to allow users to create compound blocks made from a group of primitive blocks. With this feature, the user would be able to make customized blocks from pre-existing ones. Another feature is to include a side toolbox which would only have the regularly used blocks or a copy feature that would allow users in clicking on the block and copying it. Finally, we plan to include these environments on Phones and Tablets in the future.

Appendices

Appendix A

Help Documentation for Flow-based Environment

Flow Based Programming Language User Guide



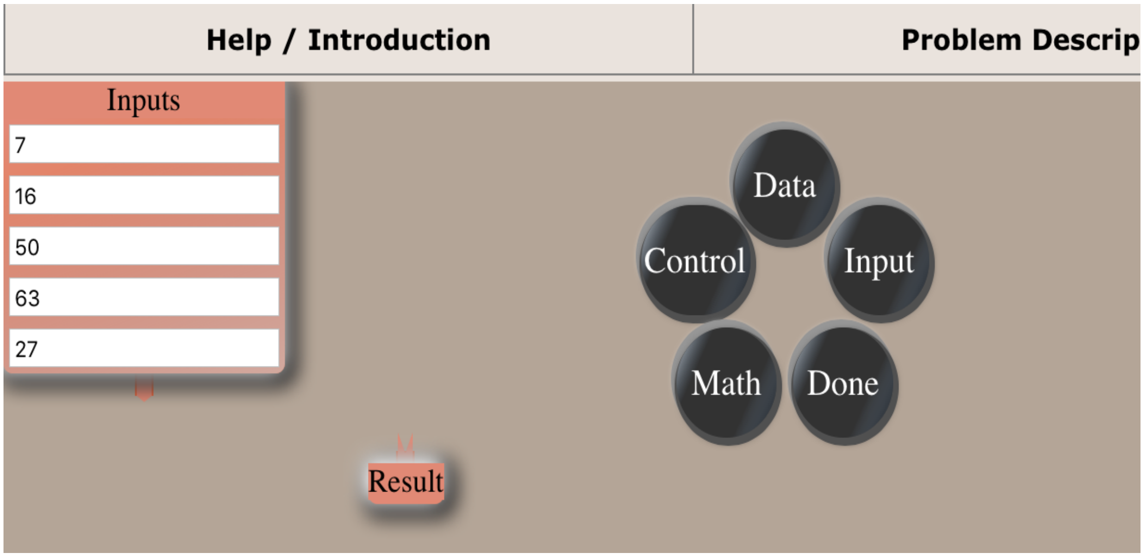
There are three tabs on this page:

1. Help/Introduction: Shows this documentation page.
2. Problem Description: Shows a problem that we would like you to program in the environment.
3. Programming Environment: This will put you in the environment that is described in this documentation page.

You can move to any of these tabs at any time.

Environment Start Page:

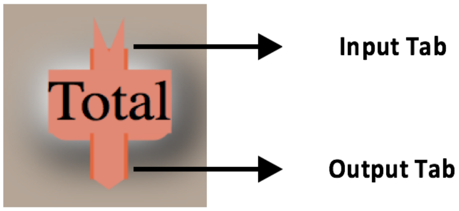
On clicking the "Programming Environment" link, the image below opens up:



In this environment, visual elements will be used to create a program. These elements can be pipes, input-output tables, and many different operators with their respective functions. They are connected to each other to produce a final result/program. Each of these elements are discussed in more detail below.

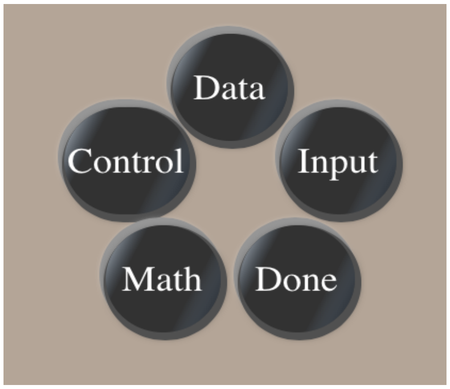
InTabs, OutTabs and Pipes

Blocks are 2-D rectangular shaped that have Input Tabs (to receive inputs) at the top and Output Tabs (to produce outputs) at the bottom.



Pipes are created by clicking on any block's Output Tab which is located at the bottom side of the block, and drag that pipe to the next block's Input Tab which is located at the top of the block. Pipes are used to link or connect one block to another. Data flows from one block to another block through pipes as shown in the examples below.

Bubble Menu:



Bubble menu shown in the above image consists of six sub-menus.

On clicking 'Input' menu, the sub-menu in figure 1 will open up.
On clicking 'Data' menu, the sub-menu in figure 2 will open up.
On clicking 'Control' menu, the sub-menu in figure 3 will open up.
On clicking 'Math' menu, the sub-menu in figure 4 will open up.
These menus will be discussed in more detail in the below sections.
'Done' menu is explained at the end of this documentation.

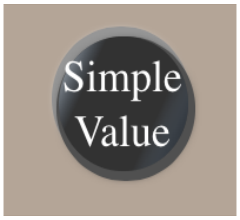


Figure 1: Input

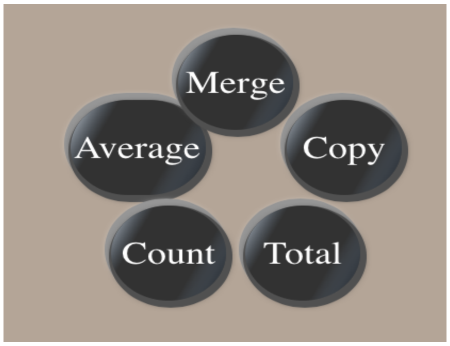


Figure 2: Data

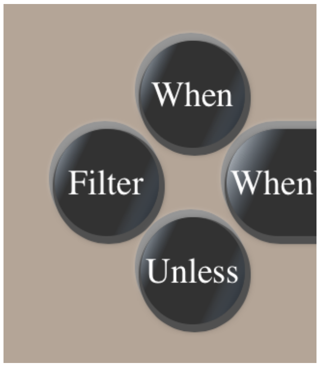


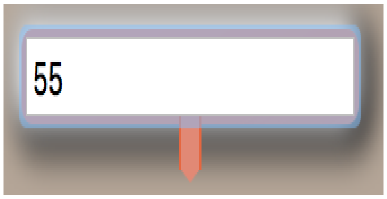
Figure 3: Con

I. INPUT

There are two types of Input Blocks:

▪ Simple Value Block

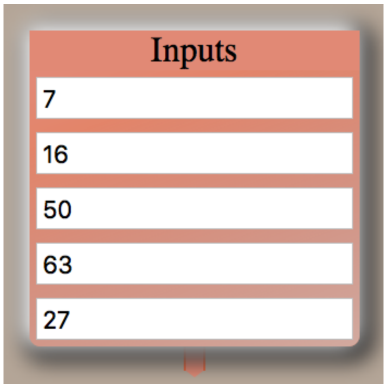
A Simple Value Block is an input block that allows a single entry value. This value can be of any of the primary types of data: it can be numeric, character, or a string. This block will be given to you in the environment according to the problem question requirements. You can create additional Simple Value Blocks from the menu if constant values are needed. An example for a simple value block is shown below:



55

▪ Input Table Block

An Input Table Block is an input block that allows multiple entry values. A table called 'Inputs' with values will be given to you in the environment according to the problem question requirements. You cannot add more values to it. An example for this block is shown in the 'Environment Start Page' figure above, where an Input table Block with values is given:



Inputs
7
16
50
63
27

II. **OUTPUT**

There are two types of Output Blocks:

▪ Result Block

A Result Block is an output block that allows a single result value. This value can be of any of the primary types of data: it can be numeric, character, or a string.

- **Result Table Block**

  A Result Table Block is an output block that allows multiple result values.

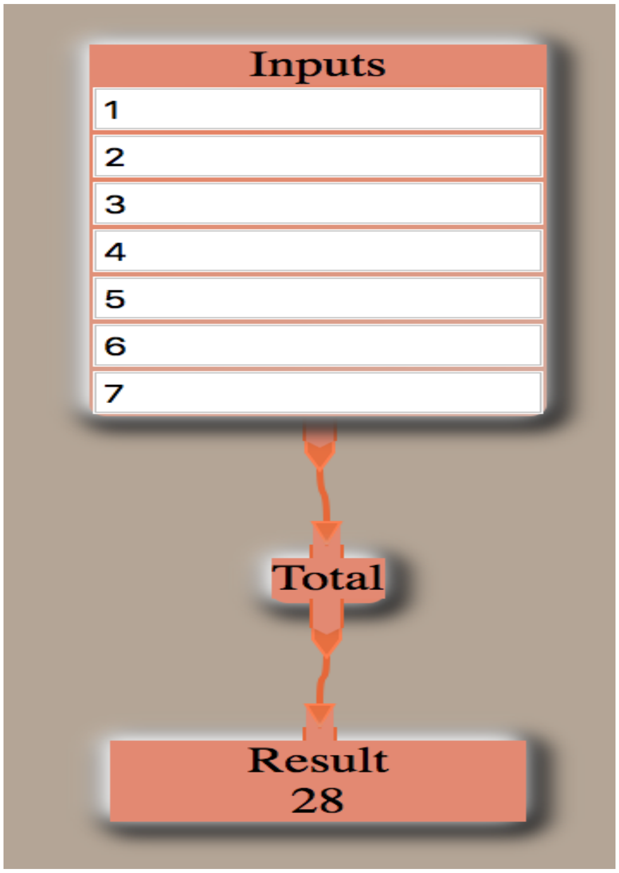
NOTE: An empty 'Result' or 'Result Table' Block will be given to you in the environment according to the problem question requirements. An image for an empty Result or Result Table Block is shown below:

