Philip Yeeles

Nico: An Environment for Mathematical Expression in Schools

Computer Science Tripos

Selwyn College

May 4, 2012

Proforma

Name:	Philip Yeeles
College:	Selwyn College
Project Title:	Nico: An Environment for Mathematical Expression in Schools
Examination:	Computer Science Tripos, May 2012
Word Count:	TBC[1] (well less than the 12000 limit)
Project Originator:	P. M. Yeeles (pmy22)
Supervisors:	Dr S. J. Aaron (sja55), A. G. Stead (ags46)

Original Aims of the Project

The aim of the project was to develop an application in the Clojure programming language which would allow users to express mathematical calculations using a graphical notation. The software was to be able to generate an abstract syntax tree from the graphical notation, evaluate it and pass the results back to the application in under 300ms. An extension to the project was to conduct a user study to evaluate the utility of the software.

Work Completed

I have successfully designed and implemented the application detailed in the previous section. That is, I have developed an application in which it is possible to express calculations using a graphical notation, that generates an abstract syntax tree from the language and that is able to parse the tree and return the results in

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

under 300ms. I have also conducted a user study to assess whether or not the software is actually of use with regard to mathematics education.

Special Difficulties

Learning the Clojure programming language.

Declaration of Originality

I, Philip Michael Yeeles of Selwyn College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date May 4, 2012

Contents

List of Figures

Acknowledgements

My thanks to Luke Church for his advice regarding user studies, and to Alistair Stead and Sam Aaron for their encouragement and patience.

Chapter 1

Introduction

The aim of this project has been to design and develop a notation and accompanying application to act as a learning aid for pre-algebra arithmetic by increasing visibility, reducing the number of hidden dependencies and making the flow of data obvious to the user. I have successfully developed such a system, intended initially for pupils in Year 5 (though extensible, through the creation of alternative question sets, to other age groups), and, as an extension, conducted a user study to assess its utility.

In this chapter, I will discuss my motivations for choosing this project, the pros and cons of the handwritten system it is attempting to augment, the technical challenges involved in developing such a system and related work that has previously been conducted with similar goals.

1.1 Motivations

My motivations behind this project lay in the limitations of the handwritten approach to solving mathematical problems that I had observed both in my own learning and in my own teaching experience. What follows is an evaluation of the pros and cons of the handwritten method of performing arithmetic calculations according to Blackwell and Green's "Cognitive Dimensions" framework [2], and a discussion of the properties a useful alternative notation should have.

1.2 Technical Challenges

Developing such an application comprises two main challenges: developing a backend that is capable of creating, storing, editing, deleting, evaluating, nesting and calculations, and a graphical, user-facing frontend that is able to render calculations into the devised notation, and allow the user to perform operations upon the notation that affect the underlying calculation.

I chose to use the Clojure language as it provided many features that would prove to be useful over the course of the application's development. As a dialect of LISP, Clojure is a homoiconic programming language – that is, a programming language in which code is represented as a data structure – which made passing around and performing operations upon calculations themselves, rather than just their results, considerably easier. A calculation can simply be represented as a piece of code, which can then be utilised as needed.

As the user experience is so crucial to the success of the application, it was also important that there be well-established GUI libraries available. Clojure runs on the Java Virtual Machine (JVM), which puts Java's considerable standard library at one's disposal, whilst still being able to program in a LISP. As I am familiar with Java and the Swing GUI libraries, it was advantageous to be able to leverage this knowledge in designing the application's interface.

1.3 Previous Work

There already exists a wide variety of educational software for mathematics, but much of this is in the form of "games", in which a series of mathematical problems to be solved is poorly disguised as a game – indeed, such problems would be more accurately said to be embedded into a game, rather than becoming the game themselves. Thus, the object becomes not to solve the problems, but to play the game that happens to surround the problems. Such software also does not often offer any means of solving the problems, other than the traditional pen-and-paper method (with a piece of paper next to the computer screen), or the mental approach. Hence, what the user is then presented with is essentially a game and a worksheet, awkwardly interleaved. In some cases, it is even possible for the user to simply press arbitrary buttons until they pass the questions, effectively removing the maths element of the game and replacing it with a series of short breaks in gameplay.

There also exist a few applications intended to represent calculations on a computer in novel ways. A relatively common approach to this has been to try to make on-screen calculations more like on-paper calculations. *Pi Cubed* takes this approach by trying to make complex calculations appear as they would be written in an exam or exercise book [6]. *Soulver*, conversely, tries to achieve this by simulating "back-of-the-envelope" calculations, whereby notes in English augment the calculation [7]. Another approach is that of the *Scrubbing Calculator* [12], which extends the *Soulver*-style environment by helping the user to solve equations by dragging values to increase and decrease them, showing how changing a value affects the overall result. Values can be linked by dragging a line between them, which means that they are two instances of the same value – hence dragging one changes the value at every location in which it appears. This is a neat means of visualising equations, but it, too, is not intended for use in education, and still requires the user to be able to formulate some kind of equation. The *Scrubbing Calculator* is more a tool for facilitating algebraic understanding, as opposed to arithmetic understanding; indeed, it is inherently a **calculator**, and so does not encourage thinking about how to work out the arithmetic parts of a calculation manually.

1.4 Summary

Existing educational "games" for mathematics either have too much focus on being a game, rather than helping to learn mathematics, or are such that the mathematical element is circumventable. There exists software to aid in calculation and arithmetic by representing it clearly, but it is not intended for educational use, and often its purpose is to make on-screen calculations appear as one would handwrite them.

There is a niche for a tool for use in education that represents calculations in a visual manner, with a particular focus on making the method by which arithmetic problems are solved clear. My project aims to provide an environment in which the user can explore the many ways in which a problem can be solved using a novel graphical notation.

Chapter 2

Preparation

This chapter concerns the work that was completed prior to beginning the project proper. It comprises a requirements analysis, followed by a discussion of the prototyping process of the graphical notation to be implemented in the final application. Finally, there will be a brief examination of the tools used in the development of the project.

2.1 Requirements Analysis

2.1.1 Current System

There are a number of problems with handwritten, pre-algebra arithmetic that this project seeks to rectify. First of all, the fact that it is handwritten entails a high level of viscosity: it is difficult to make changes to a written calculation without

(a)	(b)	(c)
24+35=59	24+35=59	24+35= ~~59~~ 49
12+48=60	12+48=60	12+48=60
59+72=131	59+72=131	~~59~~ 49+72= ~~131~~ 121
1+60=61	1+60=61	1+60=61
131+61=192	131+61=192	~~131~~ 121+61= ~~192~~ 182

Figure 2.1: Illustrating the hidden dependencies and viscosity inherent in pre-algebra, handwritten arithmetic. The original calculation is shown in (a), has its otherwise-hidden dependencies highlighted in (b), and is altered slightly in (c).

sacrificing clarity. In particular, there is a lot of repetition viscosity involved in the modification of an existing piece of work; if a number is changed that is used in several calculations, then it is time-consuming to change it everywhere it appears in the working. If several calculations are dependent upon each other, then this entails a lot of knock-on viscosity in recalculating each stage after changing the number. This is exacerbated by the hidden dependencies between chained calculations in handwritten arithmetic (*Fig. 2.1*). The problem of hidden dependencies is made worse by the low juxtaposability of the system; although the notation is quite visible, in that every calculation can be seen easily on the page, juxtaposing two sets of calculations entails considerable premature commitment on the part of the user, as components cannot easily be edited or relocated due to the system's high viscosity. Whilst viscosity can be acceptable in some situations, it is harmful with regard to modification and exploration within a notational system, once again requiring a non-trivial amount of premature commitment on the part of the user. In many cases, it can actually quicker for the user to start all over again, as opposed to making the changes required to rectify their calculations.

Handwritten arithmetic does have one key advantage: it has a very low initial abstraction barrier. Other than learning the appropriate symbols for each operation and digit, and how to combine them, the traditional system of arithmetic allows a learner to begin using it almost immediately. Algebra, on the other hand, has a much higher abstraction barrier, requiring the much more abstract concept of a variable, rather than a set quantity, to be used effectively. Although it is possible to add abstractions to arithmetic by the use of secondary notation, there is no provision for abstraction included in the primary notation. It can, therefore, be said that algebra is an *abstraction-hungry* system, whereas arithmetic is an *abstraction-hating* system. Without abstractions, arithmetic is easy to get started with, but can be a very verbose and inefficient notation with low visibility, as outlined above.

2.1.2 Proposed System

2.1.2.1 Overview

To improve upon the standard approach of listing the steps comprising a calculation, a system must acknowledge and try to overcome the drawbacks listed above. To this end, I have designed and developed a notational system and accompanying applciation that aims to overcome many of the disadvantages inherent in traditional, handwritten arithmetic. The intention of the system is to

reduce viscosity, increase visibility and to remove many of the hidden dependencies that beleaguer the traditional method.

2.1.2.2 Functional Requirements

To be an improvement upon the current system detailed above, the new system must satisfy the following properties:–

- Allows the user to create graphical structures representative of complex calculations
- Provides the user with a suitable environment in which to do so
- Allows the user to reposition elements of the structure at will
- Displays the user's current progress
- Is able to evaluate the correctness of the user's answer
- Accepts a file containing a set of questions to be answered
- Displays the current question being answered
- Progresses through the current question set as the user answers each question correctly

2.1.2.3 Non-Functional Requirements

The system must also satisfy a number of requirements outside of its basic functionality. These are listed below.

