Philip Yeeles

Nico: An Environment for Mathematical Expression in Schools

Computer Science Tripos

Selwyn College

May 7, 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 7, 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 [7]. *Soulver*, conversely, tries to achieve this by simulating "back-of-the-envelope" calculations, whereby notes in English augment the calculation [8]. Another approach is that of the *Scrubbing Calculator* [13], 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 [10] 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?)

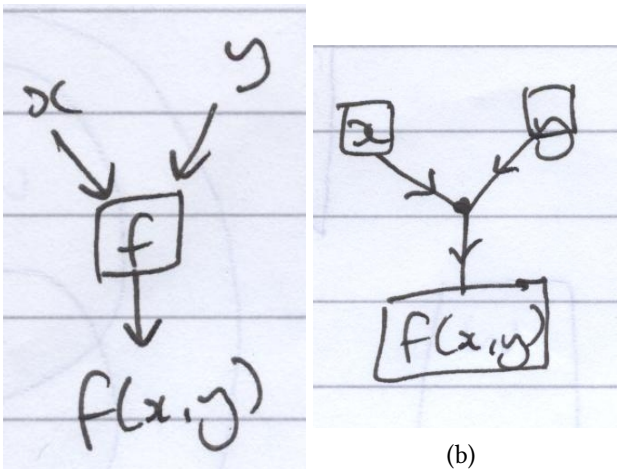
2.2 User Interface

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.

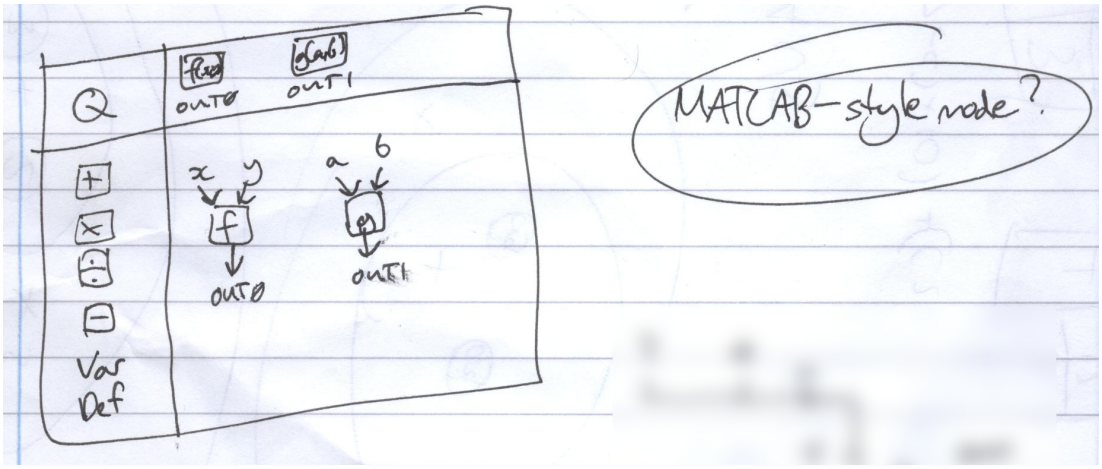


Figure 2.3: A more refined sketch of a flowgraph-based language and application.

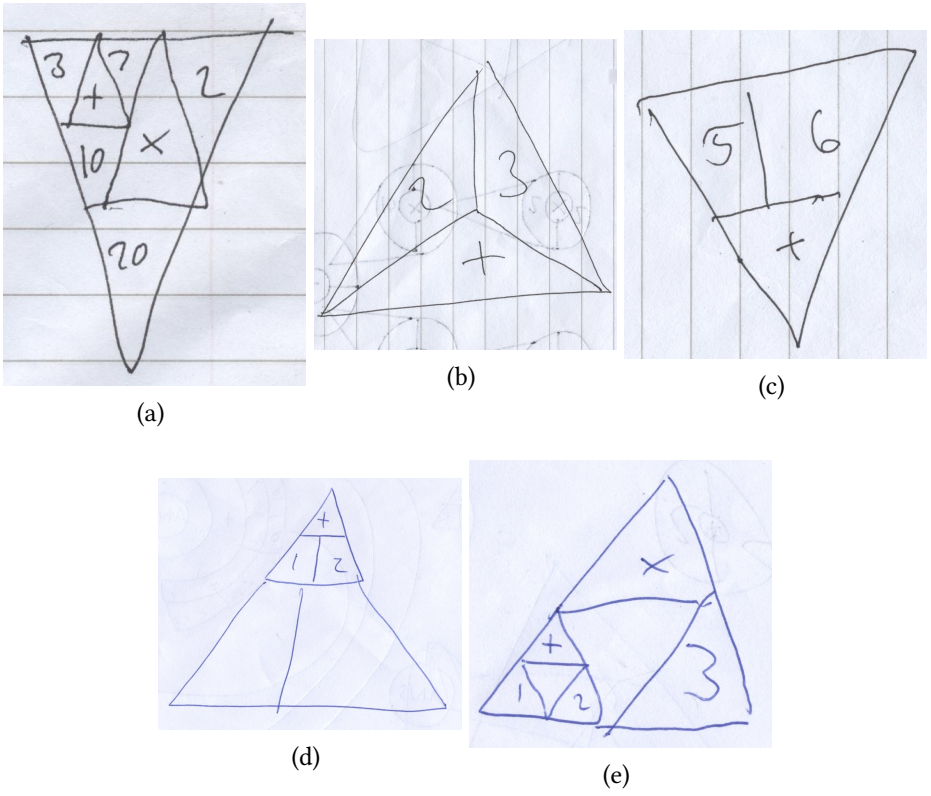


Figure 2.4: Early designs for triangle-based calculation metaphors.

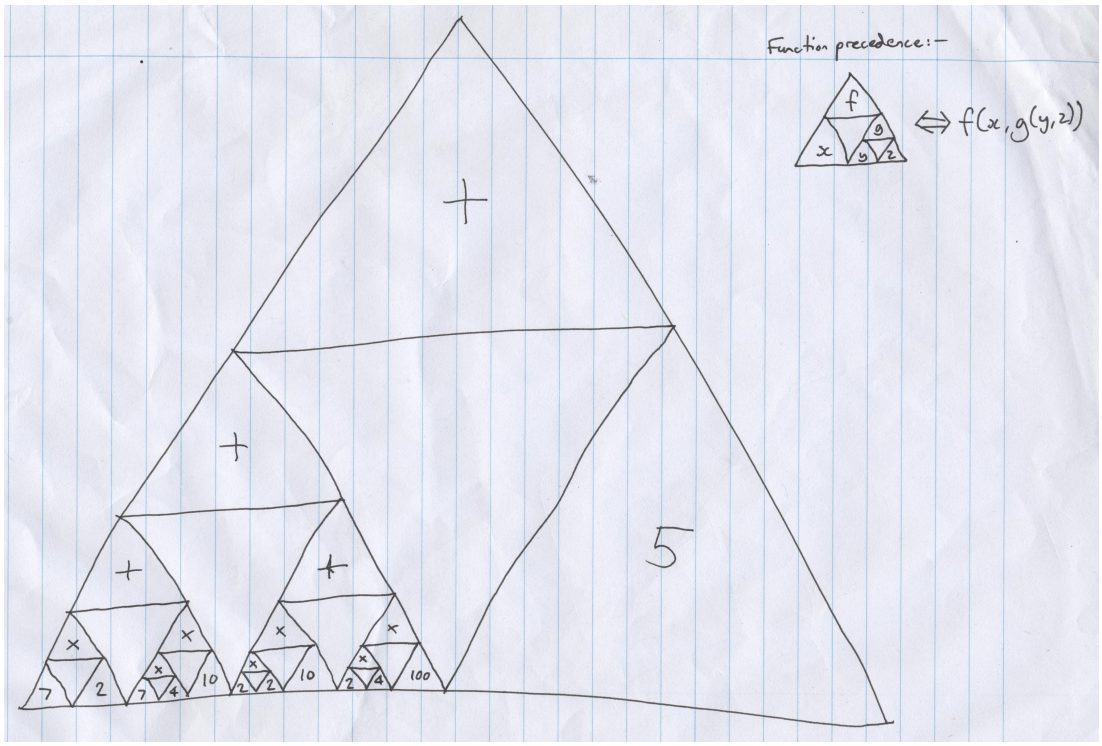


Figure 2.5: A more refined sketch of a triangle-based language.