III. **DATA**

Every 'Data' block has some activity or function associated to it. There are five types of blocks in this category:

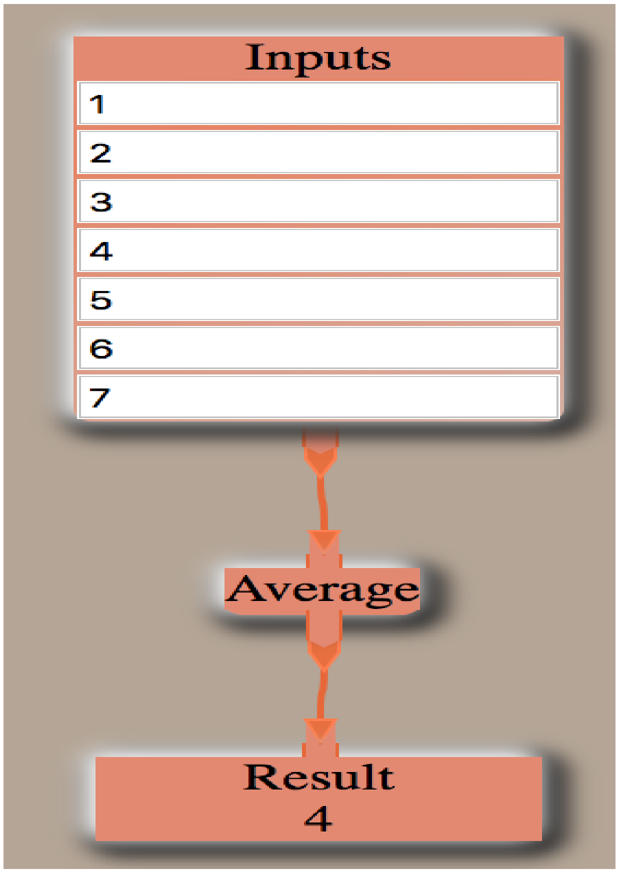
- **Total Block**

  A Total Block has a stream of values as its input, and a single numeric value that represents the summation of its input stream of values as an output. An example for this block is shown below:



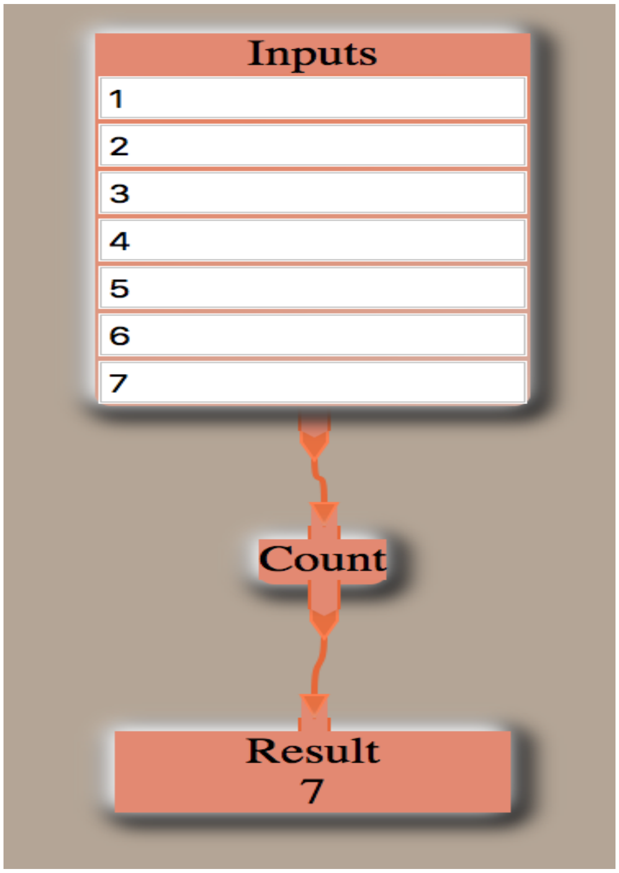
- **Average Block**

  An Average block calculates the average (mean) of its input stream of values and outputs the result as a single numeric value. An example for this block is shown below:



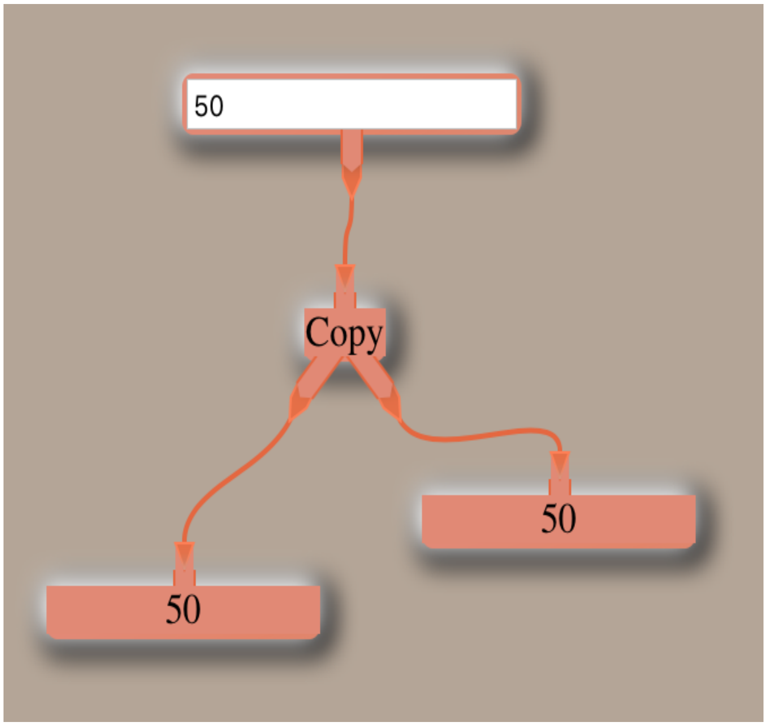
- **Count Block**

  A Count block calculates the number of elements in its input stream and outputs the result as a single numeric value. An example for this block is shown below:



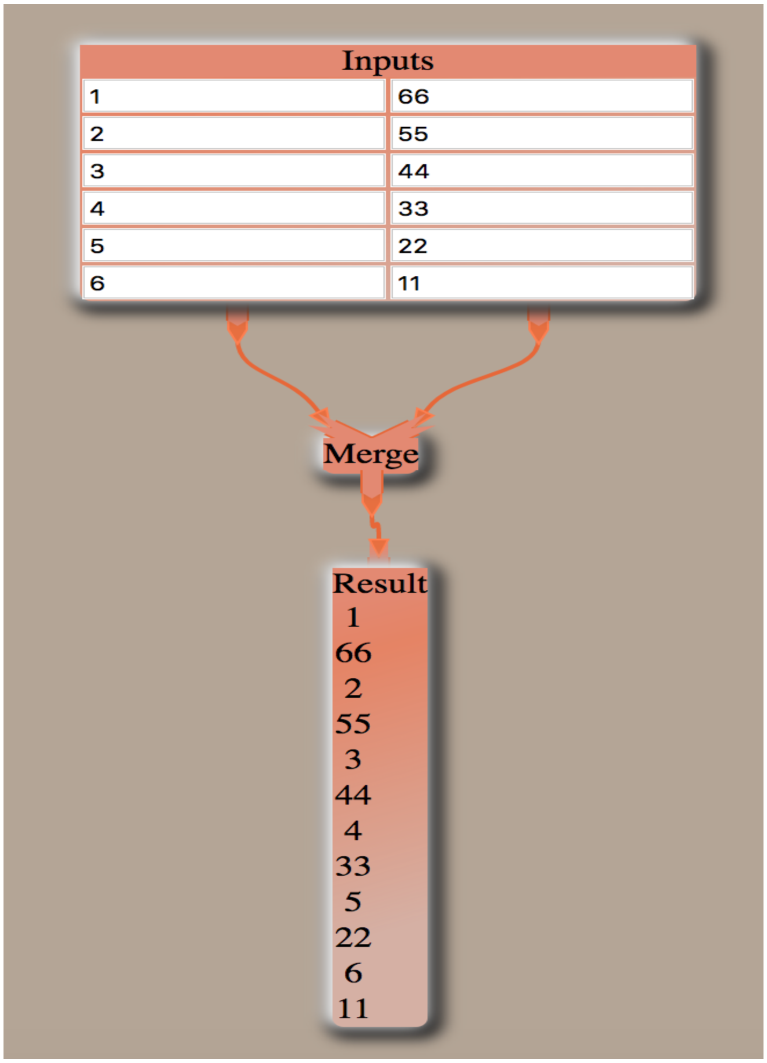
- **Copy Block**

  A Copy block will create multiple streams of any source whether it is a single value or an input table. An example for this block is shown below:



- **Merge Block**

  A Merge block combines the values of two source streams in one output stream considering mixing them up equally with respect to their order. An example for this block is shown below:



