Visual Programming Language for Tacit Subset of J Programming Language

Nouman Tariq

Dissertation 2013

Erasmus Mundus MSc in Dependable Software Systems

NUI MAYNOOTH

Ollscoil na hÉireann Má Nuad

Department of Computer Science

National University of Ireland, Maynooth

Co. Kildare, Ireland

A dissertation submitted in partial fulfilment

of the requirements for the

Erasmus Mundus MSc Dependable Software Systems

Head of Department : Dr Adam Winstanley

Supervisor : Professor Ronan Reilly

June 30, 2013

Declaration

I hereby certify that this material, which I now submit for assessment on the program of study leading to the award of Master of Science in Dependable Software Systems, is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

Signed:_____ Date:_____

Abstract

Visual programming is the idea of using graphical icons to create programs. I take a look at available solutions and challenges facing visual languages. Keeping these challenges in mind, I measure the suitability of Blockly and highlight the advantages of using Blockly for creating a visual programming environment for the J programming language. Blockly is an open source general purpose visual programming language designed by Google which provides a wide range of features and is meant to be customized to the user's needs. I also discuss features of the J programming language that make it suitable for use in visual programming language. The result is a visual programming environment for the tacit subset of the J programming language.

Table of Contents

List of Figures

Introduction

The idea of Visual Programming Environments (VPEs) stems from the common proverb that "A picture is worth thousand words". Generally speaking the concept of VPE can be defined as "use of meaningful graphic representations in the process of programming" [1]. VPEs provide users with graphical icons and notations which can be combined together to create computer programs. This pictorial approach to programming helps users in visualizing the data and/or control flow of the program as well as removing the burden of learning syntax.

In this dissertation I apply the idea of a visual programming environment to the J programming language. J is an array-based, general purpose programming language. Its design and features make it ideal for mathematical, statistical and logical analysis of data. J provides function-level programming through its so-called tacit subset. In this project the entire focus is on this tacit subset. We get a comprehensive, demonstrable and useable VPE by implementing all the features in the tacit subset of J.

Problem Statement

According to [2], one of the primary reasons text based programming languages dominate computer programming is the greater flexibility and rich expressions that they provide. Another challenge facing visual programming is to breach the natural language and cultural boundaries through the use of icons which may have different meanings in different cultures. While words have consistent meaning in natural language, icons have different meanings in different cultures. Even if such a vocabulary of icons was to be created by some means, there would still be the burden of learning this vocabulary to be able to write and understand an already written code using the designed visual programming language.

The basic premise of this work is that functional programming languages are better suited for Visual Programming than object-oriented or imperative programming languages. Also, with J's terseness we should be able to solve any problem that can be solved through J through the developed VPE. Through this work I discuss the suitability of the tacit subset of the J programming language to be adapted for visual programming. An attempt is made to make use of a general purpose visual programming environment which when adapted for J should provide flexibility of expression. Visual programming adds a layer of abstraction to the underlying programming language e.g. the visual language may have one construct for loop but the underlying programming language would provide for, while and do-while loops thus resulting in a layer of abstraction. This abstraction should not hinder the programmer in expressing their solution and should form a basic design principle of the visual programming language being designed.

Motivation

J provides a terse syntax for programming as compared to many of the modern and widely used programming languages which are much more verbose and support an imperative

programming style. This terseness of syntax, combined with the tacit programming paradigm, makes it difficult to learn J. The developed tool reduces the learning burden.

With many of the modern object-oriented programming languages like C++ and Java, it is very easy for a programmer versed in one of the languages to understand much of the code written in the other language. This ease stems from the fact that these languages are much more verbose and are developed on the common principles of OOP. Without a deep understanding of the J programming language, it is very difficult to understand the control and/or data flow of a J program. The developed tool provides a much easier way to understand and visualise the control and data flow throughout the program. Glinert [3] proposes that one of the advantages to visual programming is its ability to preserve (and even enhance) parallelization in programming. Thus a VPE for J would have an added effect of allowing us to visualise what parts of the program can be executed in parallel. This understanding would help developers to take advantage of the multi core feature of modern computers and write more efficient programs.

This leads into another problem area with visual programming: the syntax. There has been a lot of work done on defining visual programming languages. [4] argues that the visual programming languages are tightly coupled with the corresponding visual programming environments and provides a possible solution for this dilemma. [5] and [6] define ways for efficient parsing of the visual programming languages such that these languages could be used more generically and not be as tightly coupled with the underlying VPE and more importantly be expressive enough to be useable for general-purpose programming.

Aims and Objectives

The resulting solution should be useable by a novice programmer. It should provide a way to set the focus on problem solving and provide a bridge towards learning J eventually. There is appeal in this tool for expert J programmers as well. The visual representation should help them visualize the solution more efficiently. More importantly it should provide visual clues for any possible parallelization that can be introduced in the system to make it faster.

The first step towards this goal is to choose a programming language which is suitable for visual programming. [7] and [8] make compelling case for suitability of functional languages for visual programming. I discuss these and other arguments for the suitability of the J programming language for the purpose of this work.

The next step is to develop or choose an existing visual programming language that is expressive enough to express all the required J constructs. The visual language should ideally make use of a commonly understood and culturally independent metaphor so that it is easier to understand than the J syntax and provides additional visual clues towards the user regarding the solution that has been implemented using it. I discuss this in much greater detail in the section on Blockly.

The “Related Work” sections below will summarize some research that has been done in the domain of Visual Programming Environments. It will also provide some background information on the different tools and technologies that I have used in my work. In the next section, I describe the developed solution in detail. This section will provide the technical details of the solution along with the design decisions that were made during development. This will be followed by an evaluation of the solution. I will present a few sample problems and discuss how the abstraction of the visual programming does not affect the programmer’s ability to express the solution. The final chapter will detail the conclusions that have been drawn as a result of my work.

Related Work

This section will look at the various tools and technologies that have been employed in the development of this solution. We will also have a look at the relevant work that has been done in the domain of Visual Programming Environment and see what the existing literature says about the advantages and disadvantages of VPEs and what kind of languages are best suited for VPE representation. We will also take a look at some of the existing tools and evaluate their suitability for the purpose of this work.

Tools and Technologies Used

This section highlights information on the tools and technologies that I employed in the development of this solution. I will provide information on the programming language used (JavaScript), the Open Source VPE that I improved for code generation (Blockly) and the J programming language.

J Programming Language

In 1960 K.E. Iverson wrote a book entitled "A Programming Language", in which he introduced a mathematical notation which later came to be known as Iverson Notation. This notation presented a way to describe array manipulations. The core idea behind it was to provide a mathematical notation which would make it easier to specify computer programs and algorithms [9]. Adin Falkof and K.E. Iverson later wrote an interpreter for the Iverson Notation which, in reference to his book, came to be known as APL. The main guiding principles behind the design of APL were simplicity and practicality thus resulting in a programming language that provided brevity and was suited for practical usage in a myriad of scenarios [10].

In 1990, J was introduced as a simplified and enhanced version of APL which supported the ASCII character set and thus it was useable on any compatible machine. The major motivation behind development of J was to provide a more mathematical friendly version of APL by replacing the APL symbols with mathematical symbols using the ASCII character set for portability. This had the advantage of not only retaining the power of expression provided by APL but improving it as well. The original implementation of J was done in C using an unusual style of programming that relied heavily on C language's pre-processor facilities [11]. J was introduced as freeware and it is now available as open source software under the GPL3 license.

I will go into all the relevant features of J in the Solution section. I will explain sentence creation, trains, verbs, adverbs and conjunctions etc. and discuss in detail their role within the language as well as their relationship to code generation through Blockly.

Blockly

Blockly is a visual programming environment developed by Google. The primary purpose behind development of Blockly is to remove the burden of syntax from programming and providing a natural means for implementing the algorithm. It is a versatile web based tool

that can be adapted to be used in a multitude of scenarios. The following image is taken from one of the free and open source Blockly sample apps called Maze. This image depicts my solution to navigate the maze from the starting point to the finish line. There are other sample apps available with Blockly that depict code generation from Blockly blocks to JavaScript, Python or XML. It is the same sample application that was adopted to create a Blockly based visual programming environment for the J programming language. I will go into more detail about Blockly in the Solution section. I will also explain the block creation and code generation architecture of Blockly in detail then.