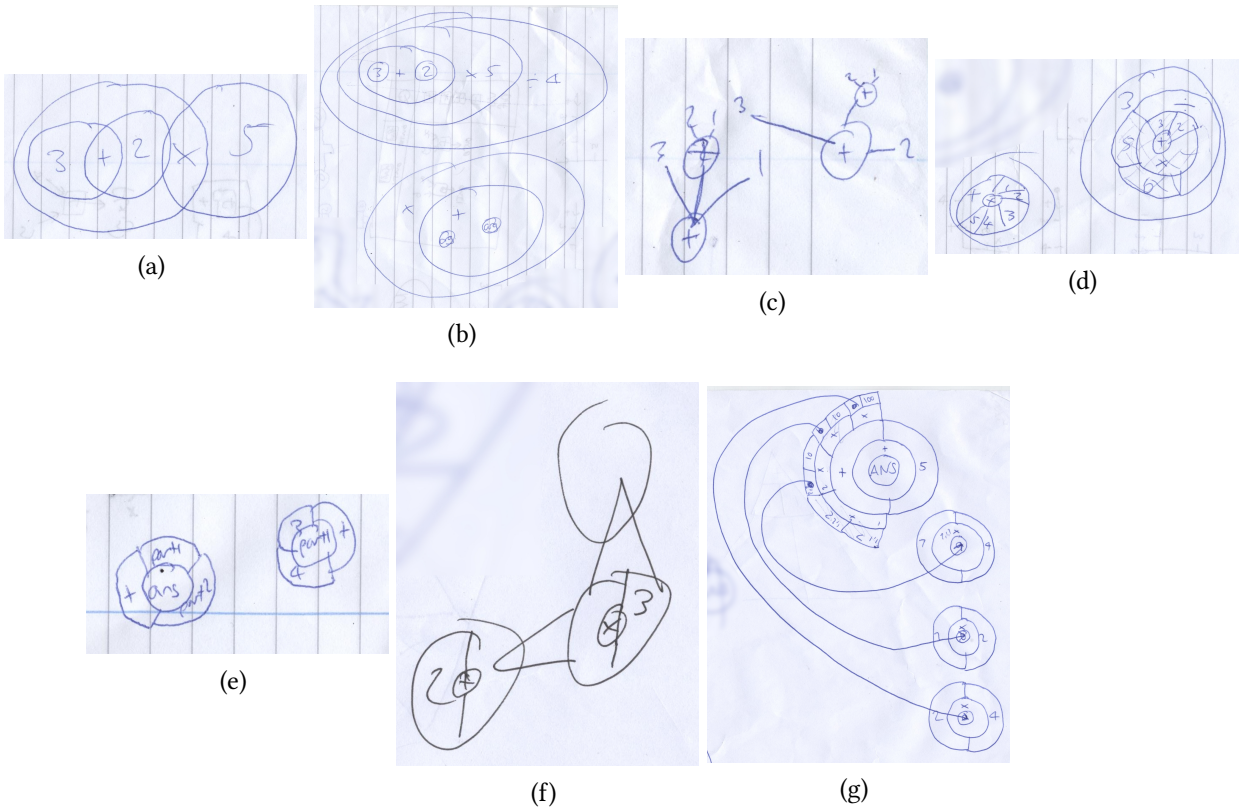


Figure 2.6: Early designs for circle-based calculation metaphors.

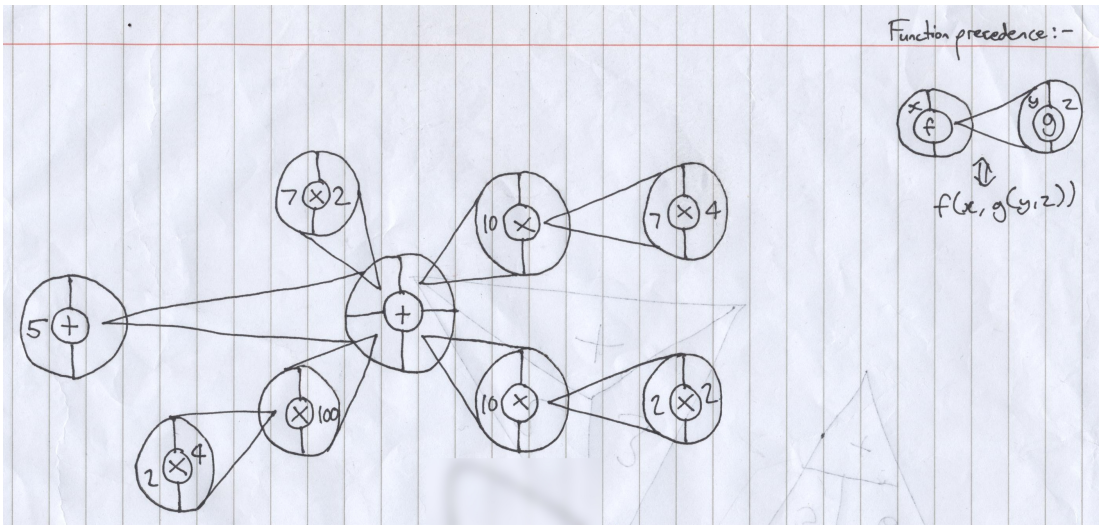
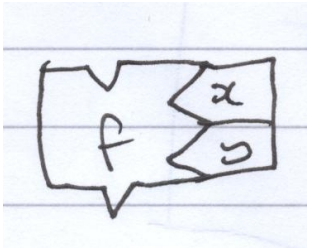
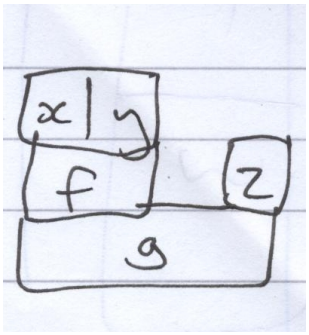


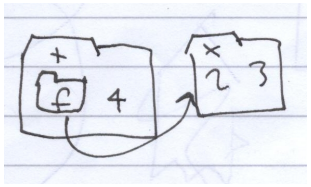
Figure 2.7: A more refined sketch of a circle-based language.



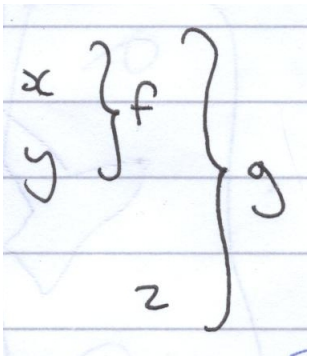
(a)



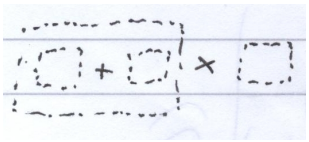
(b)



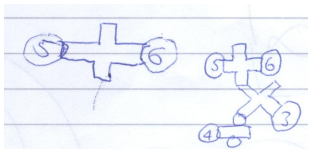
(c)



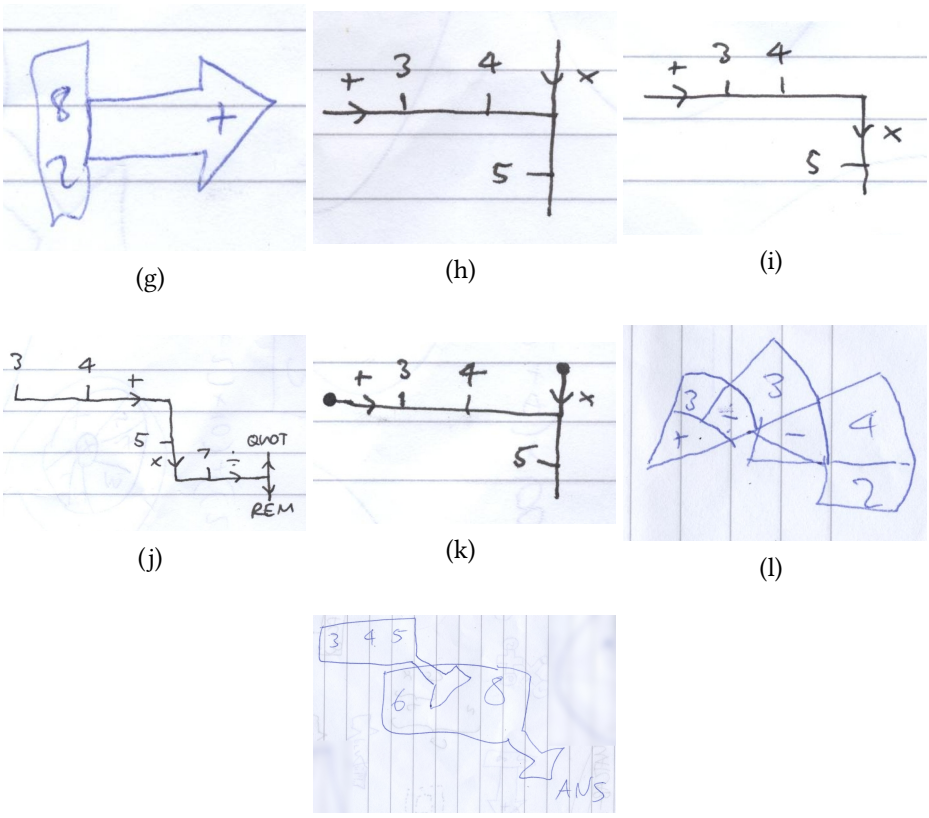
(d)



(e)

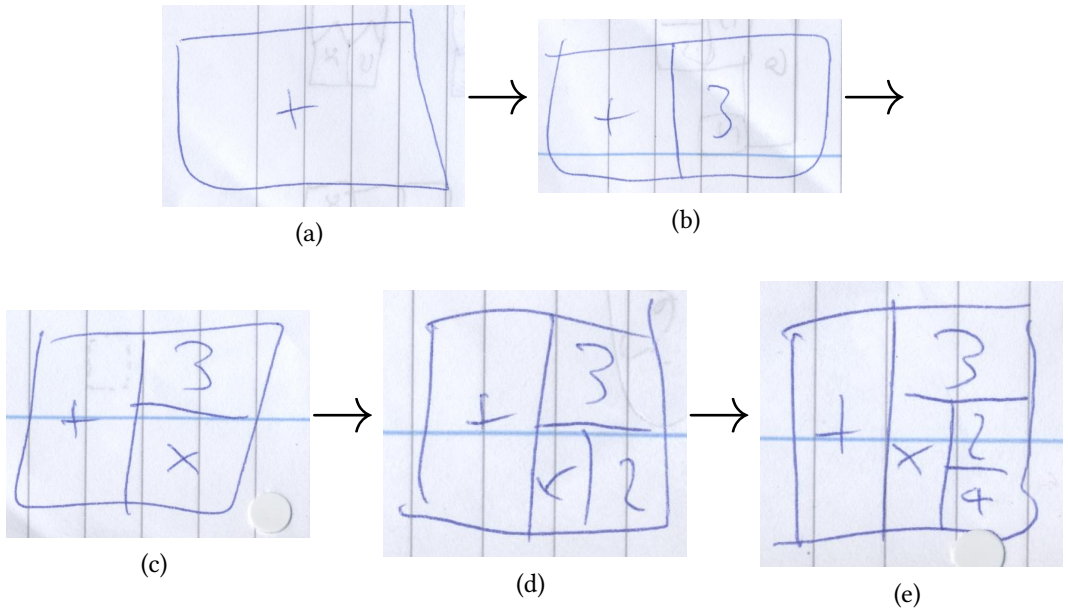


(f)