- Offers a significant improvement in visibility over handwritten arithmetic
- Offers a significant improvement in juxtaposability over handwritten arithmetic
- Reduces premature commitment relative to handwritten arithmetic
- Reduces hidden dependencies relative to handwritten arithmetic
- Is interactive: is able to pass results back to the user in less than 300ms[1]

[1]Roca and Rousseau [9] have this to say on the subject of interactivity: "An abundance of studies into user tolerance of round-trip latency [...] has been conducted and generally agrees upon the following levels of tolerance: excellent, 0-300ms; good, 300-600ms; poor, 600-700ms; and quality becomes unacceptable [...] in excess of 700ms."

- Is appealing to the target audience of 9- to 10-year-olds without being childish – must be applicable to a wider audience if needed (e.g. could it be extended for use in adult education?)

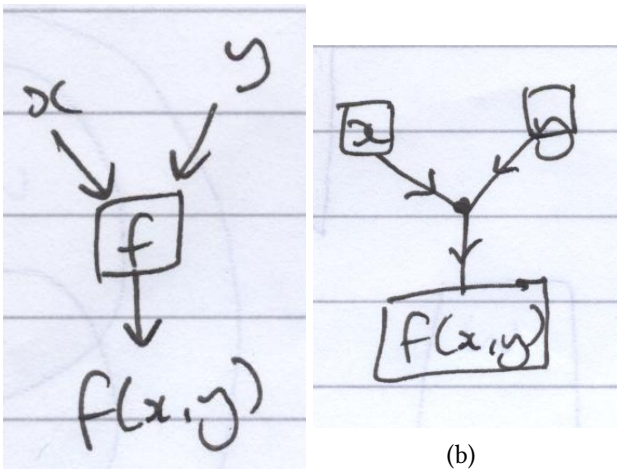
2.2 User Interface

As this project is primarily concerned with human-computer interaction, the user interface and experience, and the design thereof, constitutes a significant part of this project. As such, this section outlines the initial development stages of the user interface, first introducing several designs for the graphical notation, and then discussing in more detail three that were developed further.

2.2.1 Prototyping

2.2.1.1 Low-Fidelity Prototyping

To try to get some initial ideas for what could become the calculation metaphor of choice for the project, a target was set of devising at least twenty, significantly different, potential designs. These were recorded as very rough sketches, a few of which were refined in larger examples, and three of which were deemed good enough to warrant an application mockup, as detailed in *Sec. 2.2.1.2*.



(a) (b)

Figure 2.2: Early designs for flowgraph-based calculation metaphors.

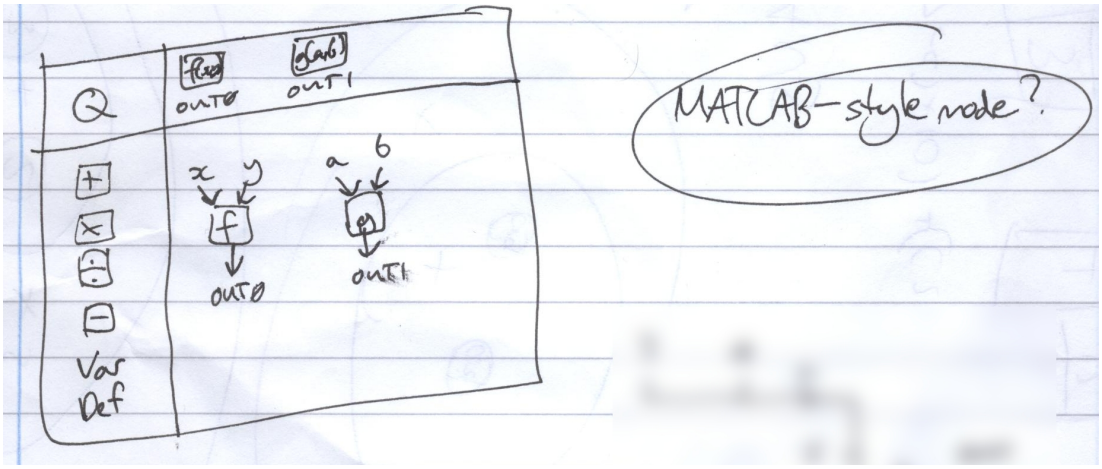


Figure 2.3: A more refined sketch of a flowgraph-based language and application.

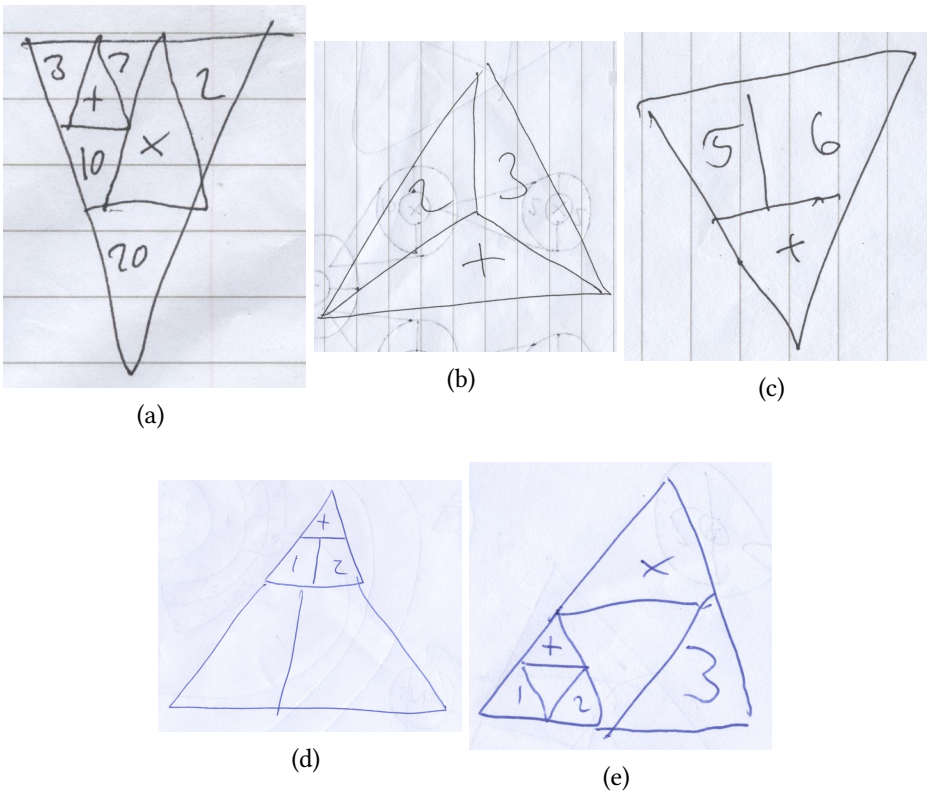


Figure 2.4: Early designs for triangle-based calculation metaphors.

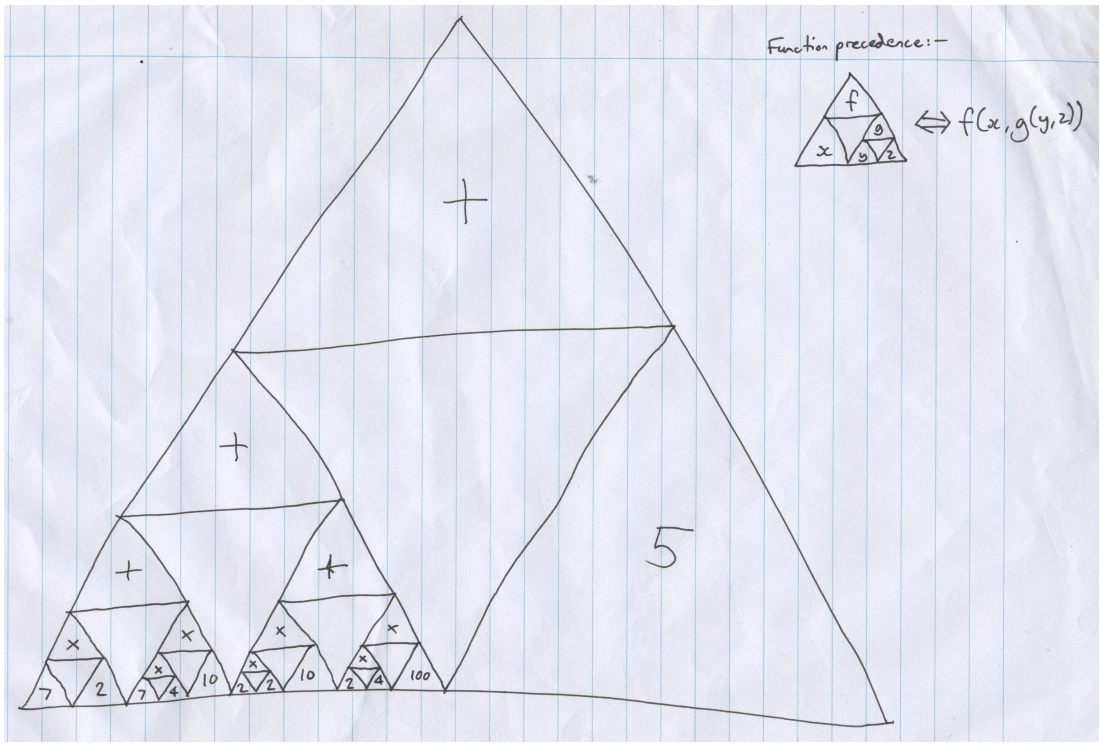


Figure 2.5: A more refined sketch of a triangle-based language.