Inputs

1	66
2	55
3	44
4	33
5	22
6	11

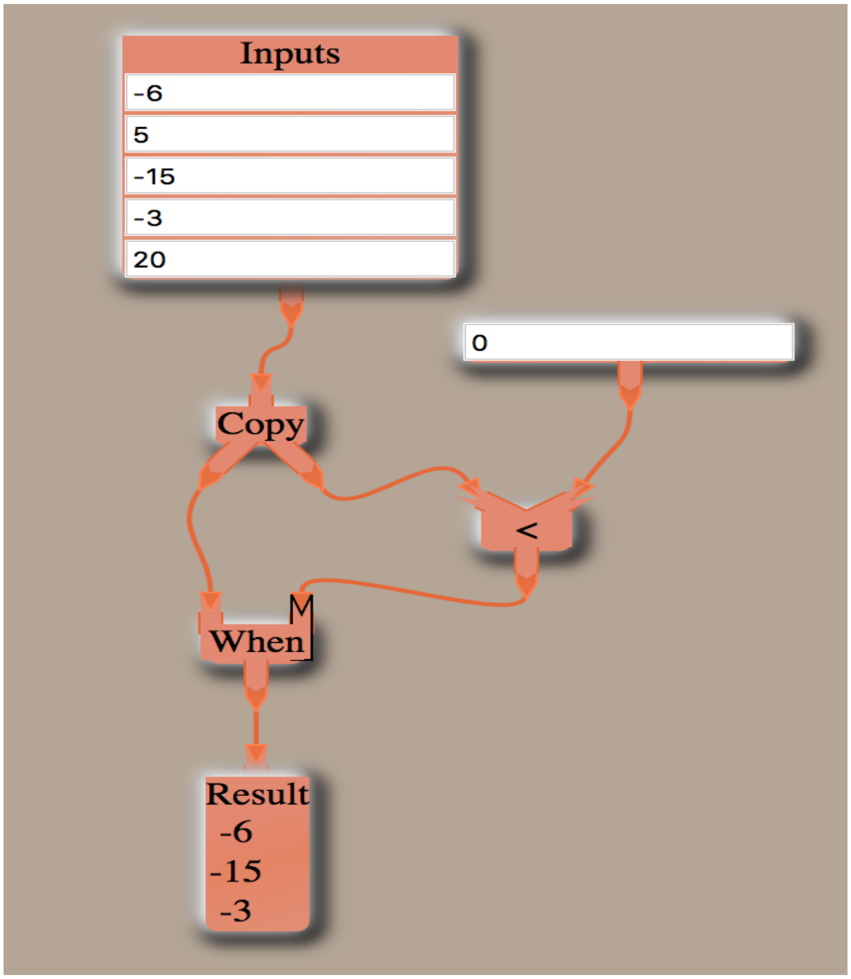
Result
1
66
2
55
3
44
4
33
5
22
6
11

IV. **CONTROL**

There are four types of blocks in this category:

- ### **When Block**

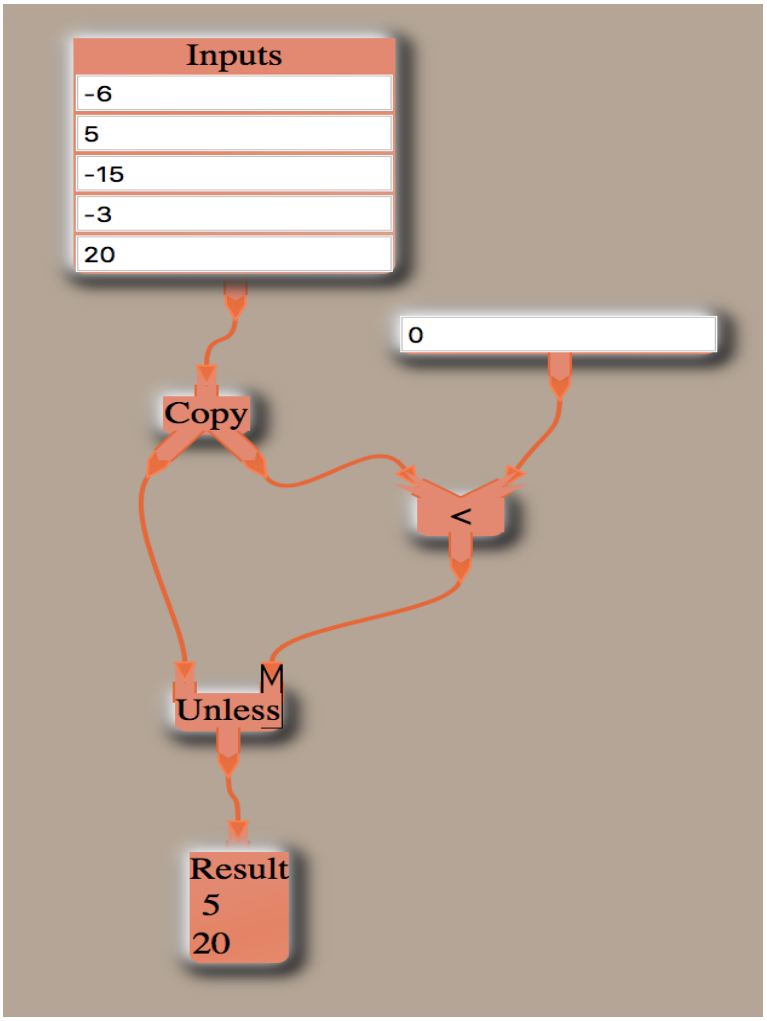
  A When Block is a conditional block. The output will be generated only if the input condition is satisfied. A When Block has two tabs: one is an 'Input Tab' and the other one is a 'Control Tab'. Input Tab will take in input stream of values and Control Tab will take care of the condition to be satisfied. An example for this block is shown below:



In this figure, the input stream of values are copied to the Result block **WHEN** they are less than zero.

- **Unless Block**

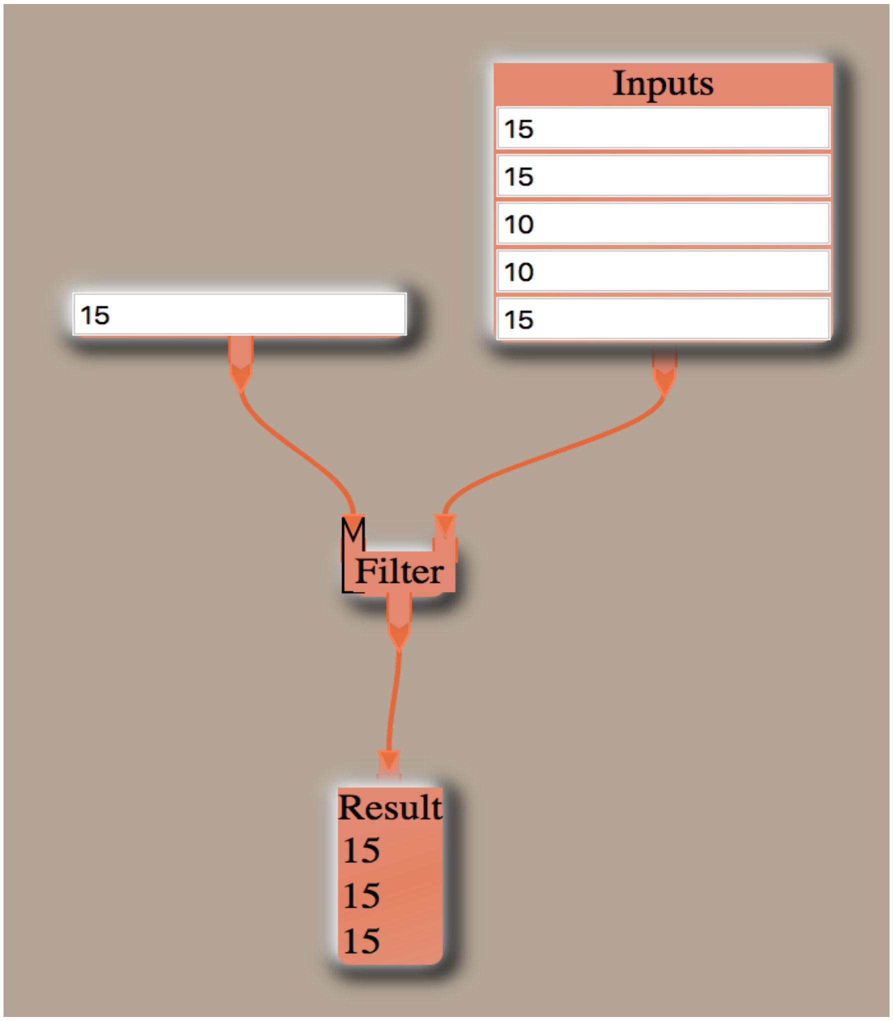
An Unless Block is a conditional block. The output will not be generated unless the input condition is false. This block has the same concept of an 'Input Tab' and a 'Control Tab' as discussed for the When Block. An example for this block is shown below:



In this figure, the input stream of values are copied to the Result Block **UNLESS** they are less than zero.