(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

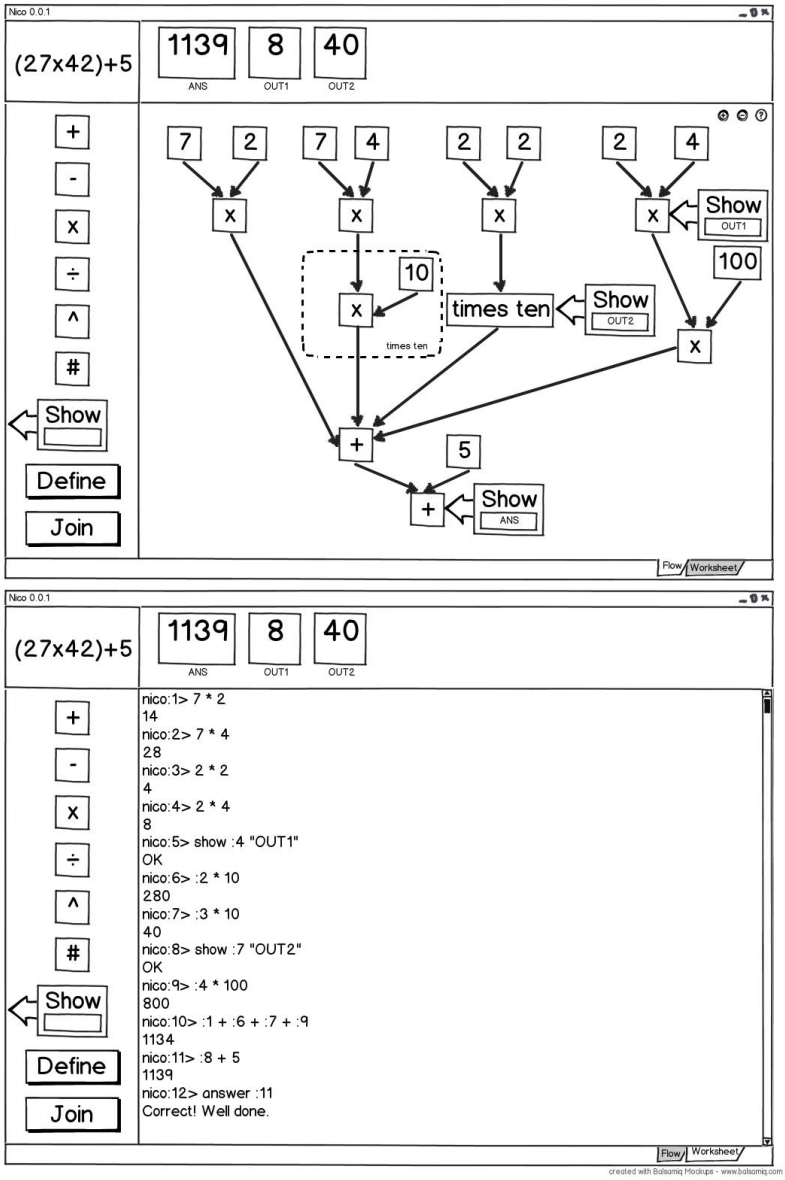


Figure 2.9: The prototype flowgraph-style language.

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

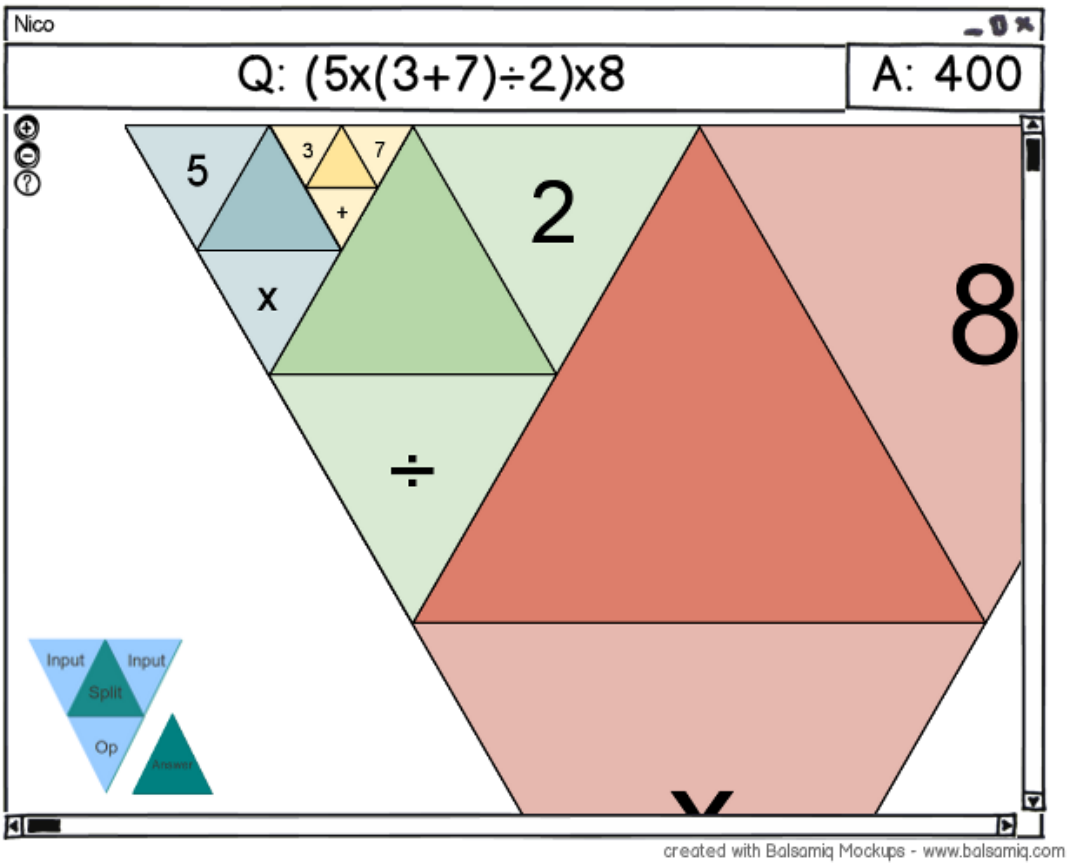


Figure 2.10: The prototype Sierpiński-triangle-based language.

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

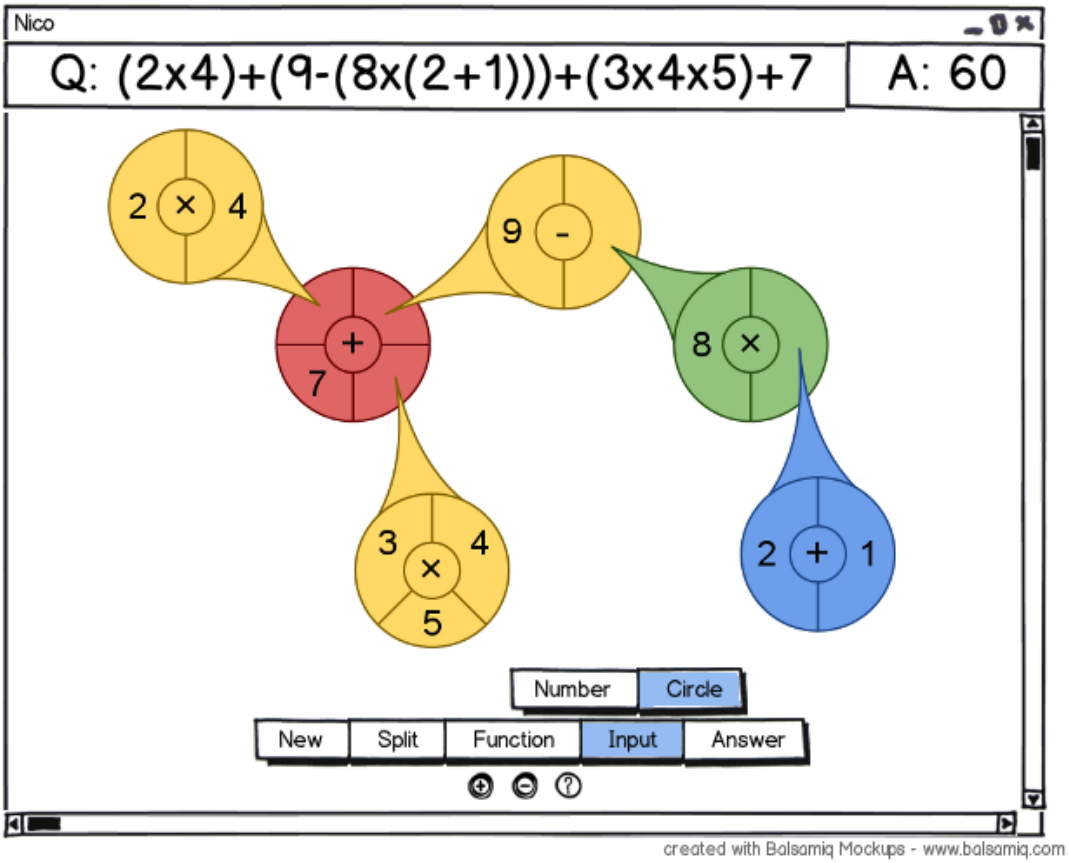


Figure 2.11: The prototype circle-based language.