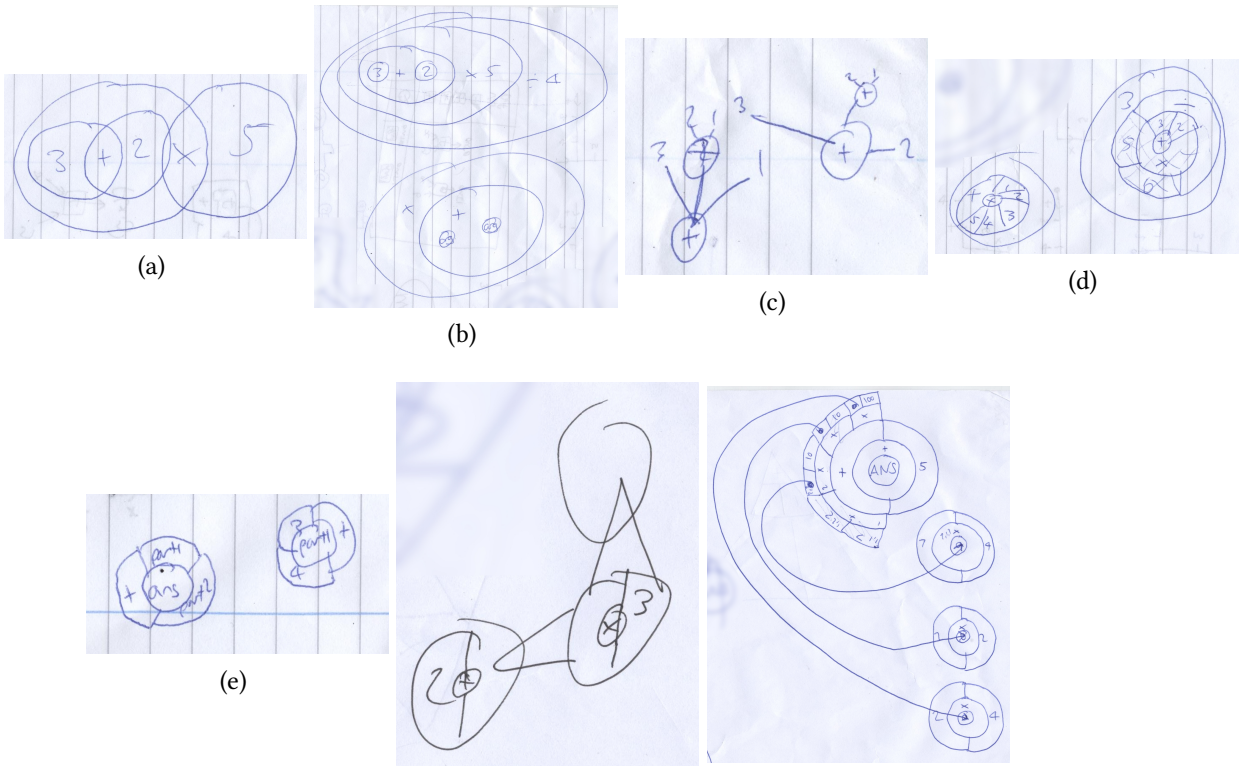


Figure 2.6: Early designs for circle-based calculation metaphors.

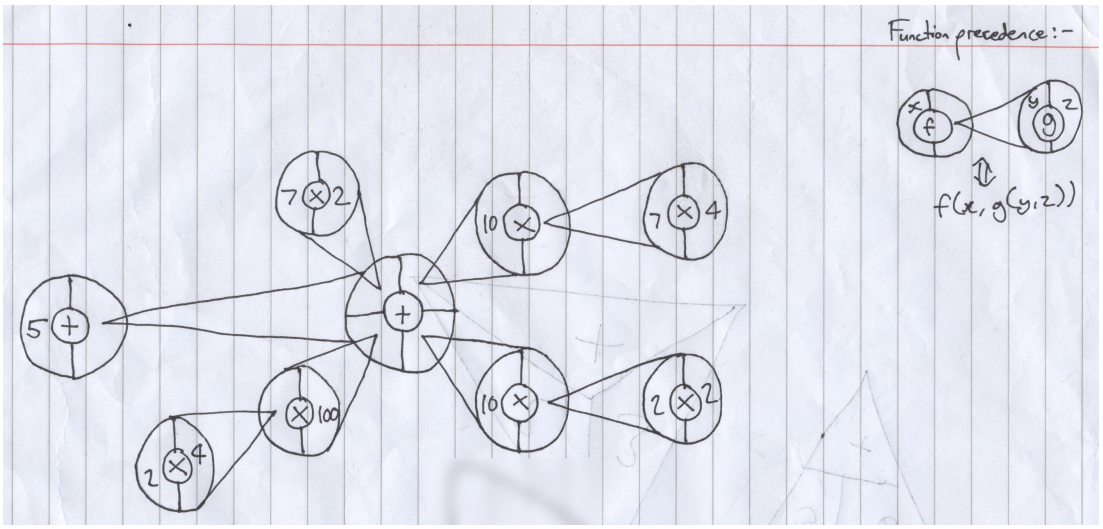
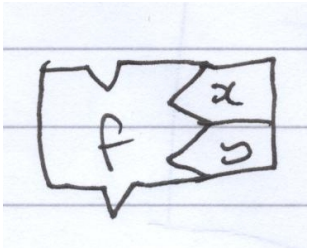
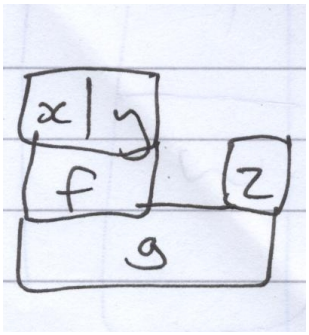


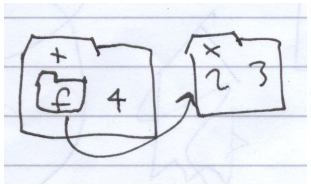
Figure 2.7: A more refined sketch of a circle-based language.



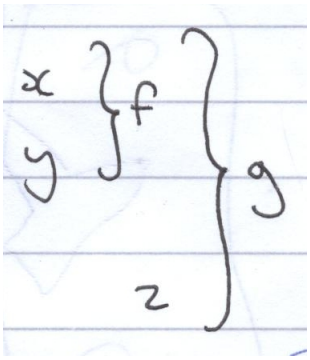
(a)



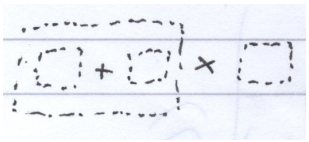
(b)



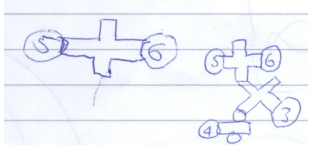
(c)



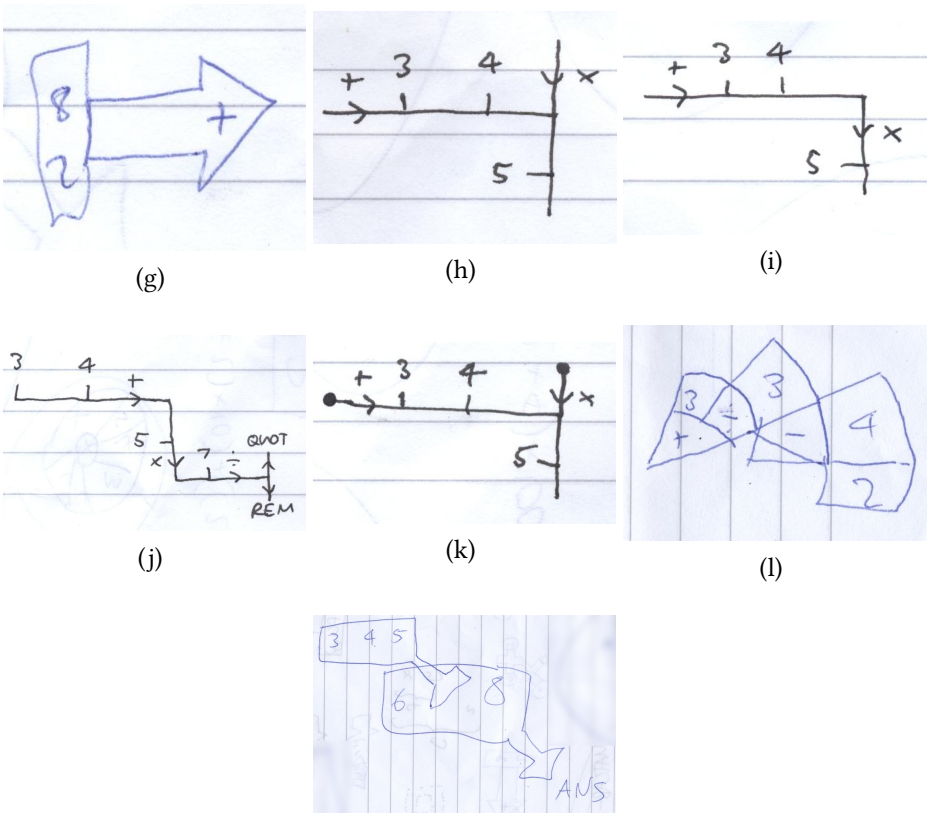
(d)



(e)



(f)