- **Filter Block**

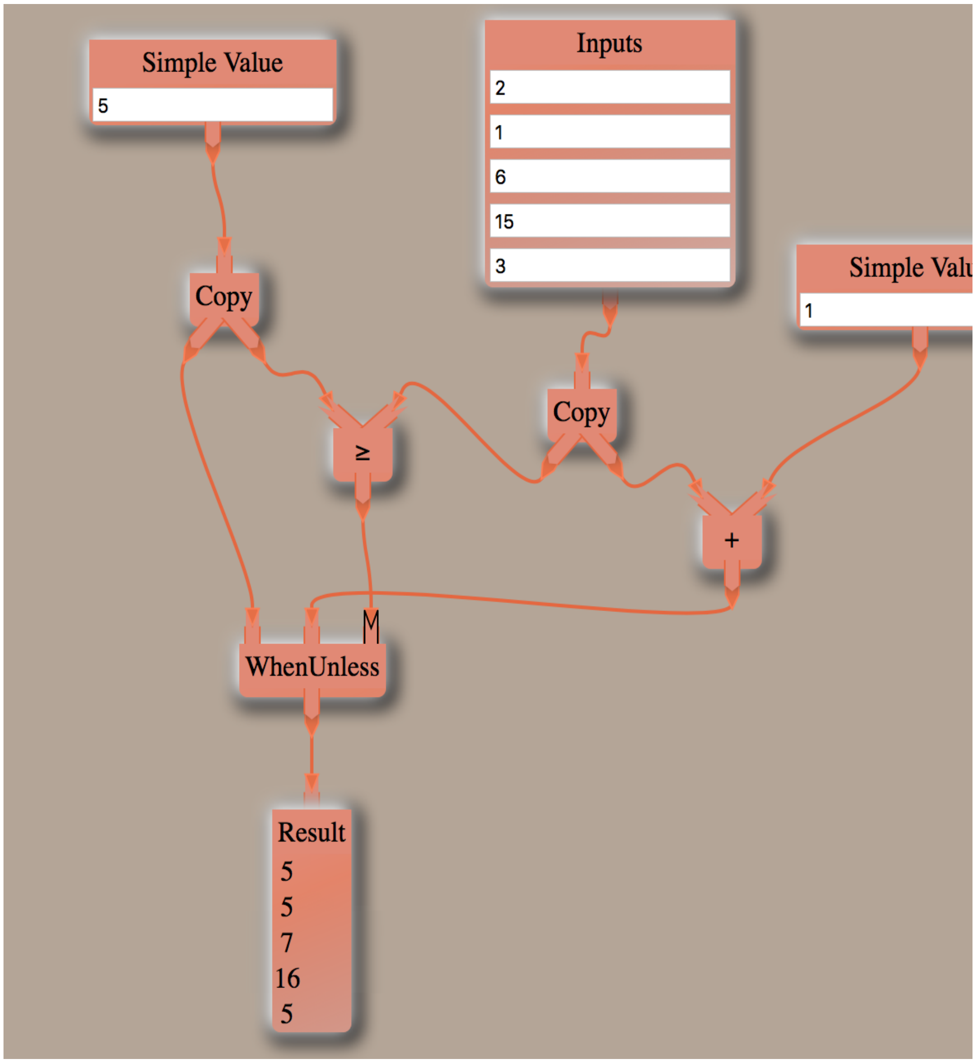
  A Filter Block is a control block that takes the first Control Tab as a filter key, and matches that key with its second Input Tab. If those two tabs match, the matching values will be outputted as a result. An example for this block is shown below:



- **WhenUnless Block**

  A WhenUnless Block is a combination of a When, Unless and a Merge Block. The output will be generated if the input condition is satisfied.

  A WhenUnless Block has two Input Tabs which take in its stream of values from its respective input blocks. The first Input Tab passes its steam of values to the output if the condition is true. The second Input Tab passes its steam of values to the output if the condition is false. WhenUnless Block has a Control Tab as its third tab which takes care of the condition to be satisfied. An example for this block is shown below:



In this figure, a Simple Value Block with the value 5 and an Input Table Block with a stream of values are compared with each other. A GreaterThanEqual block is the condition block connected to the Control Tab of WhenUnless block.

If the value in the Simple Value Block (5) is greater than or equal to the value coming in from the Input Table Block, the condition is true and the value in Simple Value Block will be outputted in the Result Block. If the condition is false, the value in the Input Table Block will be added to 1 and then outputted in the Result Block.

V. **MATH**

+	Adds two numbers. For example: 2 + 2 will return 4.
-	Subtracts one number from another number. For example: 5 - 2 will return 3.
÷	Divides two numbers. Gives the quotient as an answer For example: 6/2 will give a quotient which is 3.
×	Multiplies two numbers. For example: 5 x 10 will return 50.
mod	Divides two numbers. Gives the remainder as an answ For example: 6/2 will give a remainder which is 0.
>	Compares two numbers to check which is greater out For example: 3 > 4 will return false, 10 > 2 will return
<	Compares two numbers to check which is lesser out of For example: 3 < 4 will return true, 10 < 2 will return
≥	Compares two numbers to check if one number is grea For example: 2 >= 5 will return false, 7 >= 3 will retu
≤	Compares two numbers to check if one number is less For example: 2 <= 5 will return true, 7 <= 3 will retur
=	Compares two numbers to check if both the numbers For example: 10 = 10 will return true.

-

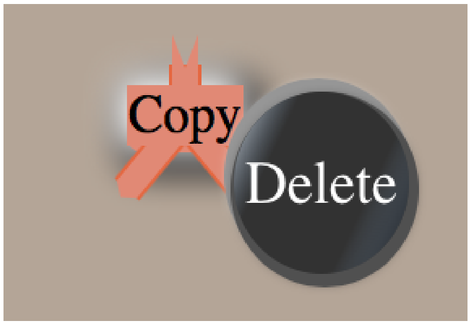
  Compares two numbers to check if both the numbers
  For example: 10 not = 4 will return true.

VI. DONE

On clicking this, you will be prompted asking if the answer/coding to the given problem question is done or not. Clicking on 'Ok' will lead to a short feedback form. Clicking on 'Cancel' will not do anything and keep you where you are, on the environment page.

VII. Delete Option

Deleting a block at any point of time can be done using the menu displayed in the below figure. This menu will open up on clicking any block. For example, a 'Simple Value' block can be deleted by clicking on the delete option from the menu:

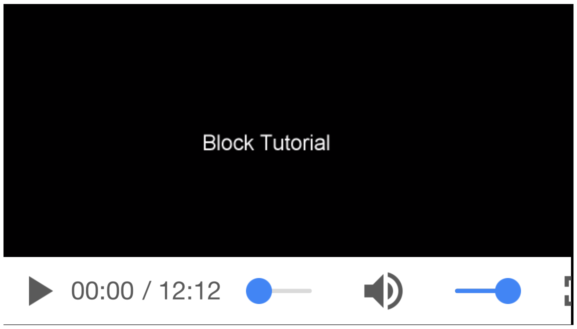


[Back To Top]

Appendix B

Help Documentation for Block-based Environment

Block Based Programming Language User Guide



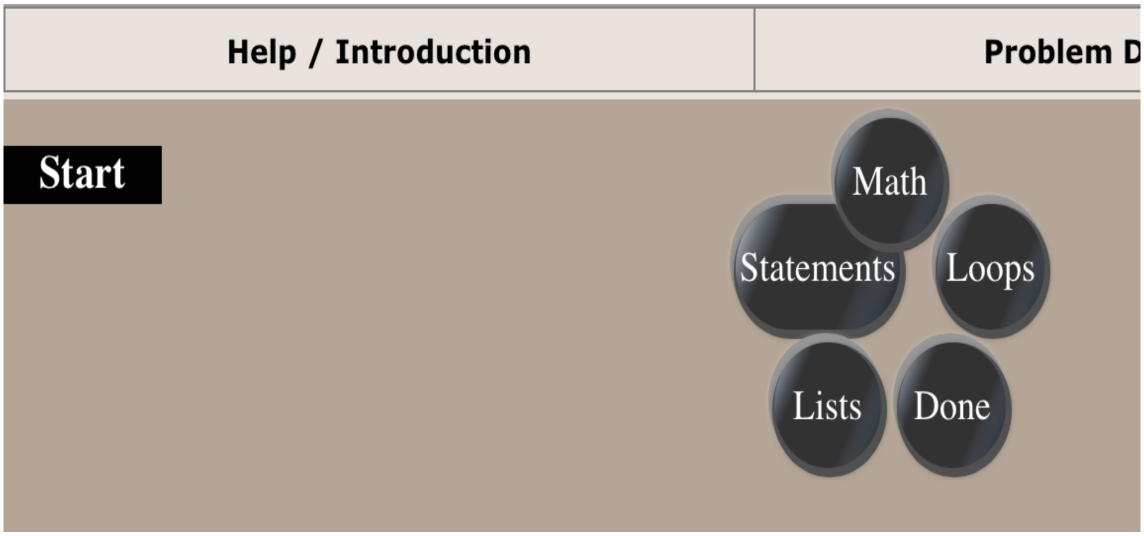
There are three tabs on this page:

1. Help/Introduction: Displays the current help documentation page.
2. Problem Description: Displays a problem question that needs to be solved on the programming environment.
3. Programming Environment: This will open the Block-based programming environment that is described in more detail in this help documentation page.