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 [11], 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

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.

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.

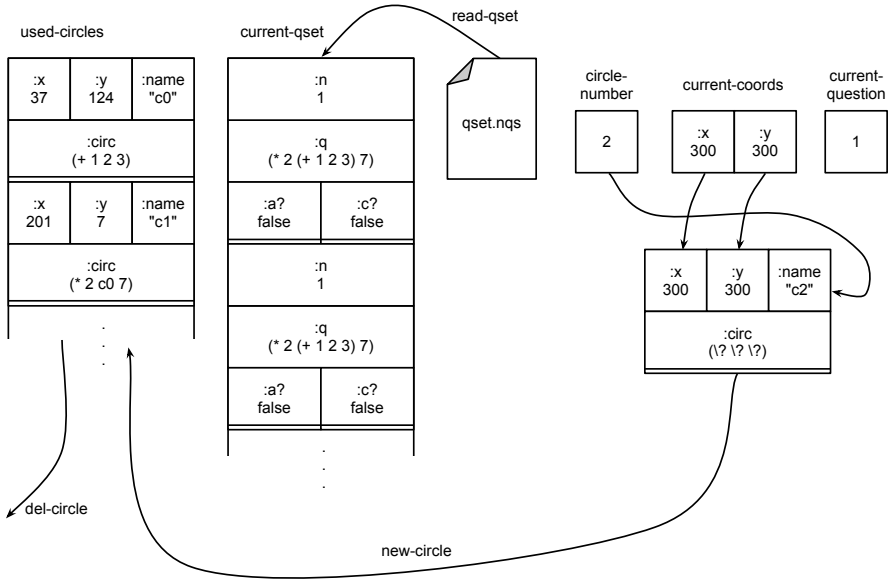


Figure 3.1: A rough model of *Nico's* management of mutable state.

`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:

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

Figure 3.2: The function `new-circle` initialises a circle 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 `(\? \? \?)`.

This map is then sent off to `used-circles`. The value of `circle-number` is incremented by one, and the user-facing canvas area is cleared and re-rendered.

`del-circle` is a more complex function that removes a circle from `used-circles`, taking either a string or two integers as its arguments. If a string is passed to it, the function iterates across the contents of `used-circles`, reconstructing the list by `cons`ing each element in turn on to a new list. If the `:name` field of the head of the `used-circles` list matches the argument passed to `del-circle`, the function recurses without `cons`ing that element on to the new list, thus constructing a new version of the `used-circles` list with the desired circle removed, which can then be sent off to the `used-circles` agent. In the case that two integers (i.e. a pair of co-ordinates) is passed to `del-circle`, the function uses two extra local variables, `c` and `c?`, to determine which circle is being deleted. To determine the value of `c`, the function calls `point-in-circle` on the provided co-ordinates, which is another function that, if the co-ordinates are within the boundaries of a circle, returns the `:name` field of that circle (or `nil` otherwise). `c?` tests whether or not `c` is equal to `nil`. If `c?` has the value `false`, then the co-ordinates passed to `del-circle` are not in any circle, and hence nothing can be deleted. In this case, the function returns `nil`. In the case that `c?` has the value `true`, the function proceeds as if id had been provided with a string rather than two integers, using the value of `c` as the string to compare to. Also included is a related function, `kill-used-circles`, that initialises `used-circles` to the empty list.

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.

```
1  (defn eval-circle
2    "Iterates across a circle list, resolving
3     symbols into their respective circles."
4    [circ]
5    (loop [c (:circ (remove-placeholders circ))
6           out '()]
7      (cond (empty? c) (reverse out)
8            (= (first c) \?) 0
9            (symbol? (first c)) (cond
10                                 (nested?
11                                   (find-circle
12                                     (str
13                                       (first c))))
14                                   (recur (rest c)
15                                          (cons (eval-circle
16                                                  (find-circle
17                                                    (str
18                                                      (first c))))
19                                                out))
20                                 :else (recur (rest c)
21                                              (cons (:circ
22                                                      (find-circle
23                                                        (str
24                                                          (first c))))
25                                                    out)))
26            :else (recur (rest c)
27                         (cons (first c)
28                               out)))))
```

Figure 3.3: The `eval-circle` source.

To manage the set of questions that the user is currently answering, *Nico* uses a function `read-qset` to load a set of questions from a persistent file, exemplified in *Fig. x.y* as `qset.nqs`. `read-qset` works by using the Clojure core function `slurp` to

read in the contents of the file indicated by the path provided as the argument to `read-qset` as one long string. The function `split-lines` provided by the `clojure.string` library then separates each line of the string into an element of a sequence. `read-qset` then proceeds to iterate across the sequence, prepending the string `"{:nn :q '"` and appending the string `" :a? false :c? false}"` to each item in turn, where *n* is a counter starting with value 1 that is incremented with each iteration. This creates a valid specification for an associative map in Clojure, so the Clojure core function `read-string` is called to evaluate each item to a map, which is then consed on to a list. `read-qset` returns a list of question maps, which can then be sent off to the agent `current-qset` to be presented to the user. This method of loading a question set requires that the file provided to `read-qset` must be a plain text file containing one S-expression per line that represents the question that the user is required to answer. For my own testing files, I have opted to use the file extension `.nqs` (*Nico* Question Set) to make it clear that these files are specifically for use as question sets. A future extension to this project could be to develop a partner application to *Nico* that allows tutors to more easily write their own question sets in more familiar language, rather than requiring them to be aware of LISP syntax. Due to time constraints, I decided not to write such an application as it was outside of the stated goals of this project.

My original intention was that the creation of circles would be handled by a macro, `defcircle`, that would allow circles to be 'defined' in much the same way as any other variable in Clojure, to be accompanied by another macro, `letcircle`, that would allow temporary circles to be locally defined within the scope of a given function. As the project progressed, it became apparent that defining circles as variables was not useful, as actions triggered from within the application (e.g. by pressing a button) were unable to access the circles that had been defined in this way. By this point in the project I had already begun to monitor the co-ordinates of used circles using an agent containing a list of maps, so it was a simple solution to use the same agent, which could definitely be accessed at runtime, to contain the circle maps instead. I also had already written a function `new-circle` that was calling `defcircle`, so I decided to abandon `defcircle` and use `new-circle` to initialise a map and send it off to `used-circles`, thus solving the problem.

I encountered some problems with the development of `del-circle` – after implementing the `del-circle` function described above, I noticed that deleting circles that formed part of a chain would cause the application to behave unexpectedly, being unable to render some circles properly and throwing `java.lang.NullPointerExceptions`. This occurred due to the fact that `del-circle` solely removed a circle, with no consideration of if it may be being used as an

argument to the expression represented by another circle. As a result, the application began to throw exceptions as the now-invalid references left behind were being followed by functions such as `eval-circle`, leading them to a non-existant location. To combat this, I developed a replacement function, `del-circle-safe`, that deletes circles in such a way that this problem does not occur.

`del-circle-safe` works by first checking if the circle being deleted is used as an argument to any other circles, using a function `is-arg-of`. `is-arg-of` takes a circle map as an argument and returns a list of maps showing which circles make use of the argument circle, essentially representing the links in the diagram that the user sees. The maps are of the format `{:c` *c* `:a` *l*`}`, where *c* is a string containing the name of the circle that uses the argument circle and *l* is a list of integers indicating the indices of where the argument circle is used in the list of arguments to *c*. To do this, `is-arg-of` iterates across `used-circles`, calling in each instance another function `is-arg?`, which returns `true` if its first argument (in this case the argument of `is-arg-of`) is used in the `:circ` field of its second argument (in this case the current element of `used-circles`). If this function returns `true`, `is-arg-of` conses a new map on to a list: the map consists of the `:name` field of the current element of `used-circles` as the contents of the `:c` field, and constructs the list to use for `:a` using another function, `arg-is`, which uses the `:name` field of its first argument and iterates across the `:circ` field of its second, using a counter to construct a list of indices where the first argument's `:name` appears as a symbol. If `is-arg-of` returns the empty list, then the circle is not used as an argument anywhere and it is, therefore, safe to proceed as before, using the same method of deletion that is used by the original `del-circle`. If not, then the function iterates across the list that `is-arg-of` produces, for each element retrieving the circle indicated by `:c`, constructing a list of indices using `arg-is` and constructing a new circle map in which all instances of the circle being deleted have been replaced with placeholder `\?` values. `del-circle-safe` then calls itself on the circle that uses the original circle to be deleted, thus removing all circles that formed the chain that incorporated the original circle. Finally, the function sends the new circle (with the `\?` values) off to `used-circles` and removes the original circle in the same manner as the unsafe version of `del-circle`; it is now safe to do so, as there are no longer any circles that refer to the circle being deleted (their references having by now been replaced with placeholder `\?` values).