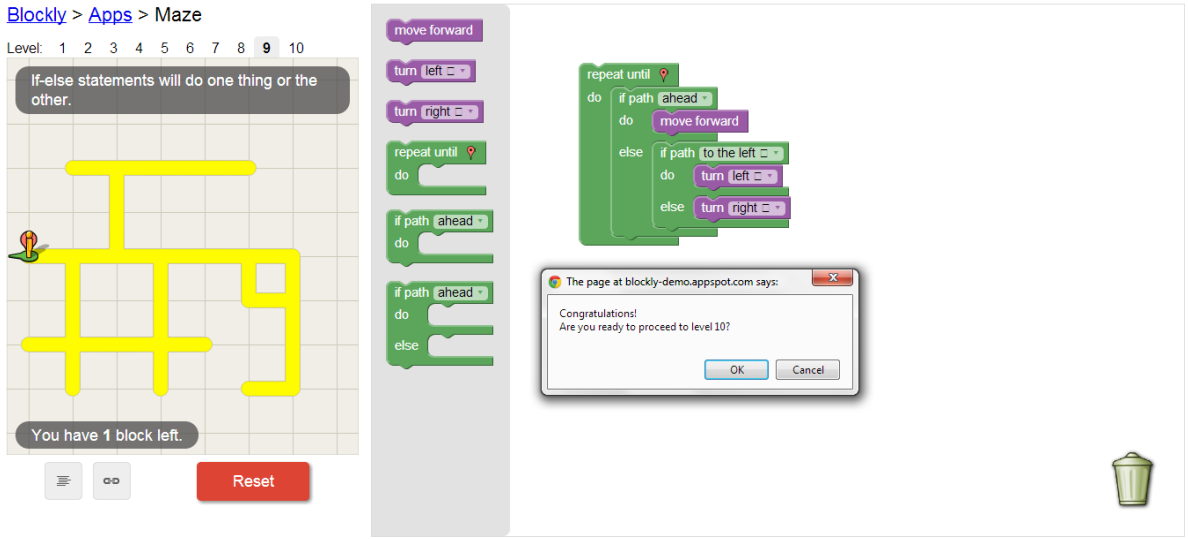

Figure 1 A sample app created using Blockly

JavaScript

JavaScript is an interpreter-based multi-paradigm programming language. It was developed by Netscape to provide a light-weight mechanism for developers to provide dynamism in web pages. All modern web browsers come with a built in JavaScript interpreter. It was standardized through the ECMAScript language standard and the 3rd edition of the ECMA-262 specifications is used by most modern browsers these days. JavaScript, along with CSS 3 and HTML 5, is leading the charge in innovation in web application development even today. It allows for creation of dynamic and rich web application development.

As the code is executed on the client side, through the browser embedded interpreters, these codes are also known as client side scripts. JS is mostly used to modify contents of a web page dynamically based on user action, location or other triggers. It can also be used to create non blocking function call backs thus enabling asynchronous activities. It can also be used for server side scripting for developing scalable network based applications. Some of the salient features of JavaScript are:

- Imperative Programming
- Object Oriented Programming
- Functional Programming
- Loosely typed dynamic binding

Blockly has been created using JS. It uses client-side scripting in defining, drawing, connecting and decoding blocks. All my development work in creating a VPE for J was done in JavaScript and showcases the power of this language which plays an extensive role in today web world. I will cover more technical details about JS in the solution section where I will go over all the features that I used and how they worked together.

Visual Programming

In this section I go over the development of visual programming and visual programming environments over time. I will describe some of the basic building blocks of visual programming environments and I will then look at some existing visual programming environments and discuss their merits and drawbacks.

Diagrams

As I discussed in the introduction chapter, visual programming environments use graphical components or icons to create computer programs. In the context of visual programming, an icon or a visual component represent the most atomic information of or about the system using a two dimensional visual representation.

To develop a visual programming environment, defining the language's iconic vocabulary would be the first step but only defining icons or symbols would be incomplete. It needs a way to show how these atomic elements connect together to create a program. Thus another relevant entity here is a diagram which comprises of these icons and depicts the interconnection between these icons to form a graphical representation of the given system. Perhaps the most comprehensive definition for diagram comes from James Maxwell (Encyclopedia Britanica, 11th Edition), "*A figure drawn in such a manner that the geometrical relations between the parts of the figure illustrate relations between other objects*".

Flowcharts

Flowcharts are used to graphically represent a process or an algorithm or a computer program. It is probably one of the oldest and most widely used graphical representations. The use of flowcharts for representing a process or more specifically a mechanical process originates from [12] where the term "Process Chart" is used to depict a processes. Goldstine [13] claims to be the first to use this concept for the depiction of computer programs in 1947. This text will use flowcharts for depiction of computer programs or algorithms.

Flowcharts comprise blocks and arrows. The blocks have two kinds of representation. The diamond shaped blocks are used to represent decision points whereas the rectangular blocks are used to depict an atomic step in the algorithm. It has a defined start and end node and all the steps required to go from the start to finish are outlined through arrows.

Data Flow Diagrams

Data Flow Diagrams (DFDs) are used to graphically represent the flow of data through an information system. They are very useful when modelling the context-level data flow through an information system. You can easily and comprehensively define all the data

sources and data sinks in the system. These sources or sinks can be internal as well as external. Each of the nodes (which are all connected through arrows) represents a process that the data flows into or out of depending on which direction the arrow(s) point to i.e. towards the node or away from the node. This can be used to deduce data manipulation or value addition of each process within the system.

These properties of DFDs make them ideal for modelling procedural programming languages. Software is usually developed to solve some real world problems and a majority of these problems can be modelled as processes thus making DFD a suitable match for graphical representation. Owing to this, many visual programming languages have been developed using DFDs. [14] provides very convincing arguments about the use of Data Flow Diagrams and its use in visual programming environments along with an analysis of the visual programming languages that use it.

A Brief History of Visual Programming

This PhD thesis [15], provides an overview of the history and development of programming language. According to Nickerson, the first visual programming environment was developed by Sutherland in 1966. It used flowchart-like visual representation to depict solutions. Similar representation has been followed by many of the visual languages that followed.

HI-VISUAL

[16] presents the iconic programming framework and a visual programming environment based on that framework called HI-VISUAL. It provides a framework for defining icons and their interactions with each other as a means for programming a system. The initial prototype for HI-VISUAL was developed on the SONY NEW under the UNIX using the X-window system and C Programming language.

HI-VISUAL used icons for representing real world objects. The interaction between these icons was achieved based on their layering over each other, the definition of icons themselves and a set of user defined rules governing the language semantics. The underlying principle behind the definition of icons was that they can only represent objects (physical or virtual) in the system. Unlike other visual programming environments which used icons to define functions as well, HI-VISUAL avoided confusing objects with functions by embedding functions within icons themselves. Icons had two views; *Internal* and *External*. The *External* view defined the graphical representation of the icon as well as a label. The *Internal* View required three pieces of information; *Concept*, *Substance* and *Messages*. The *concept* defined the name of the icon for semantic reasons within the language. *Substance* defined the internal state of the icon. Messages were further divided into two types. The *Acceptable Messages* are the messages that the object can receive from other objects and the *Transmittable Messages* are the set of messages that the object can send to others. These concepts can be mapped onto Object Oriented paradigm by thinking of the concept as class, substance as the instance variable defining the object state and messages as methods of the object.

Although this sounds confusing but it becomes less confusing once the interpretation of the icons comes into play. Each object can have an active role or a passive role. [Paper Ref] explains this through the following example. Consider a scissor and a paper objects. The scissor object has the transmittable message “cut” and the paper has transmittable message “wrap”. If the scissor object is considered to be the active object, it will perform “cut” operation on paper but if the paper is considered the active object, it will perform the “wrap” action over scissor.

HI-VISUAL also provides an additional feature to define rules. These rules provide greater flexibility when discerning object interactions e.g. by defining a priority rule which gives the priority to the scissor object thus the cut operation has more priority than wrap message of paper. Another important thing to note here is that, an action is only performed if one of the transmittable messages in the active object matches one of the acceptable messages in the passive objects.

HI-VISUAL provides a generic framework which allows the user to develop a set of objects and define their interactions using messages and rules that are suitable for a particular environment. But this same property binds it tighter towards a particular environment that is being modelled. Furthermore the icons have to depict entities within the system which may be difficult to do for complex virtual objects e.g. in an office automation environment it would be difficult to define graphical representations for applications and memos.

Visual Logic Programming

Visual Logic Programming (VLP) provided visual programming for Prolog. There were two main forces that drove the design principles behind this language [17]. The first was to use experimentally proven psychological studies in designing the icons that would act as the building blocks for this language [18] [19]. This design decision was motivated by the argument that by coming up with better pictorial representations, it would remove the burden of reading the visual representation from left to right or top to bottom or in any sequential manner rather allowing the user to process the graphical information being depicted in a parallel fashion.

The second design principle was to not do away with text altogether and instead combine the two aspects of to get the best of both worlds. Textual programming languages owe their dominance to the richness of expression that they provide. By combining the two it was hoped that VLP would help overcome the limitations of visual programming.

Although the icons being used were derived from experimental research, the downside to this approach was that they tried to model Prolog operations which being logical and mostly abstract in nature did not lend themselves well to graphical representation. But by basing it on some facts and due to the structure of Prolog, VLP was able to provide a good way of handling arguments to a function.

VLCC

[20] takes a very different and interesting approach towards visual programming. They developed a graphical tool that can be used to design a customized visual programming environment. The tool they developed is called Visual Language Compiler-Compiler. It provides a way for the language designer to define symbols, syntax and semantics for the visual programming language and then generates an integrated environment with graphic editor as well as a compiler for the designed language.

VLCC uses positional grammar as the basis of its design. A number of visual language parsers have been proposed [5]. One primary limitation in such parsing algorithms stems from the complexity of the language class involved. To get around this limitation, VLCC allows selection of a language class but supports multiple language classes at the same time for greater flexibility. The user can choose between language classes e.g. iconic design or DFD style design. The provided positional grammar model [6] uses the LR Parsing algorithm to support parsing of multiple visual languages.