You can move to any of these tabs at any time.

Environment Start Page:

On clicking the "Programming Environment" link, the image below opens up:

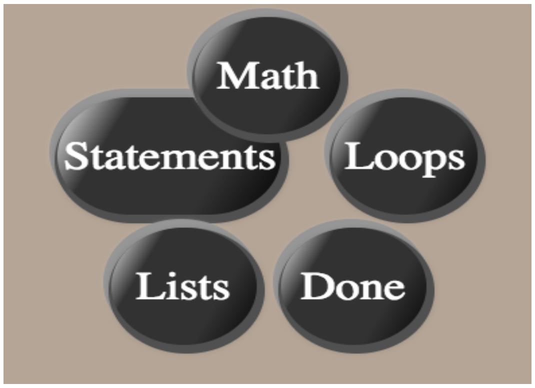


This page is divided into two parts:

1. Program Area:
   This area produces a 'bubble menu' on clicking anywhere on the page. You can select appropriate blocks from the different categories displayed within the bubble menu and organize the blocks in this area.
   You can then **drag and drop** your created program below the 'Start' button to produce the desired output.
   A program can either be created directly below the 'Start' button or it can be created outside and then be dropped below the 'Start' button.

2. Display Area:
   The output of the program created in the program area is shown in the display area.

Bubble Menu:



Bubble menu shown in the above image consists of five sub-menus.

On clicking 'Statements' menu, the sub-menu in figure 1 will open up.
On clicking 'Loops' menu, the sub-menu in figure 2 will open up.
On clicking 'List' menu, the sub-menu in figure 3 will open up.
On clicking 'Math' menu, the sub-menu in figure 4 will open up.
These menus will be discussed in more detail in the below sections.
'Done' menu is explained at the end of this documentation.

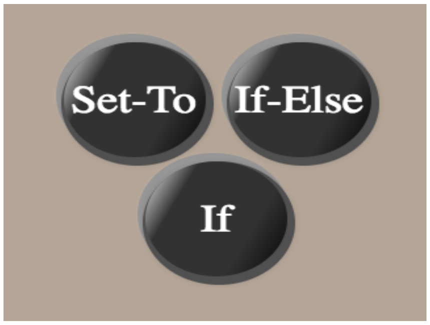


Figure 1: Statements

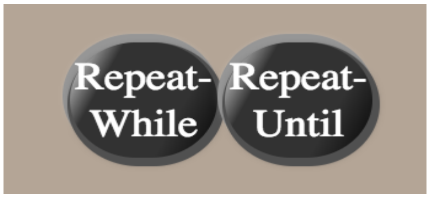


Figure 2: Loops

Fig

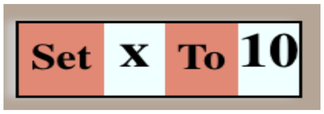
I. **STATEMENTS**

- ## Variables:

  A Variable can be understood as a location that holds different values at different times. You can change the value of the variable according to your needs.

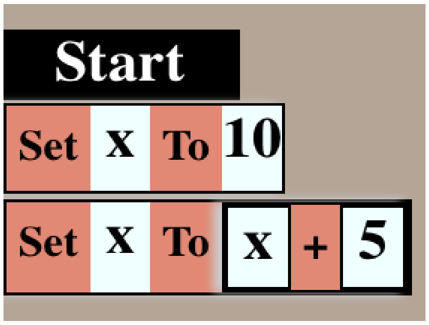
Let us understand this concept with few examples.

Assume 'Set-To' block assigns a value 10 to a variable 'x':



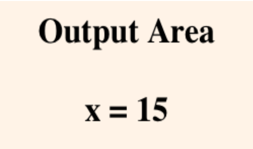
```
Set x To 10
```

This block is then connected with another 'Set-To' block to create a small program that manipulates the value of the variable 'x':



```
Start
Set x To 10
Set x To x + 5
```

The above code will change the value of the variable 'x' from 10 to 15. The display area will display only the final value of 'x'.
The output for this program is:



```
Output Area

x = 15
```

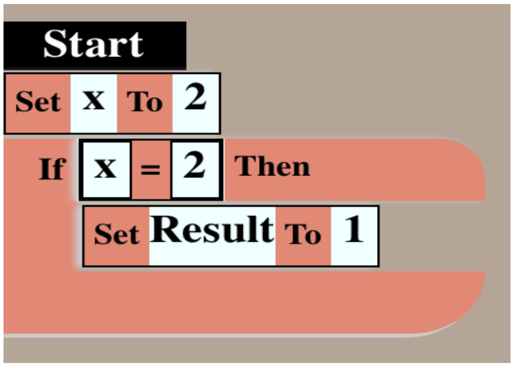
Note: If this problem question requires a single value as an output, a variable called 'Result' will already be displayed on the environment. You would just need to set an appropriate value to it.

- ## Conditional Statements:

Programs need to make decisions on a regular basis. This is the basic concept and an integral part of this and the next section - 'Loops'.
There are two types of conditional statement blocks here:

- **'If-Then' Block**
  Learning this block with an example will be easier. Hence an example for this block:

  Initially, set a variable 'x' to a certain number (x is set to 2 in this case).



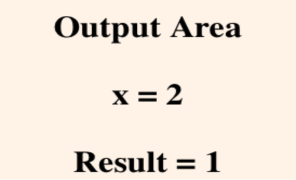
```
Start
Set x To 2
If x = 2 Then
    Set Result To 1
```

In the above figure, the block below 'Set-To' Block is the 'If-Then' Block. An 'If-Then' Block has the following form:

```
IF  (condition is true)
    [execute THEN body]
```

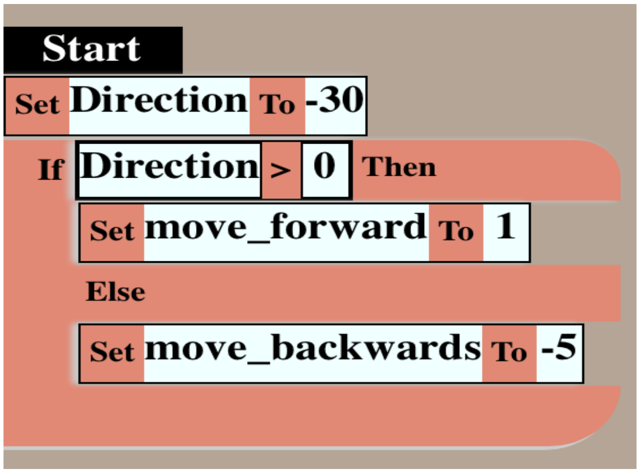
Here, the condition to check is if the two numbers are equal:
IF "x = 2" THEN "*Set 'Result' equal to 1*"
If the condition is true, the block within the "Then" body will be executed. Otherwise, the program will not execute.

The output for this program is:



```
Output Area

x = 2

Result = 1
```

- **'If-Then-Else' Block**
  An example for this block is as follows:



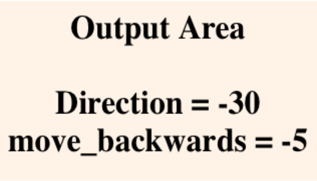
```
Start
Set Direction To -30
If Direction > 0 Then
    Set move_forward To 1
    Else
    Set move_backwards To -5
```

In the above figure, the block below 'Set-To' Block is the 'If-Then-Else' Block. An 'If-Then-Else' Block has the following form:

```
IF  (condition1 is true)
     [execute THEN body]
ELSE IF  (condition2 is true)
             [execute ELSE body]
```

In the above example, a variable called "Direction" is set to a particular value (-30). Here, the condition to check is if the value of Direction is greater than zero. If the condition is true, the block within the "Then" body will be executed, otherwise the block within the "Else" body will be executed.

In this case, if the value of direction is greater than zero, variable "move_forward" is set to 5 steps, and if the value is not greater than zero, variable "move_backwards" is set to -5 steps. The output for this program is:



```
Output Area

Direction = -30
move_backwards = -5
```

II. Loops

In computer programming, conditional loops are a way for computer programs to repeat one operation multiple number of times until a particular condition is satisfied. A conditional loop is controlled by a variable (with a number value) that counts up from an initial value to an upper limit. This variable is called as a 'loop control variable' or a 'counter variable'. They help in stopping the program from going in an infinite or a continuous loop. An error will be raised if the program goes into an infinite loop.

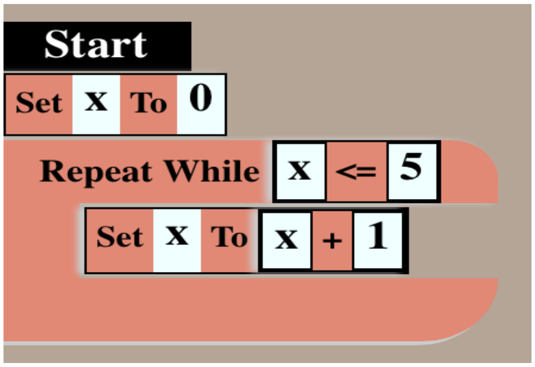
A loop has three parts that must be correct:

- The counter must be initialized.
- The test must end the loop on the correct count.
- The counter must be increased or decreased depending on the condition.