This solved the problem of the leftover references to non-existant circles, but it has also given rise to a new problem: due to the way that I have implemented `del-circle-safe`, and in particular due to the stage in which `del-circle-safe`

must call itself upon the other circles in the chain, deleting a circle that forms part of a chain deletes all of the links between the rest of the circles in the chain, from the circle being deleted down to the root circle. On reflection, a better way to implement safe deletion would have been to include as part of the rendering loop a function to inspect `used-circles`, checking for any invalid references and replacing them with `\?`s.

3.1.2 Interaction Handler

The interaction handler is a collection of functions that form the 'middle layer' of the application: that is, the functions that arbitrate between the backend and the user interface.

The function `render` plays a significant rôle in handling interaction. 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.

At the heart of the interaction handler is a structure called `main-window`. `main-window` is what defines much of the user interface: namely, the main window. It controls what is displayed and the actions to be performed under certain conditions, and listens for certain events such as user input.

Of particular note are the various listeners that monitor the canvas for mouse-based events as part of `main-window`. As the mouse moves around the canvas, `main-window` listens for a `:mouse-moved` event.[1] When this is detected (i.e. when

[1]The Seesaw library refers to several events defined in `java.awt.event.MouseEvent` in this way.

the mouse is moved), `main-window` calls an anonymous function that retrieves the current co-ordinates of the mouse cursor, and calls a function `point-in-circle`, which compares a pair of co-ordinates to those of the circles currently in `used-circles` to determine whether or not the mouse is hovering over a circle. `point-in-circle` returns the name of the circle that the mouse cursor is currently over, or `nil` if it is not currently over a circle. The current co-ordinates of the mouse are sent off to the agent `current-coords` in the form of a map, `{:x` *x* `:y` *y*`}`. The agent `currently-dragging-circle` is also reinitialised to `nil`, as mouse motion without a button being held implies that no circle is currently being dragged. The canvas is then cleared using `clear-screen` and `render`, and in the case that `point-in-circle` does not return `nil` at the current co-ordinates, the circle that the mouse is hovering over is highlighted. If `point-in-circle` returns `nil`, the question text is explicitly unhighlighted.

When `main-window` detects a `:mouse-clicked` event, it retrieves the current co-ordinates of the mouse cursor, and tests for left-click, right-click and Ctrl being pressed. It also uses `point-in-circle` to return the name, if any, of the circle that the mouse cursor is currently over, as well as `arg-in-circle` and `op-in-circle?` to determine the index of the argument, if any, that that mouse cursor is currently over by comparing the current mouse co-ordinates to those of the predetermined placements of arguments in each case of the circle having two to eight arguments, and whether or not the cursor is over the circle's operator, again by comparing the current co-ordinates to the predetermined placement of the operator relative to the circle's co-ordinates. It is also further determined whether or not the cursor is currently over a placeholder argument, using the results of `arg-in-circle`. This information is then used to implement *Nico*'s control scheme, as illustrated in *Fig. x.y*.

For example, a `java.awt.event.MouseEvent.MOUSE_CLICKED` is referred to as `:mouse-clicked`, and a `java.awt.event.MouseEvent.MOUSE_RELEASED` is referred to as `:mouse-released`.

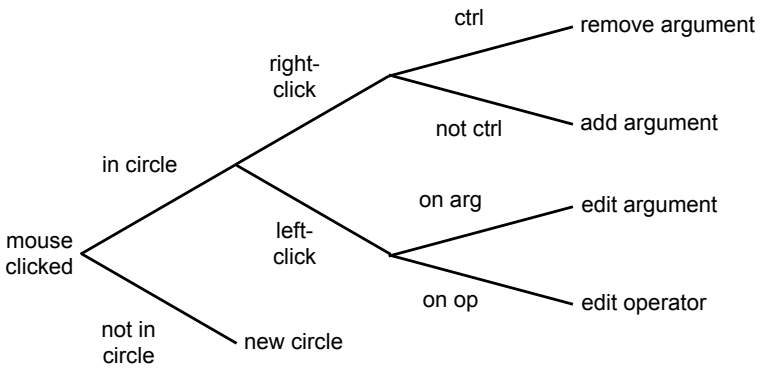


Figure 3.4: Decision tree to determine the appropriate response to a `:mouse-clicked` event.

In the case that the cursor is not over a circle, `new-circle` is called, with the current co-ordinates as arguments, creating a blank circle, as described in the previous section. If the cursor is over a circle and the mouse has been right-clicked, the circle is removed from `used-circles` and a new version is sent off, either with its final argument removed or with a placeholder argument added. In the case that the user tries to remove an argument when there are only two, or to add an argument when there are already eight, an alert box appears, warning the user that they are only allowed a minimum of two and a maximum of eight arguments.

If the mouse is over a circle, has been left-clicked and is over one of the circle's arguments, `main-window` draws a small blue circle around the argument and launches a dialogue box using the function `mod-arg-dialogue`, which defines a simple window containing a spinner and returns the contents of that spinner upon the user pressing the `OK` button. The circle that had been clicked originally is then removed from `used-circles`, and replaced with a version in which the chosen argument has the value that the user has input. `clear-screen` and `render` are called, and the answer displayed at the top of the application is updated by using `find-root` to find the root circle of the diagram, and then re-evaluating the entire expression. The process is similar in the case that the mouse was over a circle's operator, except that the function `mod-op-dialogue`, which defines a small dialogue box containing four radio buttons corresponding to the four arithmetic operations and returns the user's selection, is called, rather than `mod-arg-dialogue`.

Nico's drag-and-drop functionality is also implemented using `main-window`'s listeners. When a `:mouse-pressed` event is detected, the mouse co-ordinates are

once again retrieved and `point-in-circle` is called.[2] If `point-in-circle` does not return `nil`, the circle map that corresponds to the name returned is sent off as part of a map to the agent `currently-dragging-circle`. This map takes the form `{:c` *c* `:t` *t* `:m` *m*`}`, where *c* is the circle map, *t* is the current system time in milliseconds and *m* is the integer representing the mouse button that was pressed. When the application detects a `:mouse-dragged` event, this information is then used to determine which action to take. First, `currently-dragging-circle` is checked. If it is `nil`, no action needs to be taken. To check that the user has deliberately dragged the mouse, the current system time is compared to the `:t` value in `currently-dragging-circle`'s map. If the difference is greater than 100ms, this is determined to be an intentional mouse drag, and the function proceeds. If the left mouse button is pressed, the mouse co-ordinates are compared to the location of the bin icon. If the circle has been dragged into the bin area, it is removed. Otherwise, it is relocated to the current position of the mouse cursor by removing it from `used-circles` and sending off a version with the `:x` and `:y` fields modified appropriately. If the right mouse button is held, a line is drawn from the circle to the current location of the mouse, and a circle is also drawn around this point. If the mouse is dragged on to an argument that is not in the source circle, the line and circle are anchored in that argument's slot and the target circle is removed from `used-circles`, being replaced by a version with the appropriate argument modified to be a symbol referring to the source circle.

As mentioned above, *Nico* has the ability to highlight sections of the question currently being answered that correspond to circles or sets of circles within the user's diagram. This is achieved using a function `highlight`, which uses a function `lisp-to-maths` to convert S-expressions into standard mathematics. `lisp-to-maths` uses a counter beginning with value 0 to append the operator of an S-expression on to a string of its arguments every time the counter is odd, producing the infix expressions many people are more comfortable with. When the counter is not odd, if the next argument is a list, `lisp-to-maths` recursively calls itself on that list, appending to the string being produced the string `"("`, the output of the recursive call to `lisp-to-maths` and the string `")"`, creating a more familiar-looking subexpression using brackets. If the counter is not odd and the current element of the S-expression is not a list, the element is simply appended to the string being created. This produces a line of mathematics from an S-expression;

[2] `:mouse-pressed` is not the same as `:mouse-clicked`. The latter only occurs when the mouse is pressed **and then** released, whereas the former occurs every time the mouse button is pressed. It is for this reason that the actions that occur when `:mouse-clicked` is detected must reinitialise `currently-dragging-circle` to `nil`.

for example, an input of `(+ 1 2 (* 3 4) 5 (- 6 7) 8)` will produce an output of "1+2+(3×4)+5+(6-7)". This function is also used to produce the contents of the Question panel from the questions contained in the question set file. `highlight` uses `lisp-to-maths` to compare the question string to the translation of a circle into regular mathematics using another function `detect-subs`, which takes two strings as arguments and returns a list of maps containing the start and end indices of every instance where the first argument appears as a substring of the second argument. `detect-subs` iterates across its second argument, character by character, using a counter that is incremented by one with each iteration to compare its first argument to the substring of its second that begins at the current value of the counter and ends at the same value, plus the length of the substring being searched for. If it detects that those two strings are equal, the start and end indices are put into a map and appended to a list. `highlight` iterates across the resultant list of `detect-subs`, and changes the colour of the text that matches the result of calling `lisp-to-maths` on the result of evaluating a circle.