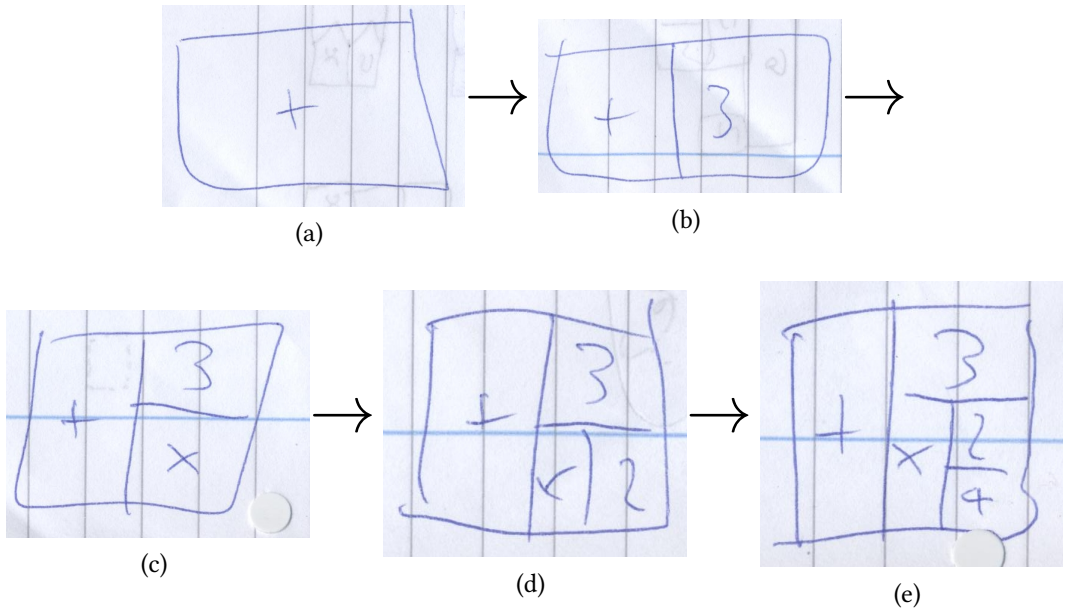


(g) (h) (i)

(j) (k) (l)

(m)

Figure 2.7: Assorted early designs for other calculation metaphors.



(a) (b)

(c) (d) (e)

Figure 2.8: A sample progression of a calculation in a language based around infinitely halving a rectangle.

2.2.1.2 Detailed Mockups

Some preliminary designs for the application and graphical language follow, being more mature versions of three of the low-fidelity prototypes presented above. First is a combined flowgraph representation and Read-Evaluate-Print-Loop (REPL) design. The second design uses the Sierpiński triangle as a basis for the visual metaphor. The final design is the circle-based language that eventually became the foundation for the rest of the project.

2.2.1.2.1 Flowgraph Metaphor

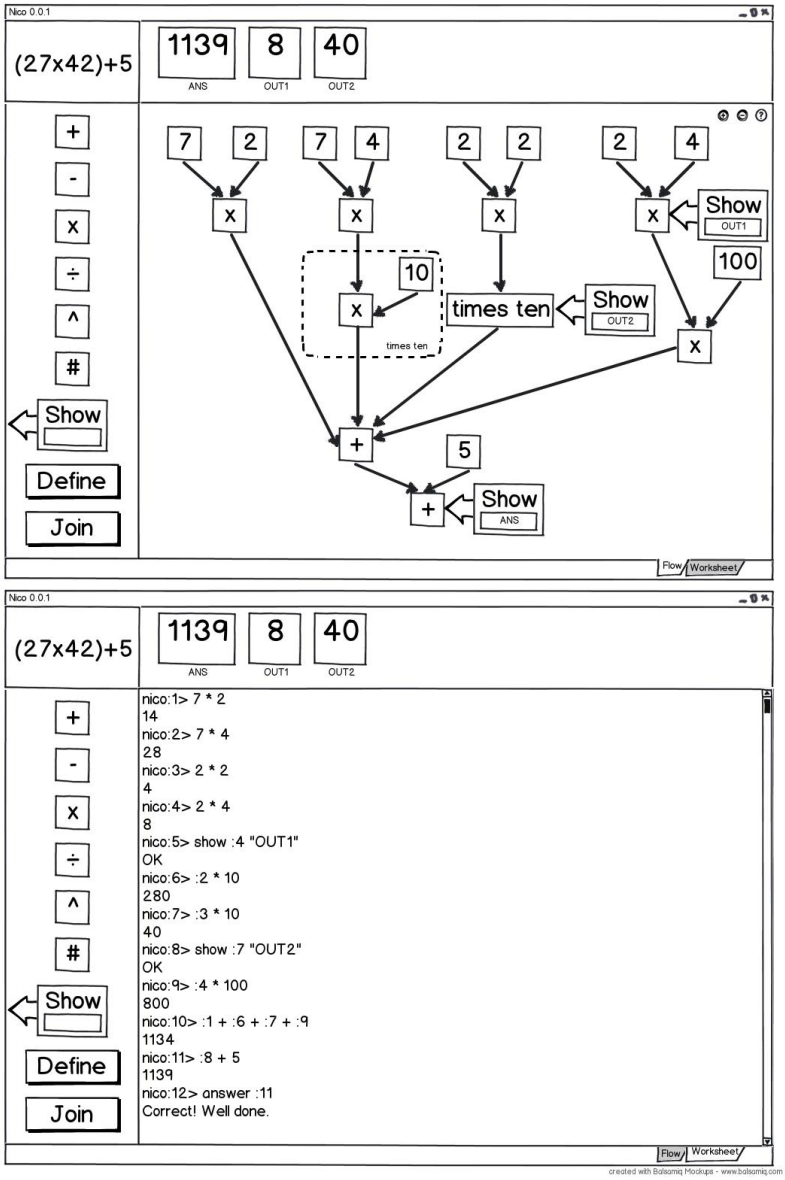


Figure 2.9: The prototype flowgraph-style language.

The flowgraph idea initially outlined in the project proposal and included in the low-fidelity prototypes above appears here as a more mature, revised prototype, showing two screens of a sample application. The top screen shows the proposed application manipulating the graphical language, whilst the bottom screen shows a REPL utilising a simple text-based language for users more comfortable with writing than the visual metaphor.

The application is comprised of one window containing several panels. The current

question being answered is displayed in the top left-hand corner. Below this, a scrollable panel running down the left side of the window contains the fundamental components of the notation: boxes representing numbers and operations, both provided and user-defined, with two buttons below to change the action of clicking the mouse (explained in greater detail below). A panel that runs along the top of the window contains output boxes, areas that display the result of a calculation at a user-determined point in the application. Further boxes appear here as the user demands them, by using the Show function. The ANS box is a specialised case of this, which displays what the user currently wishes to submit as the answer to the question. The canvas panel occupies the majority of the window, and this is the area in which the answer is constructed using the various components of the graphical language. In the top right-hand corner of the canvas are three buttons, to zoom in and out of the current calculation and to bring up a help window. For especially large or zoomed-in calculations, the canvas is scrollable. At the bottom of the window are two tabs, Flow and Worksheet. When Flow is selected, the graphical language is available for use in answering the question. When Worksheet is selected, the user is able to use a simple textual language in a REPL to answer the question.

Of all the prototypes shown here, this version of the graphical language is the most like a traditional programming language, in that it allows the user to define reusable functions, and includes functionality similar to print statements. The icons at the side of the application are used to select the desired function, from a set that includes addition, subtraction, multiplication, division, exponentiation, number input, Show, Define and Join. Addition, subtraction, multiplication, division and exponentiation are all dragged-and-dropped from the icons at the side to a point on the canvas, causing a box containing their respective functions to appear at that point. Number input (#) is similar, but the user inputs a number using the keyboard after the box is created. The box then contains that number. Show displays the output of the junction to which its arrow points, in the output box at the top of the screen with the label input by the user after placing it. If the box with the label does not yet exist, it is created next to the existing output boxes. Define and Join are not boxes to be dragged-and-dropped; they are modes of operation. When Define is activated, dragging the mouse on the canvas draws a box around sections of the diagram – the user is able to define functions by doing so, and by providing a label for the section of diagram that has been highlighted, a corresponding box can be dragged-and-dropped from the list of icons (as show in *Fig. 2.2* with the times ten function). When Join is active, dragging between two points on the canvas creates an arrow between them that means that the source of the arrow is used as an argument to the destination of the arrow. In defining functions, if arrows cross the

boundary of the definition box and they are incoming, input is required to future uses of the function. One outgong arrow is allowed to indicate the output of the function. Using this notation, answers are submitted by Showing the output at a point in the calculation to the output box `ANS`.

This particular graphical language tries to have a very high visibility, minimising the number of hidden dependencies between calculations by making every connection explicit: as a dataflow representation of a calculation [2], the flow of information through the diagram is made very clear. This language is considerably less viscous than the handwritten method, as it is trivial (albeit not demonstrated in *Fig. 2.2*) to remove and replace a box or arrow in the diagram, as opposed to writing out a new set of calculations or replacing several instances of a component by crossing them out. Each box is able to be relocated on the canvas, increasing the juxtaposability of the system, and the user is able to define their own abstractions.