VLCC has the added advantage of being able to support text with visual languages as well. The overall idea limits its potential user base to language designers only. It also does not provide any guidelines for language designs and thus leaves the onus of designing the graphic layout and symbols lies with language designers entirely.

VisaVis

VisaVis [8] provides a visual functional programming environment. The visual programs from VisaVis are converted to FFP (Formal system for Functional Programming). FFP provides support for higher order functions that form the basis for functional programming. It is used as the evaluation engine for generating results from the visual program.

VisaVis provides support from zeroth-order functions to second-order functions through a system of graphical representation. Functions of every order are assigned a meta-Icon signifying the function's order. Further information about each function is encoded through the use of colours and shadows thus providing a mechanism to present information about the function using graphical means. To parse the graphical representation (from VisaVis perspective) and to provide syntax (from the user's perspective) a strategy called substitution is introduced. This defines the interaction between the functions and evaluation order for these functions.

Although it is very well thought out, VisaVis fails to address some key issues. First of all, it does not address one of the primary characteristic of visual programming which is its ease of use for novice user or non-programmers. By developing a graphical syntax based around the order of functions, it requires the user to be well versed in lambda calculus or at least have sufficient concepts about the functional programming to be able to use the system. The use of FFP, while practical, limits its use as a general purpose programming language which is a critical success factor for any programming language.

Solution

This section will detail the technical details of the developed solution. I will go through the features of J language and look in detail as to what these are and how they are used. This will be followed by an overview of the Blockly architecture. I will explain the architecture for block generation and code conversion and take a look at how all these different components come together to create Blockly. Finally I will explain the required modifications and additions to Blockly to adapt it for code generation of J. I will go over the design decisions and the motivations behind these design decisions.

J Programming Language

This section will take a closer look at J Programming Language. I will go through the basic design principles of APL and J followed by the features of J language specifically the ones that are relevant to this work.

Background

J is a general purpose, array based programming language. It provides a comprehensive functional programming environment through its so-called tacit subset. Owing its background to APL, it is a very terse language where its primitives consist of one or two characters. It is, by design, very well suited for large scale and complex mathematical, statistical and logical operations over large data sets.

APL was originally developed at IBM based on the Iverson Notation, a mathematical notation for presenting array manipulations proposed by K. E. Iverson in his book A Programming Language. That is also the source of its name. The programming language APL was developed as an interpreter for the Iverson Notation [9]. It took advantage of the IBM Selectric Typewriter which could be used to depict a large number of symbols due to its changeable type elements. This was used to define a language with single character words which could be combined together to form sentences. The only exception in this was the variables which could be name using any combination of the alphabets. [11]

The use of special characters while advantageous in making the language terse did prove challenging when APL was being implemented. One major drawback of using symbols was that these symbols were difficult to print and not all system supported all the symbols (as the character sets had not been standardized back then). Thus in the paper titled APL/?, J was introduced as an enhanced version of APL. By using the ASCII character set in defining the language tokens, J successfully achieves portability while retaining the advantage of using single or double character words that form the language itself.

The basic mathematical symbols were adopted in J to bring it back to its mathematical roots. But as these symbols are limited (and even more limited in the ASCII Character set), J uses dot and colon to give new yet related meanings to mathematical symbols and to depict functions associated with these primitives. For example, J uses the symbol <. to represent

the min in an array. By using the dot with the less than sign makes it easier to associate meanings to these notations by acting as a mnemonic device. [11]

J is available as an open source project now through the J Software Inc. The primary source of distribution is through their website www.jsoftware.com. The current version of J is 7.0.1 and is available for Windows, Linux and Mac platforms in both 32 bit and 64 bit versions. J IDE is also available as iOS application that can be run over iPhone or iPad.

Array Based Programming in J

As I mentioned earlier, J is an array based programming language. It can be argued that only supporting arrays would take away from J's ability to be a general purpose programming language as not all real world problems can be resolved using arrays. J uses an ingenious approach in solving this problem where it remains an array based programming language yet still enable it to work as a general purpose programming language. For example, consider the following statement in J:

```
x + y
```

In this statement x and y can either be scalars or even multi-dimensional arrays and yet J would be able to handle it correctly in both the cases.

This example also highlights another of the fundamental characteristics of J i.e. it eliminates the need for a loop unlike a procedural or imperative programming language. The code written in J is much more mathematical in nature thus much more compact. Given that x and y are one dimensional arrays, most languages (like C/C++ or Java etc.) would need one loop to calculate the same sum. The number of loops required to calculate this same sum increased with the dimension of the arrays. But with J the programmer does not have to worry about any of these things and the language itself takes care of all these issues.

Structure of J

This section will discuss the structure of J language, more specifically I will discuss the constructs of J that are most relevant for the functional aspects of J through its tacit subset.

J uses a vocabulary that is based around the concepts of English grammar. A statement in J is called a sentence. As J is an interpreted language, each sentence is executable and if using J's gtk IDE, the output of each sentence is displayed immediately after pressing enter. Each J primitive in a sentence is similarly associated with an English grammar concept. The primary components of J are very similar in nature to the primary parts of speech in the English grammar. They are named similarly as well; noun, verb, adverb and conjunction.

Nouns

Using the popular programming convention as reference, nouns can be thought of as an object. They hold data in J. The important thing about nouns in J is the fact that the programmer does not have to provide a data type for it. The data type is determined

implicitly by the interpreter based on the value being assigned to the noun. Thus J is a weakly typed language.

Another way of looking at J nouns is in the perspective of functional programming. In functional programming, a zeroth order function is defined as a function that takes no arguments as input but still returns a value. [Need Ref] A noun is very similar in nature, such that it holds some data, and in J, when trying to determine the value of a noun, one just needs to type the name of the noun and the interpreter responds with its value hence the zeroth order function.

Verbs

It is best to explain verbs with reference to more commonly used programming terminology as well. Thus a verb can be thought of as a function or operator which takes in one or two nouns as input parameters and returns a noun as return value. When looking at verbs in the functional programming context, they can be thought of as first order functions which by definition take zeroth order functions as input to return a value [Need Ref].

Valence of Verbs

Valence of a verb defines the number of nouns that it takes as arguments. As in any other programming language, where each operator has a valence i.e. it is either a unary operator or a binary operator. Similarly, verbs in J are also characterised by its valence although the naming convention is different here.

A monad is a verb which acts on only one noun. And dyads are the verbs that act on two nouns. The cleverness in J lies in the fact that each verb in J has a monadic and dyadic behaviour. This holds true not just for the built in verbs but even for the user defined verbs as well. Every verb definition has these two associated behaviours. The interpreter decides the appropriate behaviour to call based on the usage of the verb. This behaviour can be best explained through an example,

```
2 >. 6
```

This statement in J returns the greater of the two operands i.e. 6. The verb '>.' appears in its dyadic representation. But the same verb when used in the following manner behaves differently (although in a mnemonically related fashion),

```
>. 4.5
```

This statement in J would return 5 because in its monadic behaviour, the same verb '>.' now acts as a ceiling function. This is an important consideration in user defined verbs and should always be taken into account.

An important thing to note here is that J limits the number of operands to two at maximum and one at the minimum. It is impossible to call a verb without operands. In case a verb requires more than two arguments, a monadic or dyadic invocation can be customized so

that rest of the arguments are passed as elements of an array (where the array itself would act as one parameter). Being an array processing language, it is very easy to unpack the array to gather all the required pieces.

Adverbs

Adverbs are part of speech of the J vocabulary which operates on one noun or verb to create a derivative primitive for J. They are used to change the functionality of nouns or verbs. The visual programming environment under discussion exploits this definition and treats adverbs as a special case of monads. This will be discussed in further detail in the relevant section when highlighting the features of the developed visual programming language.

Conjunctions

Conjunctions are part of speech of J vocabulary which operates on two nouns of verbs to create a derivative primitive for J. Conjunctions and Adverbs are also called modifiers in J as they both modify the behaviour of the associated noun(s) or verb(s). The proposed solution here exploits this and treats conjunctions as a special case of dyads.

Order of Evaluation

The monadic and dyadic behaviour of verbs, when looking just at the definition, may look confusing. But understanding the order of evaluation helps remove any ambiguity from the equation. To understand the monadic and dyadic invocations, the verb precedence and statement evaluations must be understood first.

There are two primary rules governing the statement evaluation in J. The first rule is that all J verbs (system defined and user defined) have the same precedence. This removes the burden of remembering the precedence of verbs and simplifies statement evaluation considerably. It also has the added advantage of getting rid of parenthesis to enhance precedence like in most other languages. The second rule is that the statement is always evaluated from right to left. For example, consider the following statement,

x * y + z

Because of equal precedence for all verbs, this statement is equivalent to x * (y + z) as opposed to the traditional mathematical interpretation (where multiplication has more precedence) of (x * y) + z.