There are two types of conditional loop blocks:

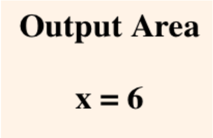
- **'Repeat-While' Block**
  In this example, the program initially sets the value of the variable 'x' to 0. The 'Repeat While' loop condition is that the value of 'x' should be lesser than or equal to 5.

  While the condition is true, the program will keep executing the statement within the repeat body increasing the value of 'x' by 1.
  'x' here is the 'counter variable' of this Repeat-While Block, initialized to the value '0' and incremented by 1.
  Once the condition **x <= 5** evaluates to false, the program will stop.



```
Start
Set x To 0
Repeat While x <= 5
    Set x To x + 1
```

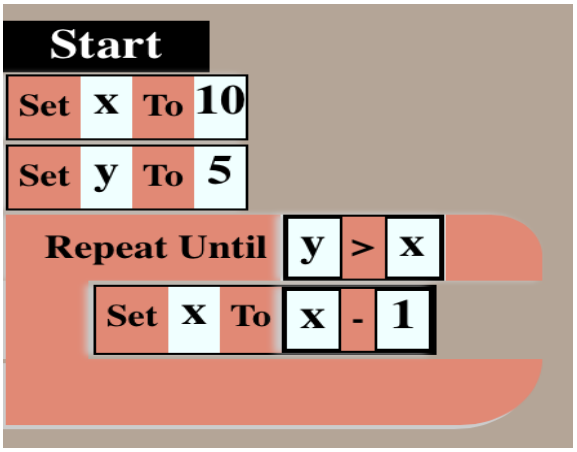
The output for this program is:



```
Output Area

x = 6
```

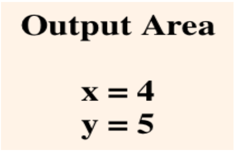
- **'Repeat-Until' Block**
  In this example, the program initially sets the value of the variable 'x' to 10 and variable 'y' to 5. The 'Repeat Until' loop condition is that the value of 'y' should be greater than the value of 'x'. Right now, the value of 'y' (5) is NOT greater than the value of 'x' (10).

  Hence, **Until the above condition is true**, the program will keep executing the statement within the repeat body decreasing the value of X by 1.
  'x' here is the 'counter variable' of this Repeat-Until Block, initialized to the value '10' and decremented by 1. Once the condition **y > x** evaluates to true, the program will stop.



```
Start
Set x To 10
Set y To 5
Repeat Until y > x
    Set x To x - 1
```

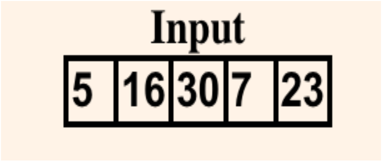
The output for this program is:



```
Output Area

x = 4
y = 5
```

III. **Input/Output List**

- **Input List**
  An input list will already be displayed on the environment and would contain values in it. No more values can be added to it. Values will need to be retrieved from the input list to solve a given question.
  For example, if the question was to find the sum of 5 numbers, an Input List with 5 numbers will be displayed in the display area section:



  Various functions need to be performed on this list to solve the question. They are described in more detail in the 'Lists' category.

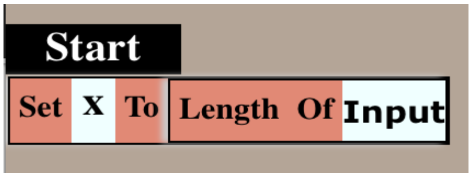
- **Result List**

The output list called 'Result' will already be displayed on the environment, if this problem question requires a series of values as its output. It will be an empty list and you can add values to it using the 'Add-To' list Block discussed in the 'Lists' category.

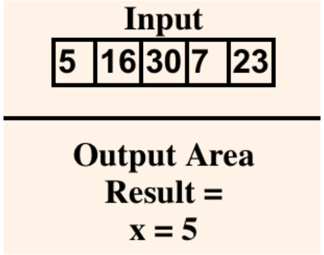
IV. **LISTS**

Lists contain collection of values. These values can be referenced by their positions in the list.
There are three types of list blocks in this category:

- **'Length-Of' List Block**
  This block will help in getting the size/length of the Input List. This block cannot be directly dropped below the 'Start' button. Before dropping, it needs to be attached to the 'Set-To' Block. An example for that:



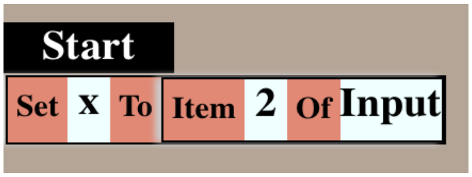
Since the size of the Input List is 5, x will be set to 5.



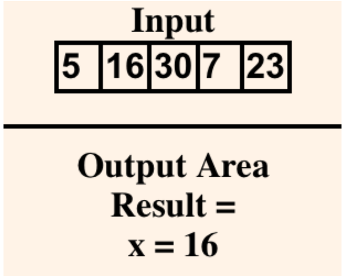
5	16	30	7	23

Output Area
Result =
x = 5

- **'Item-Of' List Block**
  This block will give the item at a particular position in the list. For example, if the item at position 2 is needed:



The display area would produce this answer:



Input

5	16	30	7	23

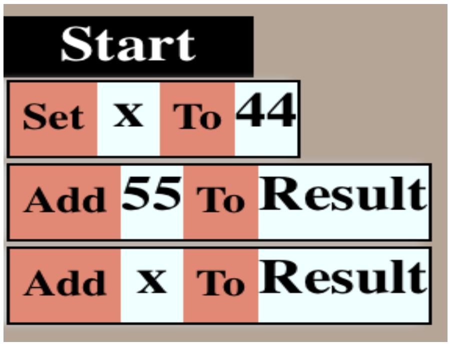
Output Area

Result =

x = 16

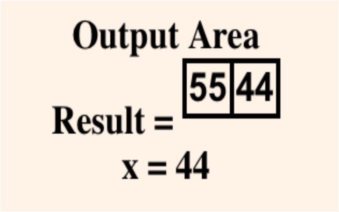
The item at the second position in the input list is 16.

- **'Add-To' List Block**
  An example of this block is shown below. Two numbers '55' and 'X' (with the value 44) are added to the given 'Result' list using 'Add-To' list block.



```
Start
Set x To 44
Add 55 To Result
Add x To Result
```

The 'Result' list in the 'Output Area' section of display area should look like this:



Output Area

Result =

55	44

x = 44

V. **MATH**

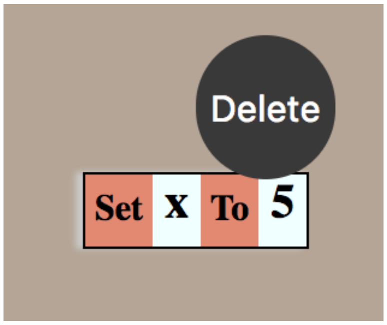
- + Adds two numbers.
  For example: 2 + 2 will return 4.
- - Subtracts one number from another number.
  For example: 5 - 2 will return 3.
- ÷ Divides two numbers. Gives the quotient as an answer
  For example: 6/2 will give a quotient which is 3.
- × Multiplies two numbers.
  For example: 5 x 10 will return 50.
- mod Divides two numbers. Gives the remainder as an answ
  For example: 6/2 will give a remainder which is 0.
- > Compares two numbers to check which is greater out
  For example: 3 > 4 will return false, 10 > 2 will return
- < Compares two numbers to check which is lesser out of
  For example: 3 < 4 will return true, 10 < 2 will return
- ≥ Compares two numbers to check if one number is grea
  For example: 2 >= 5 will return false, 7 >= 3 will retu
- ≤ Compares two numbers to check if one number is less
  For example: 2 <= 5 will return true, 7 <= 3 will retur
- = Compares two numbers to check if both the numbers
  For example: 10 = 10 will return true.
- ≠ Compares two numbers to check if both the numbers
  For example: 10 not = 4 will return true.

VI. **DONE**

On clicking this, you will be prompted asking if the answer/coding to the given problem question is done or not. Clicking on 'Ok' will lead to a short feedback form. Clicking on 'Cancel' will not do anything and keep you on the environment page.

VII. **Delete Option**

Deleting a block at any point of time can be done using this option. This option will open up on right clicking any block. For example, a 'Set-To' block can be deleted by right clicking on it and choosing the delete option:



[Back To Top]

Appendix C

Feedback Form

Please help us evaluate our environment by giving your valuable FEEDBACK!

Your Background Information

Your age:

Select one

Gender:

Select one

Country of Residence:

Select one

Education:

Select one

Experience with Programming:

Select one

Years of Experience with programming?

Select one

Your Experience with our Programming Environment

Programming with visual elements difficulty level:

Select one

Programming questions difficulty level:

Select one

Help / Introduction documentation difficulty level:

Select one

Was the environment consistent overall? (In terms of syntax, structure or anything else)

Select one

Was the environment easy to grasp?

Select one

Were the elements/components used to code easy to learn, remember and use?

Select one

Did the labels on the blocks make it easy to understand the block's designated function?

Select one

Was the visual layout of the environment comfortable enough to construct a program (connecting the blocks together to form a program)?

Select one

How easy was it to make a change in your program? For example, if you made a change in one part of the program, did it reflect in the other parts of the program as well and produce the output according to the new change?