The textual language improves upon the traditional means of handwriting arithmetic by revealing otherwise-hidden dependencies using references to line numbers. As the target audience is not yet required to have formally learnt (or, indeed, encountered) algebra, this notation includes a function `:`, which takes a single number *n* as an argument and returns the result of the calculation performed on line *n*. Answers are submitted using a function `answer` that takes a single number (shown here using a line reference that is resolved to a number) as an argument. Other functions include addition, subtraction, multiplication, division and exponentiation, all of which take two or more numerical arguments and are used in the familiar infix form. A function `define` is also included, although not shown here. Finally, there is the `show` function, which behaves similarly to a print statement. It takes a numerical argument *n* followed by a string argument `str`, and displays *n* in the output box named `str` at the top of the screen. If there exists no such output box, it is created next to the existing boxes. Note that `answer n` is logically equivalent to `show n "ANS"`; the `answer` function was included here to increase clarity.

This design was abandoned as it was deemed to be too complex for the target audience (Year 5). The intention of this project is not to implement yet another programming language for learners, and by providing features like the REPL and function definition, the system unnecessarily overprovides. In addition to this, the flowgraph representation had a number of shortcomings. For example, it is not immediately clear in what order the arguments feed into an operation, which is confusing for non-commutative functions such as subtraction. Furthermore, the language is quite verbose. Although this is not inherently a disadvantage, a more

compact representation would be preferable, in order to make expressions written in the notations more readable to people other than the author.

2.2.1.2.2 Sierpiński Triangle Metaphor

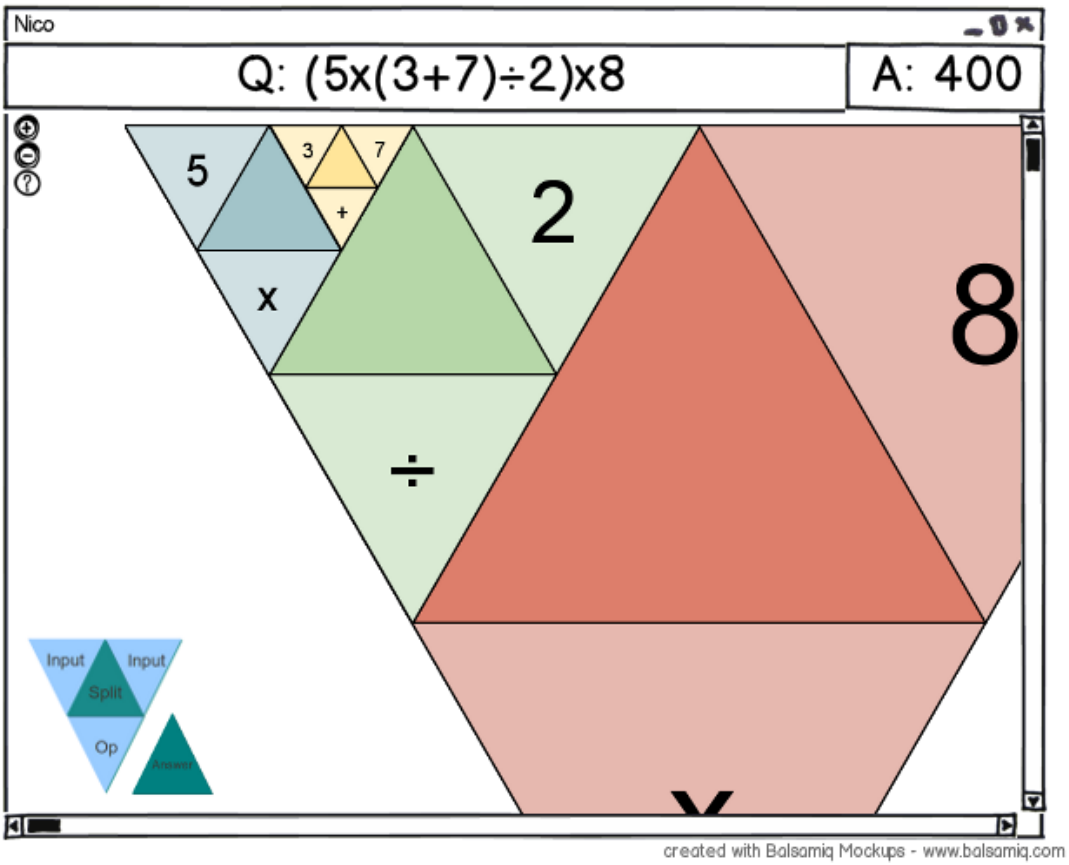


Figure 2.10: The prototype Sierpiński-triangle-based language.

This prototype uses a zooming instance of the Sierpiński triangle fractal as the basis of its notation. It is shown here manipulating a simple calculation.

The application consists of a single window containing two panels and a canvas area. The right-hand panel displays the current total of the calculation, and the other panel displays the question currently being answered. In the canvas area, there are buttons for zooming in and out, a help button and buttons controlling the composition of structures within the language. When a triangle has been selected by clicking on it, clicking the `Op` button presents the user with a choice of operations (namely +, -, × and ÷) to use in the bottom segment of the triangle.

Clicking either of the Input buttons allows the user to input a value into their respective sections, and clicking on the Split button creates a new triangle expression within the left-hand triangle.

This notational system is based around binary operations represented as the component triangles of a Sierpiński fractal, the logic behind this being that the fractal structure emphasises that each calculation is part of a larger whole. The notation is read by using the left and right sub-triangles as arguments (left first, right second) to the operation in the bottom triangle. The structure is colour-coded, such that each subexpression is a different colour to its precedent superexpression. This allows the user to abstract away subexpressions as units to be manipulated as any other input to an operation would be. The hidden dependencies of the handwritten method are greatly diminished in this manner, as viewing the internal structure of a calculation's subexpressions is simply a matter of zooming-in to the appropriate triangle. Viscosity is also considerably reduced, as making a change in one expression does not require the change to made anywhere else, as the results of that expression are passed to the rest of the calculation. This does not completely remove viscosity, as a misresrepresented value still has to be replaced several times if it appears in several different expressions, but this is still an improvement over handwriting. Indeed, the fact that calculations are editable more than once, by virtue of using a computer rather than paper, makes this less viscous than pen and paper. The system is quite visible, but has a very low juxtaposability, as all calculations are restricted to their respective places in the triangle template.

This design was ultimately abandoned as it was deemed to be awkward for a number of reasons, not least because it was based solely around binary operations, meaning that to solve the simple question 1+2+3+4, one has to either calculate ((1+2)+3)+4, which is inefficient, or mentally calculate that 1+2+3=6, and then calculate 6+4, which is obviously unacceptable for a system that intends to help the user express themselves mathematically. The severe lack of juxtaposability is also a problem: as each expression must fit into the larger structure, it is not possible to reorder them at will, without fundamentally altering the calculation being performed. It also enforces a top-down approach to calculation: one is required by the system to begin with the outermost calculation and work inwards. There is no provision for putting a larger triangle around an existing calculation. In addition to this, a somewhat-informal survey of potential notations was conducted, asking approximately ten subjects to answer some sample questions in this notation and the circle-based notation (below). Subjects found this notation less clear, and were much better able to complete questions correctly using the circle notation.

2.2.1.2.3 Circle Metaphor

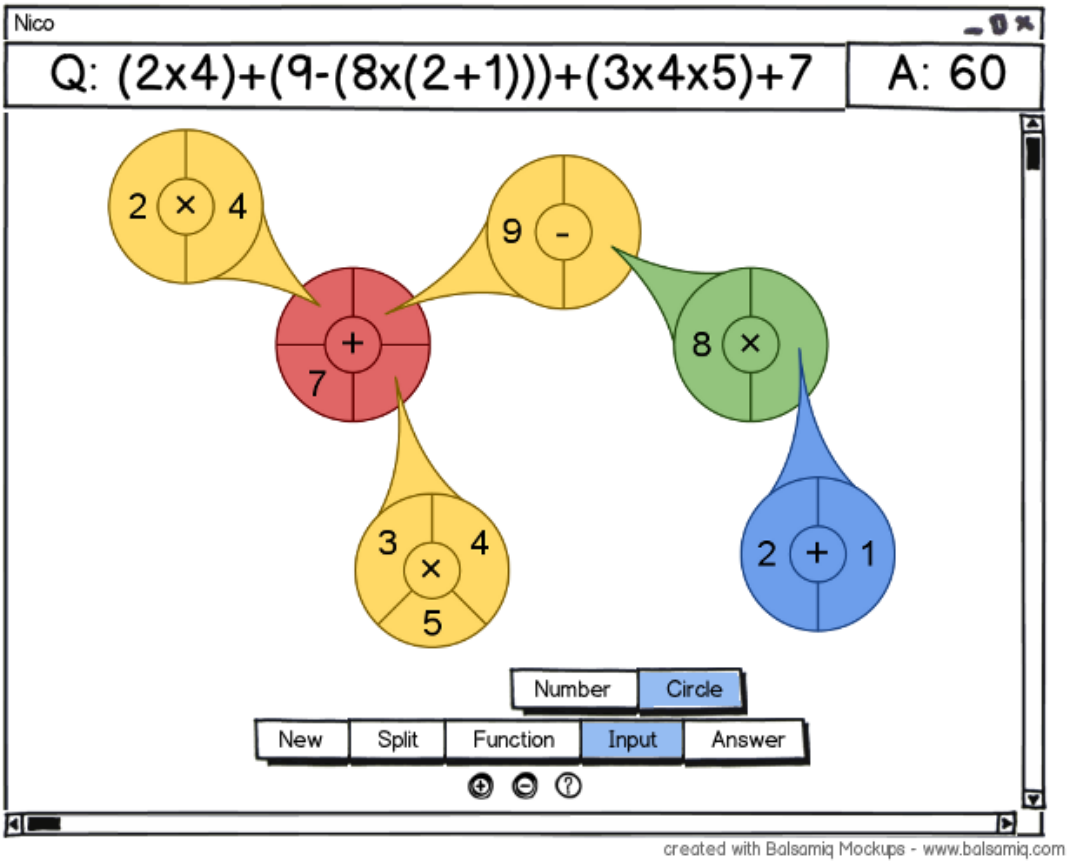


Figure 2.11: The prototype circle-based language.