When evaluating a statement, it is divided into fragments. A fragment is an executable bit of the statement. For example, each verb with its operands forms a fragment which is in turn independently executable. Thus a sentence is divided into fragments and these fragments are then executed from right to left. The result of each fragment's execution is then inserted back into the sentence and it is again evaluated from right to left based on the new fragments that now exist. Consider the statement,

x + - y

This statement can be divided into two fragments –y and x + (-y) forms the second fragment. Thus in order to evaluate x + (-y), the –y fragment needs to execute first. The negate verb (-) returns the negative of y (represented by _y) and thus the statement takes the form x + _y which when executed adds _y to x.

Another important thing to look at in the above example is the monadic and dyadic invocations involved. The simple rule governing interpretation is that if the verb has an operand to its left, the dyadic interpretation will be invoked; otherwise the monadic invocation will take place. In this example, the '-' verb acts as a monad because it only has one operand to work on. Similarly, '+' acts as a dyad and returns the sum of two operands. It is critical to know the correct part of speech being represented in a statement to understand the order of evaluation of a J sentence. Consider the following statement,

```
x verb 5
```

While the first reaction on looking at this statement may lead one to interpret it the dyadic verb being called upon two operands x and 5, it cannot be determined without knowing exactly what is being represented by x. 'verb' would only be dyadic if x is a noun but in case x is a user defined verb, this statement will generate very different result because verb and x will both be have monadic invocation here.

Tacit Programming in J

Tacit Programming is the style of programming which enables programmers to define functions without specifying any information about its arguments when defining the function. This paradigm provides even more terseness to J. A very important consideration here is that this tacit programming is a feature of J and is achieved through a subset of J primitives. But even while using a subset of J, it still is not limiting and allows the programmer to write general purpose code but with even more compactness. A good example of tacit programming is,

```
avg =: +/ % #

avg 1 2 3 4 5
```

The first line in the code snippet above defines a function 'avg'. It uses the '+' verb and using the '/' adverb, assigns the resultant verb to the variable avg using the =: assignment operator and finally it applies the operator to all elements in the array in the second line. The sum of the array is then divided using the '%' division operator with the length of the array, calculated using the '#' operator. The result of the second line's execution would give 3, which is the average of the given array.

Blockly

Blockly[1] is a web based visual programming environment developed by Google. It provides an easy to use interface for creating solutions by allowing the user to use drag and drop components from given toolbox. The basic idea behind Blockly is to provide means for the users to solve problems by piecing together given blocks. Being a web based tool, Blockly also removes any setup costs or barriers. Users can just open their preferred browser and start using the tool. The original code base (available through Google Code website), for Blockly, provides means for the user to generate code in JavaScript as well as Python.

Language Philosophy

Blockly is meant to be a tool for novice programmers[2]. It facilitates its target audience by providing a simple to use interface which removes the burden of syntax from programming and helps the users to focus on their problem solving skills instead. At the same time the tool is not meant to create any hindrance in learning the underlying language being utilized. This principle derives a lot of the design decisions in Blockly. Some of the prominent design decisions are highlighted in this section. I also describe the impact these had on the development of my solution.

String and Array Indexes

Majority of programming languages have the indexing scheme which starts with a zero. Most commonly it applies to arrays and strings. This can be a bit confusing for the novice programmer. Blockly solves this problem by starting all the indices from 1 and converting them to zero internally so that the process is transparent to the end user. Once a user is comfortable with the idea of array indexes, they can then adapt to the zero based indexing scheme more easily rather than having to learn it from the start.

My solution does not face this problem. As J is an array based programming language, thus all the verbs in J vocabulary work on arrays as well as scalar numbers. But for the rare occasion that indices are needed, they need to start from zero. But keeping with Blockly's design philosophy, the one based indexing scheme has been used here as well. It would be trivial for the user to revert to the zero based indexing scheme once he/she is comfortable with the language concepts and syntax.

Variable Names

Blockly provides a case insensitive variable naming scheme to keep things simple for the target audience. Thus a variable defined as var and another defined as Var are exactly the same. Furthermore, Blockly also does not restrict based on the variable naming conventions that a variable name should not start with an alphabet or that the only acceptable special character is underscore (_). The users are even permitted to use white space (character spacing) in variable names. The idea behind this is again to provide simplicity for the end user. Variable naming conventions are secondary for the user to learn and the more

[1] Blockly can be found at the Blockly's Google code page: https://code.google.com/p/blockly/
[2] https://code.google.com/p/blockly/wiki/Language

important thing is for them to understand the importance and role of variables in programming. Variable naming rules can be learned while learning the language syntax and should not inhibit the development of logical thinking. J is a case sensitive language. As the developed tool is based around the tacit subset, variable definitions are not included in it for now.

High Level Blocks

Blockly strives to provide as much abstraction as possible between the logic and the code as possible. For example, if a user wants to add two numbers and then later wants to add all the elements in an array, Blockly specifies that two separate blocks should be provided for these scenarios to maintain abstraction. The user may be able to get the same results using the addition operator combined with a looping block but to keep the focus on logical thinking and remove the semantics as much as possible, these simplifications should be provided.

This design philosophy is not embedded into the Blockly core design but is more a guideline for development of tools based on Blockly. As I only implemented a visual programming environment for the tacit subset of J, it is already very terse. Providing such layers of abstraction would have limited the strength of the developed visual programming environment. Thus, all the vocabulary of J has been provided in its atomic form as blocks and no consideration was given to providing any layer of abstraction for the end user. This was done in the hopes of making it easier for the end user to switch from visual programming to text based J if or when the need arose.

Language Dependence

Blockly was designed to be able to handle code generation for JavaScript primarily but it was designed not to be limited to JavaScript alone. Thus no language dependant assumptions were made during the design process. This makes Blockly ideal for code generation of any language that can be depicted precisely using Blockly's visual programming language.

Blockly's architecture goes one step ahead in this flexibility by providing separate code generation paths for each language such that code for multiple languages can be generated at the same time (as long as the visual language represent a similar solution for all the languages involved). Also, by keeping the code generation for each language separate, Blockly enables the user to make language specific assumptions in the code generation process without affecting anything else in the system.

Blocks

Blocks are the basic building block of Blockly. These act as the primitives of the visual programming language utilized by Blockly. Blockly uses a jigsaw puzzle style to represent blocks such that each block has protrusions and sockets. This jigsaw style of visual representation makes it very easy for the user to understand which blocks go together and how they can be connected to create the solution. Blockly enhances this understanding by providing a highlighted connection point when two compatible blocks are brought closer.

A protrusion represents the output of a block and the socket represents the inputs of a block. Each block can have at maximum 1 protrusion signifying that only one value can be returned by the function being represented by a block. Similarly, a socket represents inputs to a block. Thus the solution being represented comes together by joining protrusions with sockets. The following diagram shows two blocks. The purpose of these blocks are not important here (these will be explained in the following section). The important thing to note is the protrusions and sockets and the way they merge together. Also note the way that the socket is highlighted when a compatible protrusion comes near it, thus helping the user in understanding which components work together.

Figure 2 Sample Blocks

Defining Blocks

As Blockly's primary purpose is as an educational tool, it provides a flexible structure for defining blocks to allow the user to adapt Blockly to their specific purpose. While it provides a framework for defining blocks such that a user can define blocks to meet their specific requirements but at the same time not disturbing the overall consistency of the visual programming language itself. Thus any application that is built using Blockly would not only look similar but behave in a similar fashion as well. This is a testament to the generic nature of the jigsaw metaphor that is used to define these blocks.

Blockly provides a web based tool[3] that provides the users with an interface to define new blocks. This tool is developed using Blockly itself and is provided with Blockly as a sample application of Blockly. It has predefined blocks which represent all the configurable aspects of a block and the user can connect these together using Blockly's semantics to create new blocks. The tool provides a visual representation of the block as the user connects these blocks together along with the JavaScript code that the user would have to add to the correct .js file in order to use the block in their application. Important thing to note here is that this is provided as a utility to help users develop applications with Blockly as well as to provide them with a working example of flexibility afforded by Blockly. The following diagram shows this tool in action.