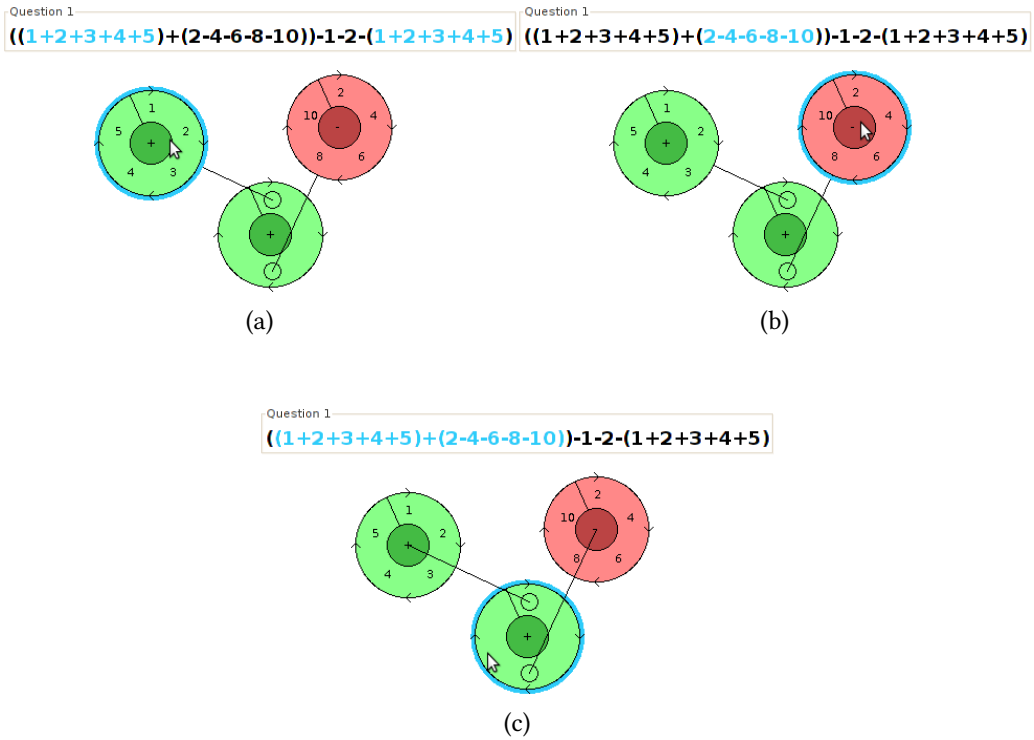


Figure 3.5: Three instances of the question text being highlighted as a corresponding circle is moused-over.

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.

3.2 User Interface

The user interface comprises the parts of the application which are user-facing; that is, they are that which the user directly interacts with. The development of *Nico*'s user interface was key to the success of the overall project, being primarily an experiment in human-computer interaction.

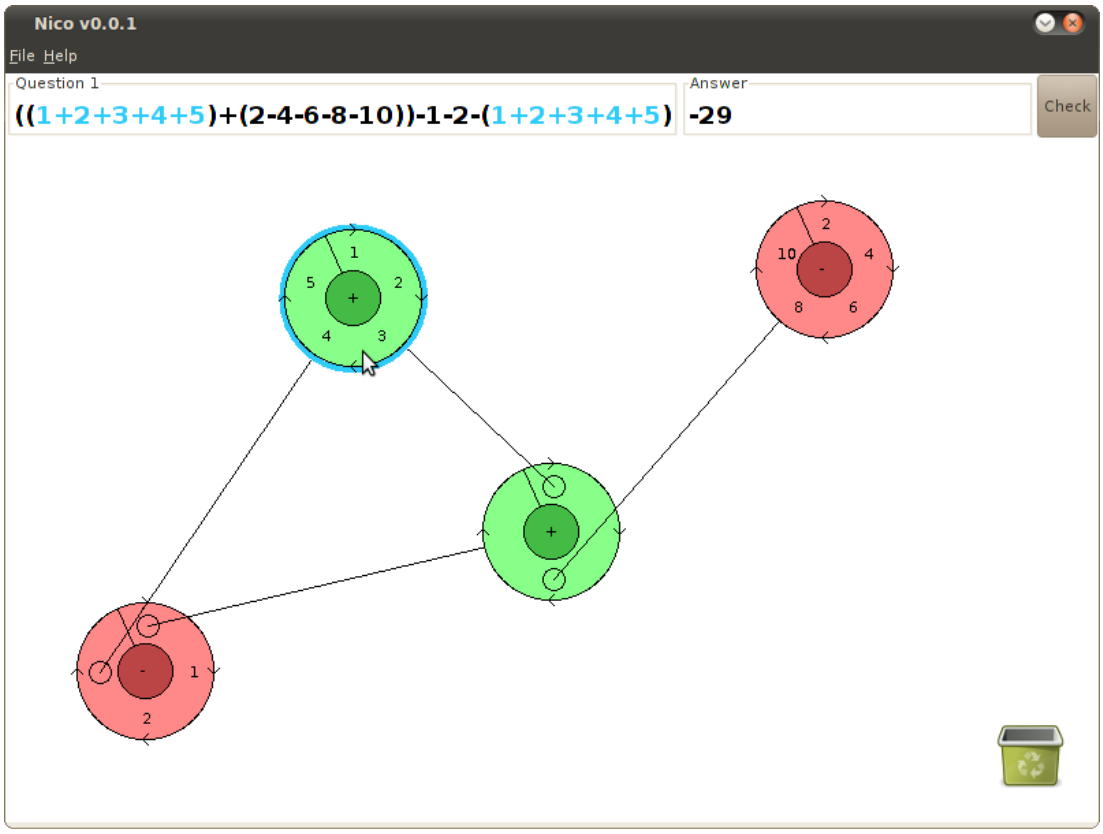
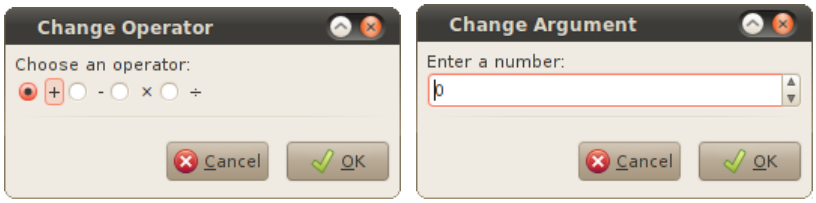


Figure 3.6: A screenshot of a *Nico* session.

The application itself was kept fairly minimal: the aim was to emphasise the notation as the most important aspect of the software. As such, the main application window, as shown above in *Fig. x.y*, is mostly dedicated to the canvas area in which the user answers the question. The main application window is split up into two main sections, one at the top, containing some information panels that display the question being answered and the current total of the user's calculation, as well as the Check button to submit the current total as the answer. The lower panel contains the large canvas area in which the user is able to construct diagrams with which to answer the question.

Nico uses a predominantly mouse-based control system for the construction of calculations in the application's notation. Left-clicking on a blank patch of canvas creates a new circle, initialised with two arguments, ? and ?, and with operator ?. Right-clicking on a circle increments its number of arguments by one, up to a maximum of eight. Holding the Ctrl key and right-clicking on a circle decrements its number of arguments by one, down to a minimum of two. Circles can be moved around the canvas by left-clicking and dragging the circle to the desired location. If

a circle is dragged to the bin icon in the bottom left-hand corner of the screen, then it is removed. Left-clicking on a circle's operator brings up a dialogue box with a selection of operators in it, with a radio button for each (see *Fig. x.y*). Similarly, left-clicking on a circle's argument brings up a dialogue box containing a spinner, which can be used to set a new value for that argument. To use one circle as an argument to another, right-clicking and dragging allows the user to drag a line ending in a circle to the desired slot on the recipient circle.

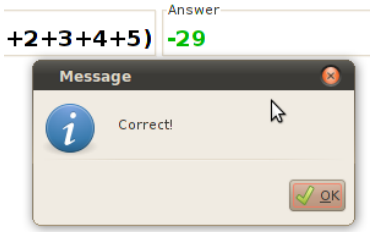


(a) Editing a circle's operator (b) Editing a circle's argument

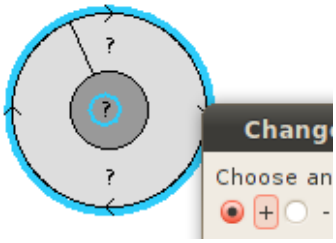
Figure 3.7: The dialogue boxes used to modify existing circles in the application.

Nico also uses colour-coding in various areas of the application. Circles are coloured according to their operators (addition is green, subtraction is red, multiplication is blue, division is yellow and ? is grey), clearly separating different kinds of operations and making the notation more readable. Colour-coding is also used in the information panels at the top of the application: positioning the mouse cursor over a circle highlights the part of the question that it is likely to represent in blue, and also highlights the circle with a blue outline. When an answer is submitted, the text in the answer panel turns red if the answer was incorrect, or green if the answer was correct, providing an unintrusive form of feedback, allowing the user to continue uninterrupted (i.e. no input is required) if their answer is wrong.

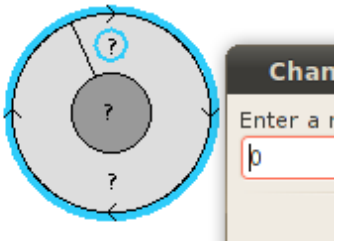
(a) Red text indicates that the answer submitted was incorrect



(b) Green text indicates a correct answer; a dialogue box allows the user to progress to the next question when they are ready



(c) The circle's operator is highlighted when it is being edited



(d) An argument is highlighted when it is being edited

Figure 3.8: Examples of colour-coding in *Nico*.