This prototype uses a notation based around circles as individual units of calculation, linked in such a way as to indicate the flow of information. Although it is related to the flowgraph metaphor (above), it has a number of key differences.

The application comprises, similarly to the Sierpiński model above, a single window with two panels and a scrollable canvas area, with some buttons to control the input of data. The right-hand panel shows a running total of the user's calculation, and the other panel shows the question currently being answered. The canvas area has three buttons at the bottom: zoom in, zoom out and help. The larger buttons above these control the action that clicking the mouse on the canvas performs. When New is active, clicking on the canvas creates a new circle at that location. When Split is active, clicking on a circle increases the number of arguments it has (initially two, up to a maximum of eight). When Function is selected, clicking on a circle presents the user with a choice of +, -, × and ÷ to use as that circle's operation. When Input is

selected, as shown in *Fig. 2.4*, the user is presented with two further options, Number and Circle. When Number is active, clicking on a circle's argument allows the user to change the value of that argument. When Circle is active, the user is able to click and drag from one circle to an argument of another, indicating that they wish to use the results of one circle's calculation as the argument to another circle's calculation. Finally, the Answer button submits the current total as the answer to the current question.

This notational system uses linked circles representing individual expressions as its primary metaphor for constructing calculations. It is similar to the flowgraph metaphor presented above, except that instead of boxes for each argument and operation all linked together, single, linkable objects in this system are complete expressions, rather than components. Also, there is only one visible output, namely the overall result of the calculation as a whole. As for the circles themselves, they consist of an operation at the centre, orbited by arguments, which are evaluated clockwise from 0°. Each argument can either be a number or the tail of another circle, representing the result of the calculation that that circle stands for. The circles are colour-coded, with each 'layer' having its own colour. In this case, the root circle is red, the circles feeding directly into it are yellow, the circles feeding into those circles are green and the circles feeding into those are blue. It is possible to move circles around the canvas by clicking and dragging them to the desired location. Compared to the handwritten approach, this notational system is much more visible, and very much more juxtaposable, allowing the user to construct the diagram in a way that is comfortable for them, and allowing them to compare elements side-by-side if they need to. Unlike with the Sierpiński metaphor, the user is not restricted to one order or pattern of constructing a calculation: calculations can be performed bottom-up, top-down or a mixture of both, as is comfortable for the individual user. The system of tails feeding into other circles also makes the dataflow clear, eliminating many of the hidden dependencies of handwritten calculation. For similar reasons to the flowgraph metaphor, the circle notation also reduces viscosity, as changing a subexpression automatically alters all of its superexpressions. Finally, the system requires little premature commitment, as piece of the diagram can be created, deleted and swapped in and out at will, so it is ideal for exploration, allowing the user to try many ways of solving a problem within the application, rather than having to use a secondary notation or device to help work through a problem in a more familiar manner and then transcripting that solution into the new notation.

This design was the one that was ultimately chosen as the basis for the project as it neatly solved, or at least ameliorated, many of the problems listed with handwritten

arithmetic, without being too complex for the target audience (as with the flowgraph system), and without being too restrictive with regard to mathematical expression. In addition to this, the results of the survey mentioned above also indicated that people got on well with this notation; the test subjects found it considerably easier to use than the Sierpiński notation, and made fewer mistakes in handwriting this notation than the alternative.

2.3 Third-Party Tools

What follows is a list of the third-party tools that were used in the development of the project.

- Ubuntu Linux 10.04, Arch Linux 2010.05, Microsoft Windows 7
- Clojure 1.2.0
- Leiningen 1.6.1.1
- OpenJDK 6
- Seesaw 1.3.1-SNAPSHOT
- swank-clojure 1.3.4-SNAPSHOT
- GNU Emacs 23.1.1
- A modified version of Overtone's Emacs configuration [10], including:–
  - SLIME/SWANK (revision as of 15/10/2009)
  - clojure-mode 1.11.5
  - undo-tree 0.3.3
- Git 1.7.0.4
- GitHub
- Balsamiq Mockups
- Google Docs
- R 2.15.0

2.4 Summary

In this chapter, the work done in preparation for beginning the main implementation of the project was detailed. A thorough requirements analysis has been conducted to outline the expected behaviour of the product, contrasting it with the existing system of handwritten calculations. In *Sec. 2.2*, prototype designs for a suitable graphical notation were introduced. Three in particular were explored in detail, before deciding on the final notation upon which to base the project. The next chapter contains a detailed analysis of the work completed to implement the project.

Chapter 3

Implementation

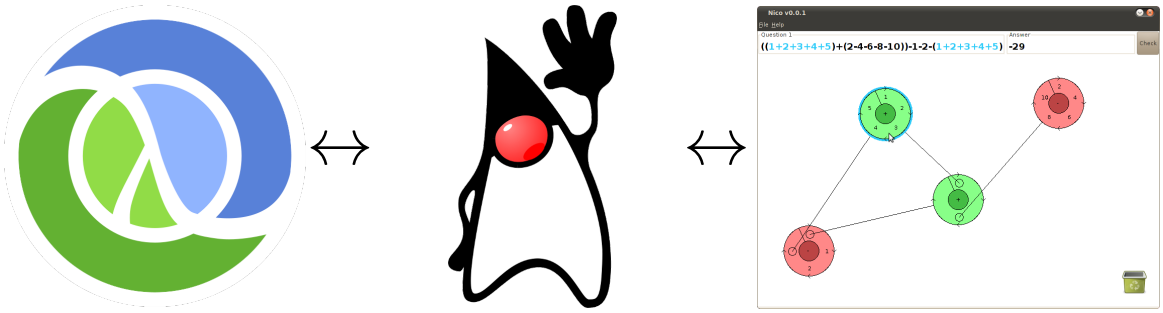
This chapter details the implementation of the project. The code itself totals approximately 1,500 lines of Clojure, and the resultant application, *Nico*, satisfies the requirements laid out in the project proposal, namely:–

> "[to be] able to generate an abstract syntax tree in Clojure from the graphical language and evaluate such a tree, passing the results back to the graphical application and displaying this to user in less than 300ms."

In addition to this, an extension to the project was completed, in the form of a study that tested the software on real people.

3.1 System Architecture

NOTE: *Bit stuck with this chapter; I feel like I'm just writing down arbitrary bits of things that happened during the project without any overall feel for how this fits together as a chapter.*



(a) Clojure backend (b) Swing/Seesaw interaction handler (c) User interface

Figure 3.1: An overview of the system architecture. *Nico* is divided into three logically-separate units: the user-facing application, the interaction handler and the backend.

NOTE: *Not sure about the ordering of these subsections; this may change.*

3.1.1 Backend

The application backend handles the mathematical aspect of the application; that is, it is able to interpret data gathered from the user's input and process it as a calculation. Each circle is represented by an associative map, of the form `{:x 123 :y 234 :name "c0" :circ (+ 1 2 3 4)}`, where the `:x` and `:y` fields are numbers representative of the circle's co-ordinates on the canvas, the `:name` field is a string used to refer to the circle and the `:circ` field is an S-expression representative of the calculation that the circle contains. This S-expression can use any of five operators: `+`, `-`, `*`, `/` and finally `\?`, a placeholder for initialising circles. Similarly, the arguments to the operator can be either numbers or the placeholder character `\?`. A Clojure agent, `used-circles`, stores a list of all the circles currently in use.

To determine the current value of the user's calculation, two functions are used: `find-root` and `eval-circle`.

`find-root` finds the circle that is not used as an argument to any other circle. If there are two circles that match this criterion, `find-root` returns the first instance that it encounters; that is, the first instance of a 'root' circle. As a complete calculation must only have one root circle (else it would be two separate calculations), this is not problematic for answer-checking. However, it can be mildly confusing for the user if they are constructing a calculation by constructing two subexpressions and merging them at the end, and a future improvement to this design could be to find the root of the calculation that the mouse is hovering over.

Regardless, `find-root` works by iterating across `used-circles` and returning the first circle that causes another function, `root?`, to return the value `true`. `root?` iterates across `used-circles`, comparing its single argument, a circle, to each of the other available circles in turn using another function, `is-arg?`, which takes the `:name` field of its first argument and returns `true` if that name appears as a symbol anywhere in the `:circ` field of its second argument.

`eval-circle` takes a circle as its argument and returns an S-expression representative of the calculation that that circle represents, recursively evaluating any circles that are contained within the argument circle. It works by iterating across the `:circ` field of its argument, consing each element on to a new list. If an element is of type `clojure.lang.Symbol`, `eval-circle` calls `eval-circle` on the circle that the symbol points to, which is resolved using a function `find-circle` that iterates across `used-circles`, trying to match each circle's `:name` field against its argument. `eval-circle` is then able to return a nested S-expression that represents a calculation. When used in conjunction with `find-root`, this function is able to resolve the entire diagram that has been constructed on the canvas into a single S-expression, which can then be evaluated or otherwise utilised as needed.

```
(defn eval-circle
  "Iterates across a circle list, resolving
   symbols into their respective circles."
  [circ]
  (loop [c (:circ (remove-placeholders circ))
         out '()]
    (cond (empty? c) (reverse out)
          (= (first c) \?) 0
          (symbol? (first c)) (cond
                               (nested?
                                 (find-circle
                                   (str
                                     (first c))))
                                 (recur (rest c)
                                        (cons (eval-circle
                                                (find-circle
                                                  (str
                                                    (first c))))
                                              out))
                               :else (recur (rest c)
                                            (cons (:circ
                                                    (find-circle
                                                      (str
                                                        (first c))))
                                                  out)))
          :else (recur (rest c)
                       (cons (first c)
                             out)))))
```

Figure 3.2: The `eval-circle` source.