[3] http://blockly-demo.appspot.com/static/apps/blockfactory/index.html

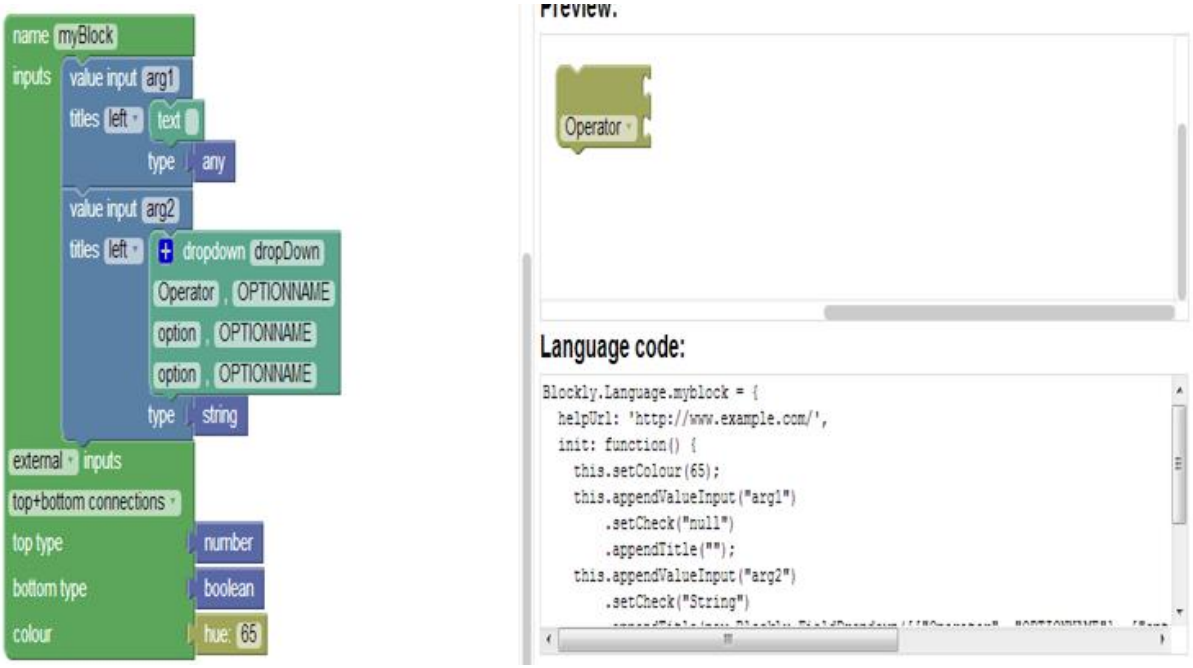

Figure 3 Blockly's sample app for Block generation

I will now discuss all these parameters that can be configured for defining a new block[4]. The following code snippet defines a block shown in the diagram above:

```
Blockly.Language.myblock = {
  helpUrl: 'http://www.example.com/',
  init: function() {
    this.setColour(65);
    this.appendValueInput("arg1")
        .setCheck("null")
        .appendTitle("");
    this.appendValueInput("arg2")
        .setCheck("String")
        .appendTitle(new Blockly.FieldDropdown([["Operator", "OPTIONNAME"], ["option",
"OPTIONNAME"], ["option", "OPTIONNAME"]]), "dropDown");
    this.setPreviousStatement(true, "Number");
    this.setNextStatement(true, "Boolean");
    this.setTooltip('');
  }
};
```

There are three main components that can be adjusted to configure the block to the user's requirements. These components are:

- Block Name
- helpURL

[4] https://code.google.com/p/blockly/wiki/DefiningBlocks

- init Function

Block Name

This appears in the first line in the code snippet above. The block name in the above code is "myBlock". All blocks share the same namespace in Blockly and thus these names need to be globally unique. The block names that I use make use of the J vocabulary (e.g. adverb and conjunction etc.) for uniqueness.

helpURL

This property allows the user to provide a URL that point to a help page with information on that particular block. This can be used to provide a user guide or documentation for your application. Another interesting use of this property can be to specify a function here instead of a URL. This function can then be used to dynamically generate help URLs, thus providing a flexible framework for specifying documentation.

init Function

This function controls the looks and behaviour of the block being created. User can control this through a host of properties. An important thing to note here is the use of "this" in the init function. "This" here points of the block itself. The following options are configurable inside the init function.

- setColour
- setOutput
- setPreviousStatement
- setNextStatement
- appendDummyInput, appendValueInput, appendStatementInput
    - setCheck
    - setAlign
    - appendTitle
- setInputsInline
- setTooltip
- setMutator

setColour

Blockly uses Hue-Saturation-Value (HSV) model for defining colours for blocks. But looking at the code snippet, only one value is passed instead of the required three arguments.

```
this.setColour(65);
```

This is by design so that the user only needs to specify the value for Hue to change the colour of the block. The values of Saturation and Value are predefined in Blockly's core. This is a deliberate design decision on part of Blockly to make it easier to specify colours. The user can set a tone for their application using Saturation and Value e.g. bright colour scheme for applications targeted to children and duller colours for business applications. Eventually just by setting the Hue at block level, user can create different coloured blocks to represent different types of blocks for easy visual recognition.

setOutput

Every block being defined needs to define a way in which it interacts with other blocks. As discussed in the Introduction to this chapter, Blockly achieves this using a Jigsaw puzzle metaphor where blocks have protrusions and sockets depicting output and inputs respectively. User can define the output of a block using the setOutput() function. For example, setting this property as follows in the code at the start of this section,

```
this.setOutput(true, "Array");
```

will create a block that would appear with a protrusion on the left as below:

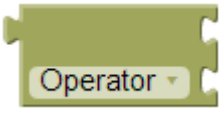

Figure 4 Sample Block with setOuput set to TRUE

This function takes two arguments. The first argument, a Boolean, causes the protrusion on the left to appear when it is set to true. If the function being represented by the block is not supposed to return any value, instead of setting it to false, this statement should not be included in the code at all.

The second argument here is the data type of the value being returned. In this example, we define it to be Array meaning that this block would return an array. Users can also provide an array of data types in this argument for cases where the function may return more than one data type. It is important to understand here that the block would still only return one value, but it may happen that for one instance it returns Integer and in another it may return String.

Notice that this argument is a string so any suitable string value describing the return type can be used here. The only consideration is that this return type should match others being used in the system as Blockly assesses the compatibility of blocks when connecting them together using this data type. Hence, a block that returns Array would not be able to connect to a block which is expecting an Integer or a String. This mechanism ensures type checking and hopes to help novice programmers understand type checking better.

setPreviousStatement

Not all functions return a value. Yet they would still require a way to connect with other components in the Blockly visual language. This can be achieved using the following statement,

```
this.setPreviousStatement(true);
```

The consequent block defined would have the following shape,

Figure 5 A sample block with setPreviousStatement set to TRUE

The statement above caused the notch on the top. This notch signifies that this block does not return a value but at the same time provides a visual clue as to how this block will fit in with other blocks in the visual language. This can be thought of as defining the control flow of the program being constructed.
None of the blocks defined in my solution use this statement. In the tacit functional subset of J that is under consideration in this project, all the functions return a value.

setNextStatement

This function is very often used in conjunction with the setPreviousStatement function to signify the role or behaviour of the block in the control flow of the program. Like the setPreviousStatement function, this signifies that this block does not have any return type but provides a visual clue as to how this block will connect with other blocks. The figure below shows a block with both setNextStatement and setPreviousStatement being used.

Figure 6 Sample block with setNextStatement and setPreviousStatement set to TRUE

Defining Block Inputs

Blockly provides three ways for specifying inputs to a block. These all depend on the type of input required. Blockly allows the user to define if the input would be a statement, a value or a dummy input and then control properties of these input statements. The primary reason for this differentiation is to make it easier to generate code by providing different avenues for different scenarios.
The three main inputs can be:

- appendValueInput
- appendStatementInput
- appentDummyInput

appendValueInput

As the name indicates, this type of input allows the user to pass values to the block. Using the following line of code, the user can create a socket (depicted in the image below) to capture input(s) to the function.

this.appendValueInput(“arg”)

Figure 7 Block depicting value input

This function accepts a string as input argument. This string is the parameter name that will be used at the time of code generation to reference it during the code generation.

appendStatement

This type of input is used for cases where instead of an input parameter; the block expects another block (representing a statement of the code mostly). Although Blockly is not limited to provide a visual programming syntax for text based programming languages alone, this

function can best be explained using example from programming languages. A common use case for this would be iterative statements or control structures in a programming language.

Figure 8 Block depicting statement input

In this figure above, the input part of the block has a protrusion (similar to the one achieved by setNextStatement function) where compatible blocks can be fit to create the control flow required.

appendDummyInput

Dummy inputs as they name suggests do not provide any form of connectivity. They are dummy placeholder to hold text and/or drop down items. These are provided as a means for more descriptive visual presentation and to allow configurable behaviour to the user. Using the figure in the previous section, the top line that reads, “repeat 10 times” has been created using the appendDummyInput function. It provides a way for the user to enter the number of iterations for the loop involved.

This highlights two other features provided by Blockly. First it is possible to combine more than one input statements together to form a block, as in this last example, hence providing more opportunities for customization to the end user. The second thing of note here is that this looping block does not bind itself to any language construct. This block can be converted to a for or a while loop during code conversion and this implementation detail can remain transparent to the end user. Defining a block for for loop or a while loop is equally easy but by this highlights how abstraction can be achieved in Blockly and the user has full control over the level of abstraction required.

Configuring Input Statements

There are more options for each of these input statements which the user can use to configure the block inputs to their specifications. I will describe them in this section as they are common to all three types of input statements.

appendTitle

This function is used to add information about or for the input statement. It should be specified for each input statement separately and can also be used multiple times for the same statement to provide more information. This additional information can be of the following forms,

- Textual titles
- Image
- Drop down menu
- Text input
- Variable

- Check box
- Colour

I only describe a few of these here which I have used in my solution. The following code can be used to give a textual title or a label to the input statement,

```
this.appendValueInput("arg1")
    .appendTitle("Title");
```

This statement gives a label (“Title”) to the input statement as depicted in the figure in the appendValueInput section.