Select one

Were the error notifications easy to understand? (These notifications pop up when the user makes an error such as dragging or dropping the block at the wrong place etc.)

Select one

Were you satisfied with the overall environment? Was it engaging enough?

Select one

Would you prefer to learn this (Visual Programming) style of coding language in comparison to learning a Text-based programming language like JAVA, C++ etc?

○Yes ○No

What problems did you encounter while programming?

Any suggestions on future improvements?

Submit

Bibliography

Alexandrova, S., Tatlock, Z., & Cakmak, M. (2015). Roboflow: A flow-based visual programming language for mobile manipulation tasks. In *2015 ieee international conference on robotics and automation (icra)*. Washington, DC, USA: IEEE Computer Society. doi:10.1109/icra.2015.7139973

Alice.org. (2007). Retrieved November 1, 2016, from http://www.alice.org/

Amazon Mechanical Turk. (2005). Retrieved May 1, 2017, from https://www.mturk.com/mturk/welcome

Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From scratch to "real" programming. *ACM Transactions on Computing Education (TOCE)*, *14*(4).

Bau, D. & Bau, A. D. (2014). A preview of pencil code. In *Proceedings of the 2nd workshop on programming for mobile & touch* (pp. 21–24). PROMOTO '14. Portland, Oregon, USA: ACM. doi:10.1145/2688471.2688481

Begel, A. & Klopfer, E. (2007). Starlogo tng: An introduction to game development. journal of e-learning.

Begel, A. & Resnick, M. (2000). Logoblocks: A graphical programming language for interacting with the world.

Blockly Google Developers. (2012). Retrieved May 17, 2017, from https://developers.google.com/blockly/

Bonar, J. G. & Liffick, B. W. (1987). Principles of visual programming systems. In S.-K. Chang (Ed.), (Chap. A Visual Programming Language for Novices). Prentice-Hall, Inc.

Bontà, P., Papert, A., & Silverman, B. (2010). Turtle, art, turtleart. In *Proc. of constructionism*. Paris, Fr.

Booth, T. & Stumpf, S. (2013). End-user experiences of visual and textual programming environments for arduino. *End-User Development Lecture Notes in Computer Science*, 25–39. doi:10.1007/978-3-642-38706-7_4

Borning, A. (1981). The programming language aspects of thinglab, a constraint-oriented simulation laboratory. *ACM Transactions on Programming Languages and Systems*, *3*(4), 353–387. doi:10.1145/357146.357147

Boshernitsan, M. & Downes, S. M. (2004). Visual programming languages: A survey. *Citeseer*.

Brown, N. C., Mönig, J., Bau, A., & Weintrop, D. (2016). Panel: Future directions of block-based programming. In *Proceedings of the 47th acm technical symposium on computing science education* (pp. 315–316). SIGCSE '16. Memphis, Tennessee, USA: ACM. doi:10.1145/2839509.2844661

Burnett, M. M. (1995). Visual object-oriented programming. (Chap. Seven programming language issues, pp. 161–181). Greenwich, CT, USA: Manning Publications Co.

Carroll, M. J., Thomas, C. J., & Malhotra, A. (1980). Presentation and representation in design problem-solving. *British Journal of Psychology*, *71*(1), 143–153. doi:10.1111/j.2044-8295.1980.tb02740.x

Catetè, V., Wassell, K., & Barnes, T. (2014). Use and development of entertainment technologies in after school stem program. In *Proceedings of the 45th acm technical symposium on computer science education* (pp. 163–168). SIGCSE 14. Atlanta, Georgia, USA: ACM. doi:10.1145/2538862.2538952

Cheung, C. J., Ngai, G., Chan, C. S., & Lau, W. W. (2009). Filling the gap in programming instruction. *ACM SIGCSE Bulletin*, *41*(1), 276. doi:10.1145/1539024.1508968

Communicating sequential processes. (1977). Retrieved April 19, 2017, from https://en.wikipedia.org/wiki/Communicating_sequential_processes#cite_note-hoare1978-5

Cunniff, N. & Taylor, R. P. (1987). Empirical studies of programmers: Second workshop. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), (Chap. Graphical vs. Textual Representation: An Empirical Study of Novices' Program Comprehension, pp. 114–131). Norwood, NJ, USA: Ablex Publishing Corp. Retrieved from http://dl.acm.org/citation.cfm-id=54968.54976

Curtis, B., Sheppard, B. S., Kruesi-Bailey, E., Bailey, J., & Boehm-Davis, A. D. (1989). Experimental evaluation of software documentation formats. *Journal of Systems and Software*, *9*(2), 167–207. doi:10.1016/0164-1212(89)90019-8

Dann, W., Cosgrove, D., Slater, D., Culyba, D., & Cooper, S. (2012). Mediated transfer. In *Sigcse proceedings of the 43rd acm technical symposium on computer science education* (pp. 141–146). Raleigh, North Carolina, USA: ACM. doi:10.1145/2157136.2157180

Data visualization & presentation tool. (2005). Retrieved October 4, 2016, from http://www.quadrigram.com/

Day, S. R. (1988). The psychology of learning and motivation. (Chap. Alternative representations, pp. 261–305). San Diego, CA: Academic Press.

Erwig & Meyer, B. (1995). Heterogeneous visual languages: Integrating visual and textual programming. In *Proceedings of the 11th international ieee symposium on visual languages* (pp. 318–325). VL '95. Darmstadt, Germany: IEEE Computer Society. doi:10.1109/VL.1995.520825

Experimental Design Definition and Examples. (2017). Retrieved June 1, 2017, from http://www.statisticshowto.com/experimental-design/

Factorial Designs. (2006). Retrieved June 1, 2017, from https://www.socialresearchmethods.net/kb/expfact.php

Finzer, W. F. & Gould, L. (1993). Watch what i do. In A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, & A. Turransky (Eds.), (Chap. Rehearsal World: Programming by Rehearsal, pp. 79–100). Cambridge, MA, USA: MIT Press. Retrieved from http://dl.acm.org/citation.cfm-id=168080.168098

Fix, V., Wiedenbeck, S., & Scholtz, J. (1993). Mental representations of programs by novices and experts. In *Proceedings of the interact'93 and chi'93: Conference on human factors* *in computing systems* (pp. 74–79). Amsterdam, The Netherlands: ACM. doi:10.1145/169059.169088

Flowchart. (1921). Retrieved March 4, 2017, from https://en.wikipedia.org/wiki/Flowchart

Flowgorithm - Flowchart Programming Language. (2014). Retrieved April 4, 2017, from http://www.flowgorithm.org/

Green, T., Petre, M., & Bellamy, R. (1991). Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture.

Green & Petre, M. (1996). Usability analysis of visual programming environments: A 'cognitive dimensions' framework. *Journal of Visual Languages & Computing*, *7*, 131–174. doi:10.1006/jvlc.1996.0009

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, *21*(8), 666–677. doi:10.1145/359576.359585

Hoare, C. A. R. (1985). *Communicating sequential processes.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc.

Hypothesis Testing (P-value approach). (2017). Retrieved October 3, 2017, from https://onlinecourses.science.psu.edu/statprogram/node/138

M Homer, J. N. (2014). Combining tiled and textual views of code. In *Proceedings of the 2014 second ieee working conference on software visualization* (pp. 1–10). VISSOFT '14. Washington, DC, USA: IEEE Computer Society. doi:10.1109/vissoft.2014.11

M Kölling, A. A., C N Brown. (2015). Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the workshop in primary and secondary computing education on zzz* (pp. 29–38). WiPSCE 15. London, United Kingdom: ACM. doi:10.1145/2818314.2818331

Ma, L., Ferguson, J., Roper, M., Ross, I., & Wood, M. (2009). Improving the mental models held by novice programmers using cognitive conflict and jeliot visualisations. In *Proceedings of the 14th annual acm sigcse conference on innovation and technology in computer science education* (pp. 166–170). ITiCSE '09. Paris, France: ACM Press. doi:10.1145/1562877.1562931

Ma, L., Ferguson, J., Roper, M., & Wood, M. (2007). Investigating the viability of mental models held by novice programmers. In *Proceedings of the 38th sigcse technical symposium on computer science education* (pp. 499–503). SIGCSE '07. Covington, Kentucky, USA: ACM Press. doi:10.1145/1227310.1227481

Malan, D. J. & Leitner, H. H. (2007). Scratch for budding computer scientists. In *Proceedings of the 38th sigcse technical symposium on computer science education* (pp. 223–227). SIGCSE '07. Covington, Kentucky, USA: ACM Press. doi:10.1145/1227310.1227388

Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with scratch. In *Proceedings of the 39th sigcse technical symposium on computer science education* (pp. 367–371). SIGCSE '08. Portland, OR, USA: ACM Press. doi:10.1145/1352135.1352260

Mann-Whitney U test. (2017). Retrieved October 3, 2017, from https://en.wikipedia.org/wiki/Mann-Whitney_U_test

Mason, D. (2017). The kit language. Retrieved August 10, 2017, from http://programmingfortherestofus.com/kitPJS/

Matsuzawa, Y., Ohata, T., Sugiura, M., & Sakai, S. (2015). Language migration in non-cs introductory programming through mutual language translation environment. In *Proceedings of the 46th acm technical symposium on computer science education* (pp. 185–190). SIGCSE '15. Kansas City, Missouri, USA: ACM Press. doi:10.1145/2676723.2677230

McKay, F. & Kölling, M. (2013). Predictive modelling for hci problems in novice program editors. In *Proceeding bcs-hci '13 proceedings of the 27th international bcs human computer interaction conference*. London, UK.