Nico's backend stores and manages circles and questions using five Clojure agents: `used-circles`, containing a list of associative maps as detailed above, `current-qset`, also containing a list of associative maps, this time representative of the questions to be answered (using a format of `{:n 1 :q (* 2 (+ 1 2 3) 7) :a? false :c? false}`, where `:n` is the question number, `:q` is an S-expression representing the current question, and `:a?` and `:c?` are artefacts left over from a previous revision of the software, and are no longer used), `circle-number`, containing an integer counting how many circles have previously been created whilst answering this question, `current-coords`, which tracks the current co-ordinates of the mouse cursor, and `current-question`,

containing an integer indicating the number of the question currently being answered.

`used-circles` is an agent containing a list of associative maps, as detailed above. *Nico* includes two main functions for the management of circles, namely `new-circle` and `del-circle`. `new-circle` creates an associative map initialised to the following values:

```
1 {:x (- (:x @current-coords) 50)
2  :y (- (:y @current-coords) 50)
3  :name (str "c" @circle-number)
4  :circ (\? \? \?)}
```

Figure 3.3: The function `new-circle` initialises a circle map centred on the current location of the mouse cursor, generating a name `c`*n*, where *n* is the number of circles that have previously been created and with an initial expression of `(\? \? \?)`.

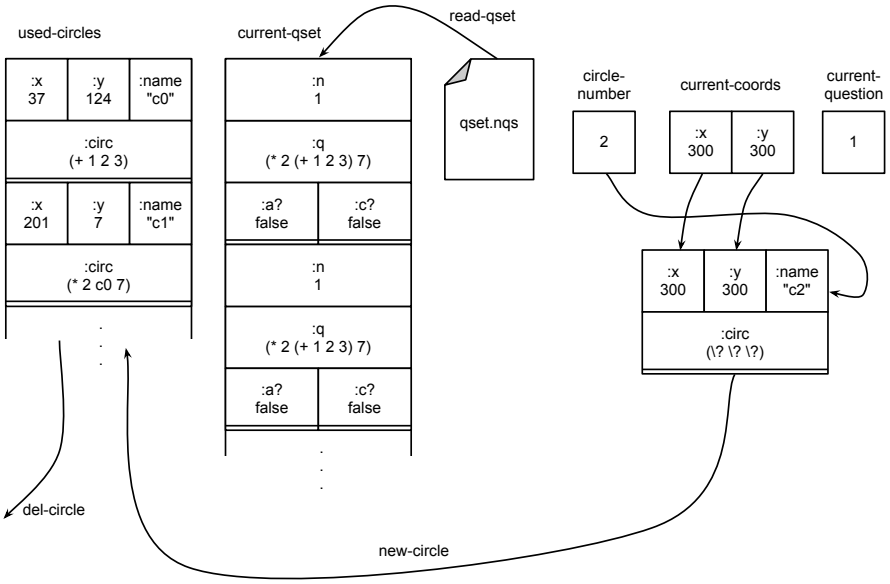


Figure 3.4: A rough model of *Nico's* management of mutable state.

3.1.2 Interaction Handler

NOTE: *These paragraphs aren't in the correct order yet; I've just been writing things that I want to include in this subsection, and will reorder them when I have a better idea of where this bit's going.*

My original intention was to develop *Nico* using the JavaFX library. However, a number of difficulties were encountered with this: firstly, JavaFX version 2.0 was

not available for Linux, my primary development environment, at the time of beginning the project. Although I had access to Windows machines, this was not an ideal situation. Regardless, I tried to begin development on a Windows machine, using JavaFX, but faced a number of problems with installation. Deciding that it would be better to get on with the project rather than spending any more time installing a library, I decided to proceed using the Eclipse Foundation's SWT toolkit and Szymon Witamborski's (santamon) corresponding Clojure bindings, GUI FTW!, to interface with it. After spending some time developing the backend, I began work on the user interface. I encountered many problems with SWT and GUI FTW!, particularly regarding drawing arbitrary shapes on the canvas area and regarding extracting user input from dialogue boxes. Admittedly, this was most likely due to my unfamiliarity with SWT, but in the interests of keeping the project on-schedule, I decided to migrate what I had already constructed of the interface over to Java Swing, using David Ray's (daveray) Seesaw library to provide convenient bindings for Clojure. Ray's excellent bindings, coupled with my own previous experience with Swing from earlier in the tripos, served to greatly assist development, overcome what had become one of the larger stumbling blocks in this project.

At the heart of the interaction handler is a function called `render`. This is the function that is called every time what is displayed on the canvas needs to be updated. It calls `clear-screen`, which paints a white rectangle over the canvas, followed by the bin icon, and then iterates across `used-circles`, calling a function `draw-circle` on each circle in turn. `draw-circle` determines what operation its argument circle uses, and selects a colour scheme accordingly. It then contains a sequence of paint instructions that construct the circle on the canvas. Finally, it calls `draw-args`, followed by `link-circles`. `draw-args` removes the first element of its argument's `:circ` field and draws each of the remaining elements at hardcoded co-ordinates relative to the `:x` and `:y` values of the circle, with different sets of co-ordinates according to how many arguments the circle has. `link-circles` uses data from the same sets of co-ordinates to determine which argument the user is trying to replace with input from another circle, and draws a line from underneath the source circle to the desired point on the destination circle, followed by a small circle around that point to make it clear that this circle is receiving the results of the other circle.

NOTE: *This is as far as I've got (about halfway by my reckoning).* `texcount` *tells me I've got about 5,800 words.*

3.1.3 User Interface

3.2 Application

3.2.1 Notation

3.3 Summary

Chapter 4

Evaluation

4.1 Backend Testing

4.2 UI Evaluation

4.3 Language Evaluation

4.4 User Study

4.5 Goals

4.6 Summary

Chapter 5

Conclusions

Bibliography

[1] R. Abraham. The Trouble with Math. *Educating the Whole Child for the Whole World: The Ross School Model and Education for the Global Era*, pages 125–137, 2010.

[2] A. F. Blackwell and T. R. G. Green. Cognitive dimensions of information artefacts: a tutorial, 1998.

[3] A. F. Blackwell and T. R. G. Green. Does metaphor increase visual language usability?, 1999.

[4] A. F. Blackwell and T. R. G. Green. A cognitive dimensions questionnaire optimised for users, 2000.

[5] A. F. Blackwell and T. R. G. Green. Notational systems – the cognitive dimensions of notations framework. *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*, 2003.

[6] Sunset Lake Software LLC. Pi Cubed | Sunset Lake Software. `http://www.sunsetlakesoftware.com/picubed`, 2012.

[7] Acqualia Software Pty. Ltd. Soulver | Acqualia. `http://www.acqualia.com/soulver/`, 2011.

[8] B. A. Myers. Taxonomies of visual programming and program visualization. *Journal of Visual Languages and Computing*, 1:97–123, 1990.

[9] A. Roca and F. Rousseau. Interactive multimedia and next generation networks. In *MIPS 2004: Second International Workshop on Multimedia Interactive Protocols and Systems (LNCS 3311)*, Grenoble, France, November 2004. ACM.

[10] samaaron and jlr. overtone/live-coding-emacs. `https://github.com/overtone/live-coding-emacs`, 2011.

[11] B. Victor. Kill Math. `http://worrydream.com/KillMath/`, 2011.

[12] B. Victor. Scrubbing Calculator.
http://worrydream.com/ScrubbingCalculator/, 2011.

Appendix A

Project Proposal

The original project proposal follows.

Computer Science Part II Project Proposal

Nico: An Environment for Mathematical Expression in Schools

P. M. Yeeles, Selwyn College
Originator: P. M. Yeeles
18 October 2011

Special Resources Required

- PWF account
- SRCF account
- GitHub account
- Toshiba Satellite L500-19X (Intel Pentium T4300 2.0GHz, 4GB RAM, 500GB disk)
- Samsung NC10 Plus (Intel Atom 1.66GHz, 1GB RAM, 250GB disk)

Project Supervisors: Dr S. J. Aaron & A. G. Stead
Director of Studies: Dr R. R. Watts
Project Overseers: Dr J. A. Crowcroft & Dr S. Clark

Introduction

Discussions with local teachers have led me to hypothesise that educational software for mathematics could be used to reinforce learning by focussing on method, rather than on a numerical answer. My aim is to develop a problem-solving system aimed at pupils in year 5 in which the solution to a problem can be represented as a tree of operations – a block-based graphical language to describe mathematical method. The correctness of the solution is then assessed with respect to the structure of the tree. The application will be written in Clojure, using JavaFX 2 for the graphical elements, though if this becomes infeasible I will use either the Eclipse SWT or Swing with GUIFTW. This dissertation will determine whether Nico offers an improvement regarding pupils' ability to recall the correct method for answering mathematical problems. The success of the project will be gauged by whether or not the software is able to generate an abstract syntax tree in Clojure from the graphical language and evaluate such a tree, passing the results back to the graphical application and displaying this to user in less than 300ms[1]. As an extension, I will distribute Nico with anonymous feedback forms to local schools, to determine if the software is actually of use in the classroom.