A drop down menu is very useful when presenting users with a preset set of options and to get them to choose one of them. Because of its widespread use over the years in web applications as well as desktop applications, a dropdown’s visual representation is widely understood. The following code adds a drop down menu to an input statement,

```
this.appendDummyInput()
    .appendTitle("repeat")
    .appendTitle(new Blockly.FieldTextInput("10"), "2")
    .appendTitle("times");
this.appendStatementInput("loopStatement")
    .appendTitle(new Blockly.FieldDropdown([["for", "FORLOOP"],
                                            ["while", "WHILELOOP"]]), "dropDown");
```

This code results in a block that looks as follows,

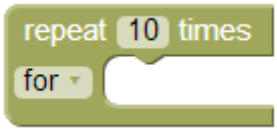

Figure 9 Block depicting dropdown menu

In the appendStatementInput section above I described an example of a looping block. I discussed the fact that the particular looping structure was independent of the implementation detail because there was no mention that the loop would be a for loop or a while loop. The example above takes the same example a step further by providing a drop down menu for the end user to choose which particular loop they would like the implementation to be in. Also, note the use of multiple appendTitle function calls for the appendDummyInput function and the way they come together to create an easy to use block by augmenting the visual representation with appropriate titles.

setCheck

I discussed type checking in the setOutput section above. Each block specifies its return type (in case it returns a value). During that discussion it was mentioned that Blockly uses this property for type checking. setCheck is also part of the type checking framework. By using this function, user can define the acceptable input values for the input statement being used. Two blocks can be connected together only if the data type(s) mentioned in the setOutput function of a block match the data type(s) in the setCheck function of the other block.

```
this.appendValueInput("arg1")
    .setCheck("Boolean")
```

This statement would make sure that only blocks that return a Boolean can be used as input to this block being defined.

Other Configurations

There are some other configurations that are available to Blockly users in creating blocks for their applications. I am listing them here for the sake of completion. I did not have to use any of these in my development.

- setInputsInline
- setMutator
- setTooltip

A Visual Language Based on Blockly

This section describes the visual programming language that I created using Blockly. As discussed in the previous section, Blockly provides a flexible framework for creating blocks to suit the user's specifications. In this case, I derive these specifications from the J programming language. I described the primary parts of speech for the J programming language in the relevant section above. I also discussed the formation and evaluation of J sentences. That discussion leads to the conclusion that if there exists a visual programming language which provides a way for the user to define these parts of speeches and connect them together, that would provide the required semantics to create a fully functional visual programming environment for J.

So, I had to define blocks that would represent these parts of speech using the framework provided by Blockly. I describe these blocks in this section.

Block for Verbs

Verbs in J are equivalent to functions in other programming languages. J provides two behaviours associated with a verb based on the number of arguments it takes. The two types of verbs are called monads (verbs with one input parameter) and dyads (verbs with two input parameters). Unlike most other programming language where number of arguments can be variable, J verbs can have a minimum of one and a maximum of two arguments. If more than two arguments are required, then one of the parameters in a dyad should be an array consisting of all other parameters which can then be extracted inside the verb definition.

There were two possible ways of handling verbs using Blockly. One way to do this would be through the use of mutator blocks. These are blocks which can be dynamically changed at run time by the user to meet their requirements. Thus I only needed to provide one block for verbs, with one value input, and the users would be able to add another input value at runtime if they wished. While technically this is possible, I see two major problems with this approach. The first problem is that this goes against the basic language philosophy of Blockly which states that a separate block should be provided for each high level function. Mutator blocks would be very good if the number of arguments was variable but with only two possible options it goes against Blockly's design philosophy. Another issue with this approach would be in code generation. While most other languages would accept input parameters in a comma separated list, J has a more mathematical notation. This would require more conditionally complex code for code generation.

The second approach which I employed for this is to use two separate blocks for each type of verbs. This approach complies with Blockly's design philosophy for abstraction as well as the code generation is similarly segregated for easy maintenance. These blocks also provide a great way for the user to learn the difference between scenarios where a verb is using its monadic or dyadic implementation (as the decision to use a particular implementation is made at run time based on the number of arguments).

The Monad Block

Monad is a verb with only one argument similar to unary operators. This makes it straightforward to define a block for monads. There are two basic input requirements for the required block. It should allow the end user to provide one input and also provide a mechanism for the user to specify the verb that is going to be used. Being a verb, it will always return a value hence the block needs to provide an output mechanism as well. The block I defined is shown in the figure below.

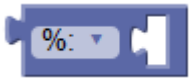

Figure 10 Monad Block

This block fulfils all three requirements highlighted above for monad. It makes use of three primary features of Blockly. It provides a protrusion that provides a mechanism for returning value. The drop down menu in the middle contains a list of all the verbs provided by J and the user can simply select the required one to use. The socket on the left is an input of value type which means it expects a noun as input. The "%:" symbol in the middle is the J verb for square root.

The Dyad Block

Dyads are verbs that expect two input parameters similar to binary operators. Other than the one extra parameter, all other requirements for a dyadic block are the same as the monad block. Thus, this block provides one extra socket for adding the second parameter. The block I used for dyads is shown in the figure below.

Figure 11 Dyad Block

With monad there is only one argument and because syntactically it is written to the right of the verb, the input socket provides a visual representation which is very close to the actual J code that will be generated against this block. The same is true for the dyad block as well. There are cases where the order of the argument may be important e.g. in case of the dyadic verb power (^). For this verb the left argument is the base and the right argument is the power. Hence the visual representation of the block matches J notation closely. If it is used as a training tool for novice programmers this, consistency between visual and textual notations, should make it easier to move from Blockly diagrams to textual J.

Inline Inputs for Verbs

One of the primary reasons for pursuing a visual programming environment is the fact that the human brains processes graphical notations in a parallel manner. [Need Ref] Combined with the fact that the visual representation can give obvious clues as to the parts of the system that can benefit from parallel processing, the blocks defined above seem counter intuitive. While they provide an excellent visual representation of the underlying J code, they hide this parallelism that can be derived from the visual representation. Thus a balance

needs to be achieved between the two. This is where another feature of Blockly comes into play. The user can change the appearance of the inputs of a block at runtime. The blocks defined above use the inline input style which provides a visual differentiation between the left and the right argument. In cases where these inputs are nouns, this representation works well. But consider a case where each argument is the output of another verb or a combination of verbs. In such a case it would be useful to see these inputs in a manner that would allow the user to decide if it would be more beneficial to execute each of the inputs in parallel. The following diagram shows the alternate visual representation for dyads and monads.

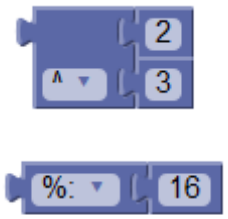

Figure 12 Difference between inline and External inputs

The user can choose between the two representations at run time by simply right clicking on the block and choose between Inline Inputs and External Inputs options from the context menu.

Blocks for Adverbs

Adverbs are like monads such that they only need one operand. But unlike a monad which operates on a noun, adverbs operate on a verb. A block for adverbs needs to cater to the same set of requirements as a monad. The only difference being that instead of a value, it requires a verb.

This difference drives the thought process towards the statement input mechanism provided by Blockly which allows a block to accept other blocks as inputs. When I discussed that input process above, I used the example of a loop to highlight its possible use with respect to code generation. While the example is very apt at describing the intended functionality of statement input, it is a little misleading in this context. The loop example works for conversion to JavaScript because all the statements inside the loop need to be executed. But in this case the adverb is only applied to one verb and not all the statements that may form part of the input.

The code generation for J also becomes complex by using this technique as it then requires a lot of string manipulations to insert the adverb at the correct place with the correct verb. While it is possible, it imposes a limitation by defining semantics in the code generation that may impose limits when the developed tool is enhanced in future. Keeping in line with the design philosophy of Blockly, I created a block where the user can choose a verb and the corresponding adverb from two dropdown menus provided within the block. This simplifies the code generation code a lot as each selection from the drop down is available as a separate string at code generation and thus it is trivial to concatenate them in the correct

syntax rather than having to do complex (and potentially limiting) string manipulations. The block I used for adverbs is shown in the diagram below.

Figure 13 The Adverb block

User can choose the verb and then the required adverb. For example, in the figure above the adverb Insert (/) is being applied to the dyad Addition (+). This combination will return the sum of all the elements in the array that can be passed as input to this block.

Block for Conjunctions

I discussed conjunctions in detail in the chapter on the J programming language above. Conjunctions are similar to adverbs and differ only in so much that unlike adverbs, conjunctions require two verbs or two nouns as input parameters. Thus the requirements for the desired block are very close to that of a dyad. Although it is very similar in design to dyads, I provide a separate block for conjunction in order to provide clear distinction between dyads and rank. The image below shows the block for conjunctions.