MIT App Inventor. (2015). Retrieved December 9, 2016, from http://appinventor.mit.edu/

Morrison, P. J. (1970). Retrieved June 18, 2017, from http://jpaulmorrison.com/fbp/

Morrison, P. J. (1994). Flow-based programming. *Journal of Application Developers' News*.

Morrison, P. J. (2005). Patterns in flow-based programming. Retrieved June 18, 2017, from http://www.jpaulmorrison.com/fbp/morrison_2005.htm

Morrison, P. J. (2012). Flow-based programming. Retrieved June 18, 2017, from https://flowbasedprogramming.wordpress.com/article/flow-based-programming/

Myers, B. A. (1987). Creating interaction techniques by demonstration. *IEEE Computer Graphics and Applications*, *7*(9), 51–60. doi:10.1109/mcg.1987.277079

Myers, B. A. (1988). *Creating user interfaces by demonstration.* San Diego, CA, USA: Academic Press Professional.

Myers, B. A. (1990). Creating user interfaces using programming by example, visual programming, and constraints. *ACM Transactions on Programming Languages and Systems*, *12*(2), 143–177. doi:10.1145/78942.78943

Papadakis, S., Kalogiannakis, M., Orfanakis, V., & Zaranis, N. (2014). Novice programming environments. scratch & app inventor. In *Proceedings of the 2014 workshop on interaction design in educational environments* (p. 1). IDEE '14. Albacete, Spain: ACM Press. doi:10.1145/2643604.2643613

Pd Community Site. (1990). Retrieved November 1, 2016, from http://www.puredata.org/

Polich, M. J. & Schwartz, H. S. (1974). The effect of problem size on representation in deductive problem solving. *Memory & Cognition*, *2*, 683–686. doi:10.3758/BF03198139

Price, T. W. & Barnes, T. (2015). Comparing textual and block interfaces in a novice programming environment. In *Proceedings of the eleventh annual international conference on international computing education research* (pp. 91–99). ICER '15. Omaha, Nebraska, USA: ACM Press. doi:10.1145/2787622.2787712

Pseudocode. (2003). Retrieved July 15, 2017, from https://en.wikipedia.org/wiki/Pseudocode

Pure Data. (1990). Retrieved October 5, 2016, from http://write.flossmanuals.net/pure-data/introduction2/

Python Software Foundation. (2001). Retrieved July 1, 2017, from https://www.python.org/

Ramsey, H. R., Atwood, M. E., & Doren, J. R. V. (1983). Flowcharts versus program design languages: An experimental comparison. *Communications of the ACM*, *26*(6), 445–449. doi:10.1145/358141.358149

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., ... Kafai, Y. (2009). Scratch: Programming for all. *Commun. ACM*, *52*(11), 60–67. doi:10.1145/1592761.1592779

Robinson, W. (2016). From scratch to patch. In *Proceedings of the 11th workshop in primary and secondary computing education on zzz* (pp. 96–99). WiPSCE '16. Münster, Germany: ACM Press. doi:10.1145/2978249.2978265

Scanlan, D. A. (1989). Structured flowcharts outperform pseudocode: An experimental comparison. *IEEE Software*, *6*(5), 28–36. doi:10.1109/52.35587

Schwartz. (1971). Modes of representation and problem solving: Well evolved is half solved. *Journal of Experimental Psychology*, *91*, 347–350. doi:10.1037/h0031856

Schwartz & Fattaleh, D. (1972). Representation in deductive problem-solving: The matrix. *Journal of Experimental Psychology*, *95*, 343–348. doi:10.1037/h0033669

Scratch - Imagine, Program, Share. (2007). Retrieved December 1, 2016, from http://scratch.mit.edu/

Sharafi, Z., Marchetto, A., Susi, A., Antoniol, G., & Guéhéneuc, Y.-G. (2013). An empirical study on the efficiency of graphical vs. textual representations in requirements comprehension. In *Icpc* (pp. 33–42). IEEE Computer Society.

Shu, N. C. (1999). Visual programming: Perspectives and approaches. *IBM Systems Journal*, *38*, 199–221. doi:10.1147/sj.382.0199

Smith. (1975). *Pygmalion: A creative programming environment.* Memo (Stanford Artificial Intelligence Laboratory). Defense Technical Information Center. Retrieved from https://books.google.ca/books-id=58U-AAAAIAAJ

Smith. (1986). The alternate reality kit: An animated environment for creating interactive simulations. In *Proceedings of 1986 ieee computer society workshop on visual languages* (pp. 99–106). Washington, DC, USA: IEEE Computer Society.

Snap! (Build Your Own Blocks) 4.0. (2011). Retrieved January 1, 2017, from http://byob.berkeley.edu/

Stein, D. & Hanenberg, S. (2011). Comparison of a visual and a textual notation to express data constraints in aspect-oriented join point selections: A controlled experiment. In *Proceedings of the 2011 ieee 19th international conference on program comprehension* (pp. 141–150). ICPC '11. Washington, DC, USA: IEEE Computer Society. doi:10.1109/ICPC.2011.9

Teasley, B. E. (1994). The effects of naming style and expertise on program comprehension. *International Journal of Human-Computer Studies*, *40*, 757–770. doi:10.1006/ijhc.1994.1036

Techapalokul, P. & Tilevich, E. (2015). Programming environments for blocks need first-class software refactoring support: A position paper. In *Proceedings of the 2015 ieee blocks and beyond workshop (blocks and beyond)* (pp. 109–111). BLOCKS AND BEYOND '15. Washington, DC, USA: IEEE Computer Society. doi:10.1109/BLOCKS.2015.7369015

Vasek, M. (2012). *Representing expressive types in blocks programming languages.* Wellesley College.

Wagner, A., Gray, J., Corley, J., & Wolber, D. (2013). Using app inventor in a k-12 summer camp. In *Proceeding of the 44th acm technical symposium on computer science education* (pp. 621–626). SIGCSE '13. Denver, Colorado, USA: ACM Press. doi:10.1145/2445196.2445377

Waterbear. (2011). Retrieved March 1, 2017, from http://waterbearlang.com/

Weerasinghe, P. K. & Cohen, M. (2012). Rhythm of music animating virtual environment models. In *Proceedings of the 2012 joint international conference on human-centered computer environments* (pp. 200–202). HCCE '12. Aizu-Wakamatsu, Japan: ACM Press. doi:10.1145/2160749.2160790

Weintrop, D. (2015a). Comparing text-based, blocks-based, and hybrid blocks/text programming tools. In *Proceedings of the eleventh annual international conference on international computing education research* (pp. 283–284). ICER '15. Omaha, Nebraska, USA: ACM Press. doi:10.1145/2787622.2787752

Weintrop, D. (2015b). Minding the gap between blocks-based and text-based programming (abstract only). In *Proceedings of the 46th acm technical symposium on computer science education* (pp. 720–720). SIGCSE '15. Kansas City, Missouri, USA: ACM Press. doi:10.1145/2676723.2693622

Weintrop, D. (2016). Bringing blocks-based programming into high school computer science classrooms.

Weintrop, D. & Wilensky, U. (2012). Robobuilder: A program-to-play constructionist video game. In *In*.

Weintrop, D. & Wilensky, U. (2015a). To block or not to block, that is the question. In *Proceedings of the 14th international conference on interaction design and children* (pp. 199–208). IDC '15. Boston, Massachusetts: ACM Press. doi:10.1145/2771839.2771860

Weintrop, D. & Wilensky, U. (2015b). Using commutative assessments to compare conceptual understanding in blocks-based and text-based programs. In *Proceedings of the eleventh annual international conference on international computing education research* (pp. 101–110). ICER '15. Omaha, Nebraska, USA: ACM Press. doi:10.1145/2787622.2787721

What is Block Based Coding? (2014). Retrieved July 1, 2017, from https://blog.penjee.com/what-is-block-based-coding/

Whitley, K. (1997). Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages & Computing*, *8*, 109–142. doi:10.1006/jvlc.1996.0030

Whitley, Novick, L. R., & Fisher, D. (2006). Evidence in favor of visual representation for the dataflow paradigm: An experiment testing labview's comprehensibility. *Int. J. Hum.-Comput. Stud. 64*(4), 281–303. doi:10.1016/j.ijhcs.2005.06.005

Wolber, D., Abelson, H., & Friedman, M. (2015). Democratizing computing with app inventor. *GetMobile: Mobile Computing and Communications*, *18*, 53–58. doi:10.1145/2721914.2721935

Wright, P. & Reid, F. (1973). Written information: Some alternatives to prose for expressing the outcomes of complex contingencies. *Journal of Applied Psychology*, *57*, 160–166. doi:10.1037/h0037045

Xie, B., Shabir, I., & Abelson, H. (2015). Measuring the usability and capability of app inventor to create mobile applications. In *Proceedings of the 3rd international workshop on programming for mobile and touch* (pp. 1–8). PROMOTO 2015. Pittsburgh, PA, USA: ACM Press. doi:10.1145/2824823.2824824