During its development, *Nico*'s user interface went through a number of revisions. Initially, the application was intended to use more buttons (see the prototype in *Fig. x.y*, above), having more controls immediately visible to the user. In the first usable version of the interface, there were a number of buttons that appeared down the side of the application. Each button (New, Edit, Delete, etc.) would cause a dialogue box to appear to the user, which required a circle to be specified by name (at this point in the development of the project, it was required that the user name each of their circles) before any operations could be performed upon it. Circles were all the same colour, and the text-highlighting and colour-coding operations had not yet been implmented. It was also the case that the links between circles did not end in argument slots, but rather fed into the centre of the target circle. This design was abandoned as the separation between the user and the diagram itself was too great; creating circles by specifying options in many similar-but-slightly-different dialogue boxes and watching the results appear on the canvas did not feel like a natural way to interact with one's calculation, and hence would have been inappropriate for the target audience. The intention of the project is not to require the user to spent a significant amount of time learning how to use the system before it becomes a useful tool to them.

Later revisions of the software removed the buttons and implemented a system of context menus with which to construct answers. Right-clicking on a blank patch of the canvas gave the user the option to create a new circle, and right- clicking on an existing circle presented the user with the option to either edit or delete that circle. Choosing to either create of edit a circle displayed a dialogue box (*Fig. x.y*), with many options for the user to choose from: the name and operator of a circle could be set, with up to eight arguments could be enabled, with a choice of either a number (set by a slider) or another circle (chosen using a list of radio buttons that was generated upon opening the dialogue box) as the value of that argument. Answer-checking was also made automatic: as soon as the total reaches a value equal to that of the answer to the question, the user is immediately told so, and moved on to the next question. The circles at this point are still one colour only, and there is no expression-highlighting. The colour-coded totals had been implemented by this point. This design was ultimately abandoned as, again, it separated the user from their own work too much. Although this design was an improvement, allowing the user to interact directly with the canvas to perform operations upon circles that were able to specified by pointing with the mouse, rather than by typing out a name, the grand unified dialogue box still required the user to perform operations without it being entirely obvious what the consequences would be, thus requiring an unnecessary amount of *premature commitment* from the user. The automatic answer-checking was also a problem, as it did not give the user a chance to review their answer before proceeding. Indeed, it did not give the user a chance to learn from their mistakes, say for example if they had accidentally reached the right answer as part of a larger, incorrect calculation that they had previously intended to make.

The final, current revision of the software reintroduces the `Check` button to address the issues with the automatic answer-checking in the previous version of the application. It also introduces the control scheme detailed above, allowing the user to feel more like they are directly manipulating their work in progress, with each action having clear consequences. A colour scheme for the circles was implemented, to make the notation easier to read, and the expression-highlighting functionality was also introduced, along with the other means of highlighting, as shown above.

OLD VERSIONS OF THE APP LOL

Figure 3.9: *Nico*'s user interface went through a number of revisions before reaching its current state.

3.2.1 Notation

OLD VERSIONS OF THE NOTATION LOL

Figure 3.10: The notation used also went through a number of revisions before reaching its current state.

The notation itself was also revised several times during the development of the application. The original vision of the notation can be seen in *Fig. x.y*. The first implementation of the notation consisted of uniformly-coloured circles connected by lines feeding into the centre of the recipient circle. There were no placeholder values; these were unnecessary as the system of dialogue boxes meant that at no point could a circle have any aspect of itself in an undefined state. There was also no indication of argument ordering, which is acceptable when the operator is commutative (i.e. addition or multiplication), but for operations where this is not the case, this can quickly become confusing. Although the ordering and placement of arguments was defined, with no indication of this, the user is required to figure this out through experience, which is counter to the intentions of the software. As mentioned above, it should not be required that the user spend a significant amount of time learning how to use the system, especially due to deficiencies in the notation. Another, greater, problem related to argument ordering was that of the circles that linked to other circles. With all input circles pointing to the operator at the centre of their target circle, it was not at all clear in which order the input circles evaluated, neither relative to each other nor to the other arguments in the circle.

The notation changed somewhat to address the problems listed above. In its current revision, the notation has had a number of features added. Firstly, the circles are now colour-coded by operator, making it easier for the user to distinguish between different kinds of circles. The circles themselves have also been embellished slightly with the addition of a line denoting the star of the expression, just to the side of the first argument at the top of the circle. Smalls arrow have also been added to the edge of the circles to indicate in which direction the arguments should be read. Finally, the links between circles have been changed such that they now occupy an argument slot that could otherwise be taken by a number, with a circle on the end of the link line to clearly show where the link terminates.

3.3 Summary

In this chapter, the implementation of the project was discussed in some detail, as well as the challenges that were faced during this development phase. The software

itself fulfils and exceeds the criteria for success outlined in the project proposal, being a complete and usable system that improves greatly, in terms of cognitive dimensions, upon the traditional method of handwritten arithmetic. Third-party libraries, in particular Seesaw, were used where appropriate. The application divides into three main sections: the backend, handling mathematical operations and memory management, the interaction handler, helping the user interface to interface with the backend, and the user interface itself. The user interface was revised many times to correct deficiencies in the design that would have led to a poor user experience, and the backend and interaction handler were also both restructured as new demands upon the software became apparent.

In the next chapter, the software will be evaluated, discussing test results and the user study that was conducted as an extension to the project.

Chapter 4

Evaluation

This chapter concerns the testing and evaluation of *Nico*; carried out to determine the quality and utility of the software.

4.1 Testing

Throughout the development of the software, I performed unit tests to ensure the integrity of my code. My development environment of choice, GNU Emacs using SLIME/SWANK and clojure-mode, includes functionality such that commented-out lines of code can be evaluated at will. As such, it was possible for me to write unit tests into the main body of my code, evaluating them whenever a change may have affected how a function behaved. In the interests of tidiness, many tests have been removed as they became unnecessary later on in the devlopment in the project, but *Fig. x.y* shows one of the test blocks that were left in the most recent revision of the source code.

```
(defn detect-subs
  "Returns a list of maps containing start and end indices showing where a substring c appears in the superstring q."
  [c q]
  (loop [s (cond (= (subs q
                          (- (count q) (count c))
                          (count q)) c) (split q
                                               (re-pattern
                                                (escape
                                                 c
                                                 {\+ "\\+"
                                                  \( "\\("
                                                  \) "\\)"})))
                 :else (butlast (split q
                                       (re-pattern
                                        (escape
                                         c
                                         {\+ "\\+"
                                          \( "\\("
                                          \) "\\)"}))))))
         i 0
         l '()]
    (cond (empty? s) (reverse l)
          (= c (subs q
                     (+ i (count (first s)))
                     (+ i (count (first s)) (count c)))) (recur (rest s)
                                                                (inc i)
                                                                (cons {:s (+ i (count (first s)))
                                                                       :e (+ i (count (first s)) (count c))}
                                                                      l))
          :else (recur s (inc i) l))))

;; (def tq (lisp-to-maths (eval (:q (first @current-qset)))))
;; (subs tq (:s (nth (detect-subs "1+2" tq) 0)) (:e (nth (detect-subs "1+2" tq) 0)))
;; (subs tq (:s (nth (detect-subs "1+2" tq) 1)) (:e (nth (detect-subs "1+2" tq) 1)))
;; (subs tq (:s (nth (detect-subs "1+2" tq) 2)) (:e (nth (detect-subs "1+2" tq) 2)))
;; (let [s "roflolmaomglolwtf" i (detect-subs "lol" s)] (subs s (:s (nth i 0)) (:e (nth i 0))))
;; (let [s "roflolmaomglolwtf" i (detect-subs "lol" s)] (subs s (:s (nth i 1)) (:e (nth i 1))))
;; (let [s "roflolmaomglolwtf" i (detect-subs "lol" s)] (subs s (:s (nth i 2)) (:e (nth i 2))))

;; (let [s "0123456789" i (detect-subs "456" s)] (subs s (:s (first i)) (:e (first i))))
;; (let [s "roflolmaomglolwtf" i (detect-subs "lol" s)] (subs s (:s (first i)) (:e (first i))))
;; (let [s "fagaha" i (detect-subs "a" s)] (subs s (:s (first i)) (:e (first i))))
;; (let [s "lalala" i (detect-subs "a" s)] (subs s (:s (first i)) (:e (first i))))
;; (detect-subs (lisp-to-maths (eval-circle (find-circle "c0"))) (lisp-to-maths (eval (:q (first @current-qset)))))
;; (let [s "dcbabcd" i (detect-subs "a" s)] (subs s (:s (first i)) (:e (first i))))
;; (let [s "dcbabcd" i (detect-subs "a" s)] (subs s (:s (nth i 1)) (:e (nth i 1))))

```

Figure 4.1: An example of a block of inline tests, taken from `core.clj`, showing some tests to be run to test the implementation of `detect-subs`.

4.2 UI Evaluation

4.2.1 Notation Evaluation

4.3 User Study

To assess the utility of the software relative to handwritten arithmetic, a user study was conducted, allowing real users to get to grips with the system, and provide feedback on it. To this end, users were given a tutorial video and a questionnaire (see *Appendix A* for the full questionnaire) to complete whilst using the software.