Figure 14 The conjunction block

Code Generation

The visual language used by Blockly was defined in a context independent framework such that its usability was not limited to visual programming language rather to provide a generic framework that can be adopted for any suitable purpose. Generally speaking the jigsaw metaphor lends itself well to modular problem solving even if it's not strictly a programming problem. This is apparent from the sample applications provided by Blockly which include a maze navigation app, a drawing app, Block generation app etc. which all let the user solve some problem in modular steps. Similarly, using Blockly for code conversion is also provided as a sample application to highlight this particular use. The provided sample app provides code generation framework for JavaScript, Python and XML. It makes use of a generic set of blocks which represent variables, mathematical functions, loops and control structures and thus allow the user to create a logical solution which the tool then converts to the desired programming language.

The same approach is not feasible for code conversion to J as J does not rely on the same constructs for many things. This is especially true for the tacit subset of J (which is the focus of discussion in this work). Looking at the basic J philosophy of sentence creation and evaluation makes us realize that it is not feasible to adapt the existing solution for our purpose. This is also the reason why new blocks had to be defined to represent J constructs more accurately.

Blockly provides a well structured mechanism for code generation. Every block, which is defined in the intended system, has a corresponding action function associated with it. This function defines the behaviour of the block. I discussed block definition in the last section and the primary focus was on the visual representation in that section and not the behaviour. This section discusses the code generation for the blocks that were defined in the last section.

Code Generation for Monads

This is probably the easiest construct to generate code for. From the discussion about Monads and then about the monad block, there are only two things that are needed to generate code for a monad i.e. the verb and the associated noun. Blockly provides a generic framework to perform actions against the blocks that the user use. The framework includes calls to retrieve the attached input parameter(s) and the selected drop down item. From the block definitions above, these are the two configurable items in the proposed language.
Using these calls, the code generation for monad consists of extracting the selected operator from the drop down and then embedding it with the input parameter. Thus the final code is simply the concatenation of the two strings.

Code Generation for Dyads

The code generation for dyads is very similar to monads. The only difference is that in this case we have two input parameters to the block. According to the semantics of the J programming language, the two nouns appear either side of the verb. Thus again the code generation is simply a concatenation of the left noun, the dyadic verb and the right noun.
Code Generation for Adverbs
In my discussion of block definition for adverbs, I pointed out the possible implication on code generation based on the design of the block and my reasons for selecting the particular block style that I used. The design of the block makes code generation for adverbs very easy. According to the semantics of the J programming language, the adverb appears between the verb and the required noun. Thus the code generation in this case boils down to simple string concatenation again with the adverb in the middle of the verb and the noun.

Code Generation for Conjunctions

Conjunctions operate dyadically over two verbs. This is why the block design is very similar to that of a dyad. I describe my design decision in keeping a separate conjunction block in the discussion of block definitions. Because of the block design being the same as dyads, the code generation follows the exact same rules as dyads as well.

Evaluation

This section provides an evaluation of my work. I explained the evaluation criterion that I used. I explain the criterion with the help of a couple of examples. I end the discussion with the results and explanation of my evaluation and how they relate to other works done in the related domain.

Textual Complexity Metrics

There are a number of metrics available to calculate code complexity. [15] provides a good review of many of the available metrics and discusses their advantages, disadvantages and their suitability for visual programming languages. The idea is to have a quantifiable means of calculating the complexity of a piece of code. This leads to the question of what are the quantifiable attributes of code that would be representative of the complexity of the program. For this work, the same problem needs to be expanded to include visual programs. Thus the question arises, is there a complexity measure available that would lend itself to both the text based program and a visual program.

Character Count

The number of characters in a textual program can be considered one measure. It is a very easy to calculate and repeatable metric, which are very desired characteristic of any complexity metric. It would only require implementing a simple character filter in any programming language which would only need to parse a program file and count the number of characters. But the main problem with this metric is that it is based around the fact that the length of the program is indicative of the complexity of the program. It does not take into account the complexity of the task being achieved and how difficult it may be to understand for a novice programmer. This metric specifically fails for J which is a terse language and achieves much complex operations using mathematical notations which would only appear as a single or a double character in the metric and would disregard the complexity of the operation performed.

Lines of Codes

The next logical attribute to be considered may be the lines of codes in the program. This is again based over the assumption that a long program would be more complex than a short one. While this may be true in some cases, it is not true for all the cases because this metric does not take into account the complexity of each line of code. But at the same time it provides a reproducible and easy to calculate metric. Like the character count, this metric is also not suitable for my purpose because of the terse nature of the programming language involved. LOC and character count both suffer from another drawback that they do not lend themselves to visual programming where there are no lines of codes and thus no characters to count.

Conditional Statements

The LOC approach suffer from the drawback that it does not take into account the complexity of each line of code. Thus the next possibility would be to quantify the

complexity of lines of codes and form a metric around that. In [21], McCabe provides a measure along these lines. In simple words, McCabe metric is based around the fact that the number of conditional and loop statements in the code is representative of the complexity of the program.

This is an interesting metric as it takes into account the length of the code and then weighs it according to the number of the conditional structures as an indicator to the complexity of the code. Another interesting aspect of this metric is that it was originally developed based on graph theory approach thus it can potentially be used for visual languages as well. A Control Flow Graph (CFG) depicts the possible execution pattern of the code and forms the basis of the McCabe metric (including the number of edges, nodes and the connected components). As I described in the related work section, many visual programming languages use a graph like structure as a basis for the visual representation involved (commonly utilizing DFD). Thus, the textual measure for McCabe is defined as,

complexity = number of conditional statements + 1

Whereas the graphical metric is,

complexity = edges - nodes + 2(connected components) [15]

While it fulfils almost all the shortcomings for complexity metrics, pointed out in the last few sections, it is still not a good indicator of complexity for my work. Firstly, it is still based around the length of the code. While J provides complete support for loops and conditional statements, being an array based programming language, they are seldom used. Unlike most other programming languages where the number of conditional statements is limited by the language constructs, a J program may have implicit conditional statements which would be difficult to detect by an automated tool. Secondly, the complexity metric is based around a graph which is not the underlying metaphor in Blockly. While it may be possible to map the jigsaw metaphor to graph one, by categorizing edges, nodes and connections, it would still require a different language parser (than the one used by Blockly) to parse the visual program and count these elements and the resultant metric will still suffer the flaw described previously.

Halstead

Halstead proposed another widely used metric for calculating software complexity. [22] In its simplest form, this metric is the sum of the number of operators and the number of operands in the code. Because of J's mathematical nature of expression and terseness, this provides an excellent metric to calculate the complexity of a program written in J.

According to [Nickerson], the Halstead metric for textual program can be expressed as the sum of the number of operands N1 and the number of operators N2 in the code.

$$N = N1 + N2$$

Halstead also provided a few other metrics based on this criterion. One of these is the Volume, which involves the length of the program and the minimum bits required to represent the program,

$$V = N \log_2 n$$

where $N = N1 + N2$

and n = n1 + n2

where n1 and n2 are the number of distinct operators and operands required. This metric is not particularly useful here as there are no memory considerations involved. The primary measure used here is the program length derived by the sum of the number of operators and operands used.

Another factor, which I mentioned in the beginning of this section, in the selection of a complexity measure is its ability to lend itself to the visual programming language involved. [15] provides one way to adapt Halstead metric for use with visual languages. The proposed solution suggests that the complexity of the visual program can be expressed as a sum of its nodes and edges. While my solution does not directly involve nodes and edges as in a conventional graph, the metric is still useable. I categorize every block as a node and every input as an edge. This categorization maps the jigsaw metaphor used by Blockly, to the graph notation required by the metric. It also provides a direct mapping between the textual measure and the graphical measure as each block represents an operator and each input represents an operand.

Visual J

Consider the following example for calculating the average of an array,

+/ 1 2 3 4 5 % # 1 2 3 4 5

The +/ operator calculates the sum of the array (1 2 3 4 5) and then divides (%) it by the length of the array, calculated by #. The Halstead measure of the complexity of this program amounts to 5 (two operands and three operators). Now the equivalent visual program using Visual J would look as follows,

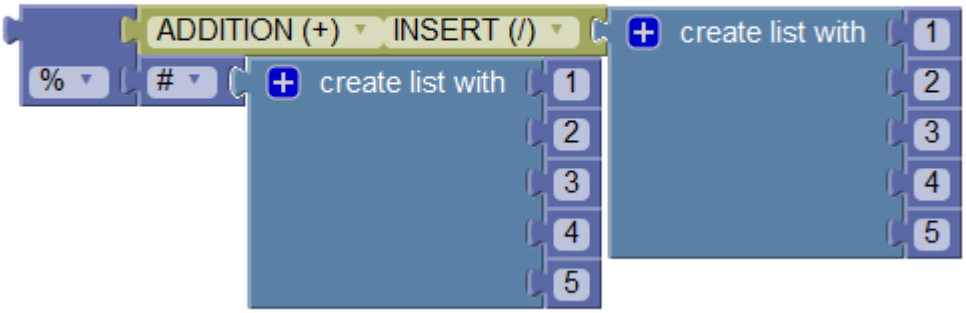

Figure 15 Visual J code for calculating average

From the visual program itself it is apparent that the same program when implemented using the visual programming language amounts to the same value of 5 (3 operators and 2 operands).