Work that has to be done

The project breaks down into the following sections:–

1. Core system
    a. A syntax for questions and a means of loading them
    b. A set of basic functions available to the student
    c. A means of inputting an answer that can be evaluated on-the-fly
    d. A means of re-expressing the question to reflect how the student works (e.g. $12 \times 34 \Rightarrow (10 \times 34) + (2 \times 34)$)
    e. A method of validating the answer
    f. A means of tracking the current result of evaluating the method input so far
    g. A system of hints for students who may not know where to start
2. GUI
    a. A collection of drag-and-drop elements that can be used to construct a diagram representing how to solve the question

[1] *Interactive multimedia and next generation networks: Second International Workshop on Multimedia Interactive Protocols and Systems, MIPS 2004 Grenoble, France, November 2004, Proceedings (LNCS 3311)* by Roca and Rousseau has this to say on interactivity: "An abundance of studies into user tolerance of round-trip latency [...] has been conducted and generally agrees upon the following levels of tolerance: excellent, 0-300ms; good, 300-600ms; poor, 600-700ms; and quality becomes unacceptable [...] in excess of 700ms."

   b. A means of validating combinations of the drag-and-drop elements
   c. A means of defining functions
   d. A means of viewing documentation
3. Evaluation
   a. Test software on non-technical but mathematically-able subjects
   b. Evaluate the correctness of Nico's translations between diagram and code
4. Extensions
   a. Create and distribute questionnaires to test classes
   b. Collect and interpret data
   c. Create a tutorial mode for new users

Difficulties to Overcome

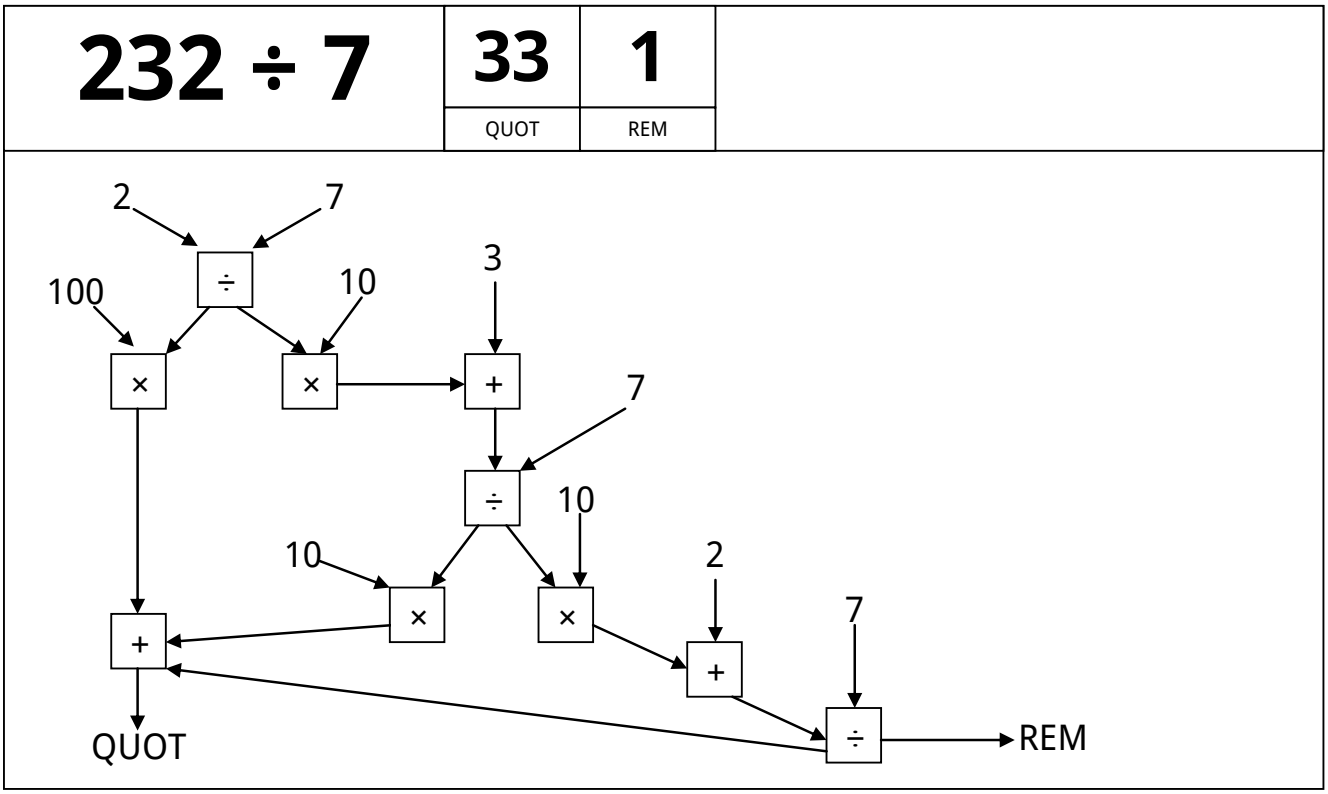
The following main learning tasks will have to be undertaken before the project can be started:

- Learn Clojure
- Become familiar with JavaFX 2
- Spend time designing the GUI and language

Starting Point

I have spent some months learning Clojure, and continue to do so. I have a good working knowledge of Java and experience teaching Mathematics and IT in Years 4 to 6. The first two years of the undergraduate course have familiarised me with Java and its libraries, and thanks to Clojure's interoperability I will be able to leverage these skills for this dissertation. My experience in schools has allowed me to develop the idea for this dissertation, and has given me an insight into what resources are useful in the classroom.

Below is a mockup of what I aim for Nico to look like. Notice how the method is expressed in the form of a flowchart, with outputs (in this case two) for the answer. Arrows show the direction of input and output, and the QUOT and REM boxes show the result of evaluating the functions being passed to them. Ideally, the question "232 ÷ 7" would also change to reflect how the student breaks down the question.



The above solution would auto-generate the following abstract syntax tree represented in Clojure code:–

QUOT is the output of:–

```
(+
  (*
    100
    (:quot
      (div
        2
        7)))
  (*
    10
    (:quot
      (div
        (+
          (*
            (:rem
              (div
                2
                7))
            10)
          3)
        7)))
  (:quot
    (div
      (+
        (*
          (:rem
            (div
              (+
                (*
                  (:quot
                    (div
                      2
                      7))
                  10)
                3)
              7))
            10)
          2)
        7)))
```

REM is the output of:–

```
(:rem
  (div
    (+
      (*
        (:rem
          (div
            (+
              (*
                (:rem
                  (div
                    2
                    7))
                10)
              3)
            7))
          10)
        2)
      7))
```

This assumes that we have a function `div` that takes two arguments `x` and `y` and returns an associative map `{:quot q :rem r}` such that `q` is the quotient of $x \div y$ and `r` is the remainder. Such a function will be included in the basic functions available to the user. Other functions of use would be addition, multiplication, subtraction, exponentiation, function definition and commenting (i.e. labels that are not evaluated), with options available in the question syntax (e.g. `:inhibit+ true`) to restrict arguments to a value of less than or equal to 10 (useful, for example, in questions on long multiplication, to prevent the student from simply giving `(* a b)` as the answer to $a \times b$). Hence a possible means of representing the question above could be:–

```
{:title “232 ÷ 7”
 :topic “arithmetic”
 :answer {:quot 33
          :rem 1}
 :inhibit+ false
 :inhibit- false
 :inhibit* false
 :inhibitdiv true}
```

Resources

This project requires little file space so my Toshiba PC’s disk should be sufficient. I plan to use the same PC as well as my Samsung PC to work on the project, and to back my files up to the PWF, the SRCF and GitHub. I will be using Git for version control.

Work Plan

Planned starting date is 27/10/2011.

October 2011

27/10/2011 - 10/11/2011
Work begins. Start covering the problems outlined in *Difficulties to Overcome*. Design the look and feel of the language and application.

November 2011

10/11/2011 - 24/11/2011
Design the question syntax. Implement the question interpreter. Implement the tree evaluator.

24/11/2011 - 08/12/2011
Implement the hints system. Begin work on the GUI.

December 2011

08/12/2011 - 22/12/2011
Finish the non-language section of the GUI. Begin implementing the graphical language.

29/12/2011 - 12/01/2012
Finish implementing the graphical language and its interpreter.

January 2012

12/01/2012 - 26/01/2012
Finish coding the core project. Begin extension work and evaluation.

26/01/2012 - 09/02/2012
Progress report written to be handed in by 03/02/2012. Preparation for presentation on 09/02/2012.

February 2012

09/02/2012 - 23/02/2012
Finish evaluation. Begin drafting the dissertation.

23/02/2012 - 08/03/2012
Finish extension work. Continue drafting the dissertation and evaluate extension work.

March 2012

08/03/2012 - 22/03/2012
Submit first draft of dissertation to supervisors by 16/03/2012. Begin redrafting on receipt of feedback.

22/03/2012 - 05/04/2012
Continuing redraft and resubmission of dissertation.

April 2012

05/04/2012 - 19/04/2012
Continuing redraft and resubmission of dissertation.

19/04/2012 - 03/05/2012
Dissertation complete 01/05/2012.

May 2012

03/05/2012 - 18/05/2012

Dissertation complete. Final edits, corrections. Binding and submission.