Understanding Results

It is apparent from the complexity metric that the visual language does not add any more complexity to the program than the original textual language. There are a number of

possible conclusions that can be drawn from these results. I discuss the significance of these results in this section. The discussion involves the evaluation of other visual languages and how my solution compares to them. I also provide more than one perspectives of looking at these results and what that means for the viability of my solution.

The result indicates significant success in reducing the complexity of the visual programming language as compared to their textual counterparts. Nickerson [15] evaluation of the Visual APL, as well as other visual frameworks, clearly shows that the complexity metric of the visual language is in most cases higher than that of the textual equivalent. My solution provides the same level of complexity as the textual version, which is a significant improvement over other attempts at visual programming languages.

It can be argued that the visual language should have less complexity than the textual equivalent to be of any help. The problem stems from the fact that the real advantage provided by the visual programming languages lie in their graphical nature. The ease of connection and better understand ability makes using them easier than conventional textual languages. Unfortunately, these characteristics are not measurable meaningfully.

Also, the use of visual programming brings with it other side effects as well. For example, the J code above for calculating the average would take much less time to type than to compose the same program through visual language. Each block involved in the visual language requires to be found and then drag and dropped into the correct place. The typing comes with its own hazards though. There is a fair chance of user to making a typo thus causing compilation errors. Also, the user will have to either remember the constructs or look them up. Again measuring these parameters is not possible as they may vary from person to person and depend on things like the users knowledge of the programming language, their skill with programming in general and their skill with computers etc.

A fair conclusion that can be drawn from this discussion is that a novice user may be much more comfortable using the visual language as it provides ease of use and removes the burden of remembering the constructs and the semantics of the language. The semantics of the language are built into the visual constructs along with the available vocabulary. On the other hand an experienced J programmer may find it much easier to write the code using the textual J because of his/her experience and knowledge. They still might benefit from the visual representation for better understanding of the control or data flow in the program and may be able to identify possible parallelization that may be achievable in the same program. This interpretation is consistent with Nickerson's [15] evaluation of visual APL as well.

There are other issues that have been addressed with this solution though. Blockly provides a mechanism for the solution to be scalable for larger programs. This is where the compactness of J becomes very helpful as verbs, adverbs and conjunctions provide complex functionalities and being an array based language, it provides independence from loops in general as well.

Conclusion

I discussed in the introduction some of the issues that plague visual programming. First of all it is imperative that the visual language should make use of a visual metaphor that transcends cultural boundaries to be universally acceptable. Secondly, there are limitations to the amount of information that can be depicted through a visual language. These limitations arise from a multitude of factors. For example, while iteration is easy to specify, recursion presents a challenge for visual languages. It is trivial to represent iteration using flowcharts and/or DFD but there is no commonly used graphical notation for recursion which leads to the problem area of coming up with a self explanatory and intuitive graphical representation for it. Another limitation is the depiction of abstract actions. While the use of graphical user interface over time has provided standardized icons for a number of actions, there are still a number of actions that can be performed in the system which do not have standardized graphical representation. Programming constructs are even more abstract as I just explained using the recursion example.

These were the challenges that were kept in mind while working on this project. The problem of utilizing a universally acceptable metaphor could be expressed by either utilizing a metaphor that is globally understandable without any cultural bounds or use one that has become accepted through wide spread use. This criterion was kept in mind while choosing Blockly which makes use of a widely understood jigsaw puzzle metaphor for the visual language. The J programming language, because of its design, solved a number of other problems associated with visual programming. The tacit subset of J worked without recursion, iteration or object oriented programming thus eliminating any reason for coming up with symbolic representations for these abstract concepts. Based on the success of Prograph and Visual APL [15] it was understood that functional programming does provide a better framework for visual representation which provided another reason for using J.

As discussed in the previous section, the results suggest that the solution is useable by both a novice programmer as well as an expert programmer. The uses that can be extracted from it are different for each type of user. The proof of concept that was developed for this tool suggests that with some more improvements, it can be used widely in a number of circumstances. This is an additional value that has been added on top of all the problems that were avoided based on the selection of J and Blockly as discussed above. With some more focused work on creating a finished tool, this could provide a user friendly environment for using J to solve complex and scalable data analysis problems.

Future Work

I believe that the ultimate direction for this tool should be to get it to a point where it provides general purpose usage. To achieve this goal there are features that need to be added to it which are useful for novice programmers as well as experienced programmers.

The most important feature that needs to be incorporated is to implement remaining features of the J programming language. While the proof of concept implements the

available verbs, adverbs and conjunctions there are still aspects of J language that are not implemented. The most important of these is to provide a framework for implementing trains. A train is a sequence of operators only which are assigned a name (through assigning them to a variable). This train can later be called over data and provides a reusable code. By providing support for trains in the tool, the user should be able to create highly scalable solutions.

In order to provide a proper development environment, there needs to be a way for the user to execute the code and see the output. J provides a web based IDE which can be adapted to connect it to Blockly for code execution. The J server can be hosted on any web server and the code generated from Blockly can then be sent to the J server for execution. The results can then be displayed in the same Blockly window for user.

I have discussed the possibility of introducing parallel programming with the help of visual programming throughout the document. The emphasis so far has been on identifying what parts of the code can be executed in parallel. This could be automated so that the programmer does not have to worry about it. The result would be better performance than textual J with no impact on the development process.

Bibliography

1. N. C. Shu, *Visual programming*, Van Nostrand Reinhold Co., 1988.
2. G.G. Roy, et al., “Towards a visual programming environment for software development,” *Proc. Software Engineering: Education & Practice, 1998. Proceedings. 1998 International Conference*, 1998, pp. 381-388.
3. E.P. Glinert, et al., “Visual tools and languages: directions for the '90s,” *Proc. Visual Languages, 1991., Proceedings. 1991 IEEE Workshop on*, 1991, pp. 89-95.
4. Z. Da-Qian and Z. Kang, “On the design of a generic visual programming environment,” *Proc. Visual Languages, 1998. Proceedings. 1998 IEEE Symposium on*, 1998, pp. 88-89.
5. K. Wittenburg, “Earley-style parsing for relational grammars,” *Proc. Visual Languages, 1992. Proceedings., 1992 IEEE Workshop on*, 1992, pp. 192-199.
6. G. Costagliola, et al., “Towards efficient parsing of diagrammatic languages,” *Book Towards efficient parsing of diagrammatic languages*, Series Towards efficient parsing of diagrammatic languages, ed., Editor ed.^eds., ACM, 1994, pp. 162-171.
7. S. Matwin and T. Pietrzykowski, “PROGRAPH: A preliminary report,” *Computer Languages*, vol. 10, no. 2, 1985, pp. 91-126; DOI http://dx.doi.org/10.1016/0096-0551(85)90002-5.
8. J. Poswig, et al., “VisaVis: a Higher-order Functional Visual Programming Language,” *Journal of Visual Languages & Computing*, vol. 5, no. 1, 1994, pp. 83-111; DOI http://dx.doi.org/10.1006/jvlc.1994.1005.
9. K.E. Iverson, *A programming language*, John Wiley \\& Sons, Inc., 1962, p. 315.
10. A.D. Falkoff and K.E. Iverson, “The design of APL,” *SIGAPL APL Quote Quad*, vol. 6, no. 1, 1975, pp. 5-14; DOI 10.1145/585923.585925.
11. R.K.W. Hui, et al., “APL\?,” *SIGAPL APL Quote Quad*, vol. 20, no. 4, 1990, pp. 192-200; DOI 10.1145/97811.97845.
12. F.B. Gilberth and L.M. Gilbreth, *Process Charts*, The American Society of Mechanical engineers, 1921.
13. H.H. Goldstine, *The Computer from Pascal to von Neumann*, Princeton University Press, 1972.
14. D.D. Hils, “Visual languages and computing survey: Data flow visual programming languages,” *Journal of Visual Languages & Computing*, vol. 3, no. 1, 1992, pp. 69-101; DOI http://dx.doi.org/10.1016/1045-926X(92)90034-J.
15. J.V. Nickerson, “Visual Programming,” New York University, 1994.
16. M. Hirakawa, et al., “An iconic programming system, HI-VISUAL,” *Software Engineering, IEEE Transactions on*, vol. 16, no. 10, 1990, pp. 1178-1184; DOI 10.1109/32.60297.
17. D. Ladret and M. Rueher, “VLP: a visual logic programming language,” *Journal of Visual Languages & Computing*, vol. 2, no. 2, 1991, pp. 163-188; DOI http://dx.doi.org/10.1016/S1045-926X(05)80028-X.
18. J. Bertin, *Graphics and the Graphic Information Processing*, Walter de Gruyter Berlin, 1981.
19. J. Bertin, *Semiology of graphics*, University of Wisconsin Press, 1983.
20. G. Costagliola, et al., “Automatic generation of visual programming environments,” *Computer*, vol. 28, no. 3, 1995, pp. 56-66; DOI 10.1109/2.366162.
21. T.J. McCabe, “A Complexity Measure,” *Software Engineering, IEEE Transactions on*, vol. SE-2, no. 4, 1976, pp. 308-320; DOI 10.1109/TSE.1976.233837.

22. M.H. Halstead, *Elements of Software Science (Operating and programming systems series)*, Elsevier Science Inc., 1977, p. 128.