Test subjects were first asked to sign a statement of informed consent, giving their consent to participate in the study, and were then asked if they had previously used *Nico*, and to indicate their confidence in their ability to solve simple mathematical problems. Users were then divided into two groups. The first watched the tutorial video and completed one question set using *Nico*, and then answered the same questions by hand. The second group completed the questions by hand before watching the video and using the software. The time taken to answer each question was recorded, with *Nico* including functionality to query the system time as the user progressed and the handwritten questions being timed using a stopwatch. Both groups were then asked to complete a series of questions to provide feedback on the software, taking questions from Blackwell and Green's sample questionnaire. [4]

4.4 Goals

4.5 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] T. R. G. Green and M. Petre. Usability analysis of visual programming environments: a 'cognitive dimensions' framework, 1996.

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

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

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

[10] 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.

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

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

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

Appendix A

Questionnaire

The questionnaire used in the user study follows.

Nico: An Environment for Mathematical Expression in Schools

Thank you for agreeing to participate in the user study for my Part II Project. *Nico* is a piece of educational software designed to aid learners in the visualisation of mathematical problems, by separating out the constituent parts of a calculation into distinct visual units on-screen.

The purpose of this study is to ascertain how well *Nico* achieves its goal of providing a clear, accessible, interactive means of calculation, and to gather feedback on how the application could be improved. The study also aims to compare *Nico* to traditional mathematical methods.

Please review and sign the attached Statement of Informed Consent (Section 1) and please feel free to ask any questions you may have about it.

1 Statement of Informed Consent

Statement of Informed Consent

I state that I am over 18 years of age and wish to participate in a program of research being conducted by Philip Yeeles at the University of Cambridge. I acknowledge that this study has been approved by the University of Cambridge Computer Laboratory Ethics Committee.

The purpose of this research is to assess the usability of a graphical notation and software application for representing mathematical calculations in a graphical manner.

The study involves the use of the application whilst being supervised. I will be asked to complete certain tasks both with and without the application, and I will also be asked open-ended questions about the application and my experience as a user thereof.

All information collected in the study is confidential, and my name will not be identified at any time. I understand that I may ask questions or terminate my involvement in the study freely and at any time without consequence.

I acknowledge that my (anonymised) responses may be published in the final report, and that this report will be made publicly available from the University of Cambridge Computer Laboratory Library and from GitHub.

Signed:

Name:

Date:

THIS PAGE INTENTIONALLY LEFT BLANK

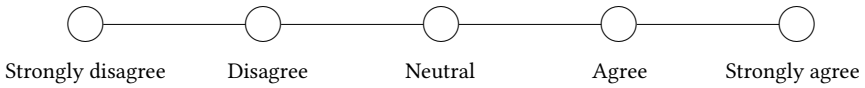
2 Study

Before we begin, please answer the following questions.

- Have you used *Nico* before (circle as appropriate)?

  Yes No

- How well do you agree with the following statement (cross as appropriate)?
  I am confident in my ability to calculate answers to simple mathematical problems.



Thank you. You will now be issued a group number.

- If you are in Group 1, please proceed to Section 2.1.
- If you are in Group 2, please proceed to Section 2.2.

If you would like to stop at any point, please don't hesitate to let me know.

2.1 *Nico*

In this section, you will use *Nico* to solve some simple mathematical problems. The purpose of this section is to evaluate how well *Nico* performs in comparison with manual calculation. We will be keeping a record of the time taken to complete each problem, but please do not let this make you feel rushed. Work at a pace that is normal and comfortable for you.

Before we begin the tasks, please watch the instructional video `tut.ogm` for a briefing on how to use *Nico* and an explanation of its controls.

Now that you have done this, please spend 5 minutes experimenting with *Nico*. Open the application and load the file `qs/blank.nqs` using the file chooser. As you explore, please tell me your thoughts about the application.

Now, let us move on to the problems. Please close the application and open it again, this time loading the file `qs/user-study.nqs`. You will be presented with a series of problems to solve using *Nico*; please solve them.

Thank you very much.

- If you are in Group 1, please continue to Section 2.2
- If you are in Group 2, please continue to Section 3

2.2 Manual Calculation

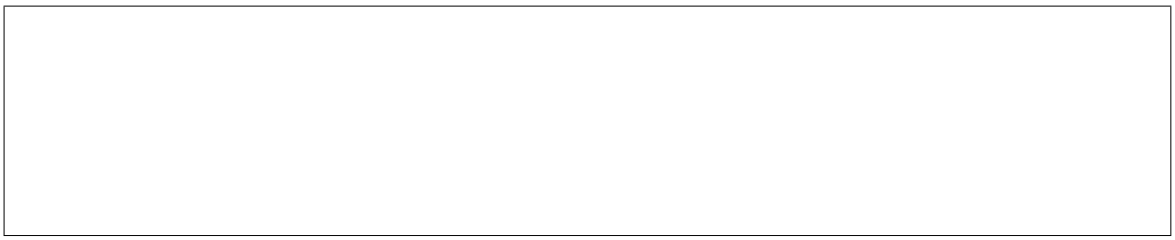
In this section, you will solve some simple mathematical problems using pen and paper. The purpose of this section is as a control, to compare to your results using *Nico*. Once again, we will be keeping a record of the time you take to complete each question, but please do not let this make you feel rushed. Work at a pace that is normal and comfortable for you, and don't forget to show your working.

Let us begin.

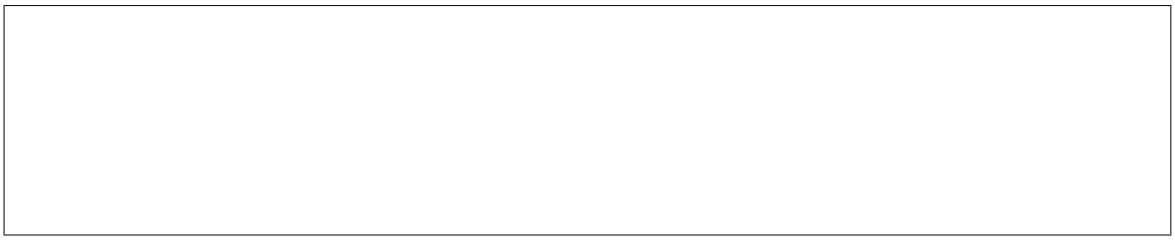
1. *2+3*

2. *9÷3*

3. *1+2+3+4+5*



4. *(2×4)+(3-5)*



5. *((3×4)÷(3+3))×8*

6. *((1+2+3+4+5)+(2×4×6×8×10))×1×2×(1+2+3+4+5)*

7. *12+14*

8. *247×35*

9. *120÷((2×10)+5+5)*

10. *((2+5)×(6÷2)×(9-8))+((3+4)-(5×6))+120*

Thank you very much.

- If you are in Group 1, please continue to Section 3
- If you are in Group 2, please continue to Section 2.1

3 Questions

3.1 Notation

When using the system, what proportion of your time (as a rough percentage) do you spend:

1. Searching for information within the notation ☐ %
2. Translating substantial amounts of information from some other source into the system ☐ %
3. Adding small bits of information to a description that you have previously created ☐ %
4. Reorganising and restructuring descriptions that you have previously created ☐ %
5. Playing around with new ideas in the notation, without being sure what will result ☐ %

3.2 Cognitive Dimensions

3.2.1 Visibility and Juxtaposability

1. How easy is it to see or find the various parts of the notation while it is being created or changed? Why?

2. What kind of things are more difficult to see or find?

3. If you need to compare or combine different parts, can you see them at the same time? If not, why not?

3.2.2 Viscosity

1. When you need to make changes to previous work, how easy is it to make the change? Why?

2. Are there particular changes that are more difficult or especially difficult to make? Which ones?

3.2.3 Error Proneness

1. Do some kinds of mistake seem particularly common or easy to make? Which ones?

2. Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?

3.2.4 Closeness of Mapping

1. How closely related is the notation to the result that you are describing? Why?

2. Which parts seem to be a particularly strange way of doing or describing something?

3.2.5 Role Expressiveness

1. When reading the notation, is it easy to tell what each part is for in the overall scheme? Why?

2. Are there some parts that are particularly difficult to interpret? Which ones?

3. Are there parts that you really don't know what they mean, but you put them in just because it's always been that way? What are they?

3.2.6 Hidden Dependencies

1. If the structure of the calculation means that some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kind of dependencies are hidden?

2. In what ways can it get worse when you are creating a particularly large description?

3.2.7 Progressive Evaluation

1. How easy is it to stop in the middle of creating some notation, and check your work so far? Can you do this any time you like? If not, why not?

2. Can you find out how much progress you have made, or check what stage in your work you are up to? If not, why not?

3. Can you try out partially-completed versions of the calculation? If not, why not?

3.2.8 Provisionality

1. Is it possible to sketch things out when you are playing around with ideas, or when you aren't sure which way to proceed? What features of the notation help you to do this?

3.2.9 Secondary Notation

1. Is it possible to make notes to yourself, or express information that is not really recognised as part of the notation?

2. If it was printed on a piece of paper that you could annotate or scribble on, what would you write or draw?

3. Do you ever add extra marks (or colours or format choices) to clarify, emphasise or repeat what is there already?

3.3 Feedback

1. Do you find yourself using this notation in ways that are unusual, or ways that the designer might not have intended? If so, what are some examples?

2. After completing this questionnaire, can you think of obvious ways that the design of the system could be improved? What are they? Could it be improved specifically for your own requirements?

Thank you very much for your help with my project! Your responses will remain confidential, although they are liable to appear in an anonymised form in the final report, of which a copy will be retained by the University of Cambridge Computer Laboratory Library (`http://www.cl.cam.ac.uk/library/`). The final report will also be available via my repository at GitHub (`http://github.com/loomcore/nico`).

Appendix B

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

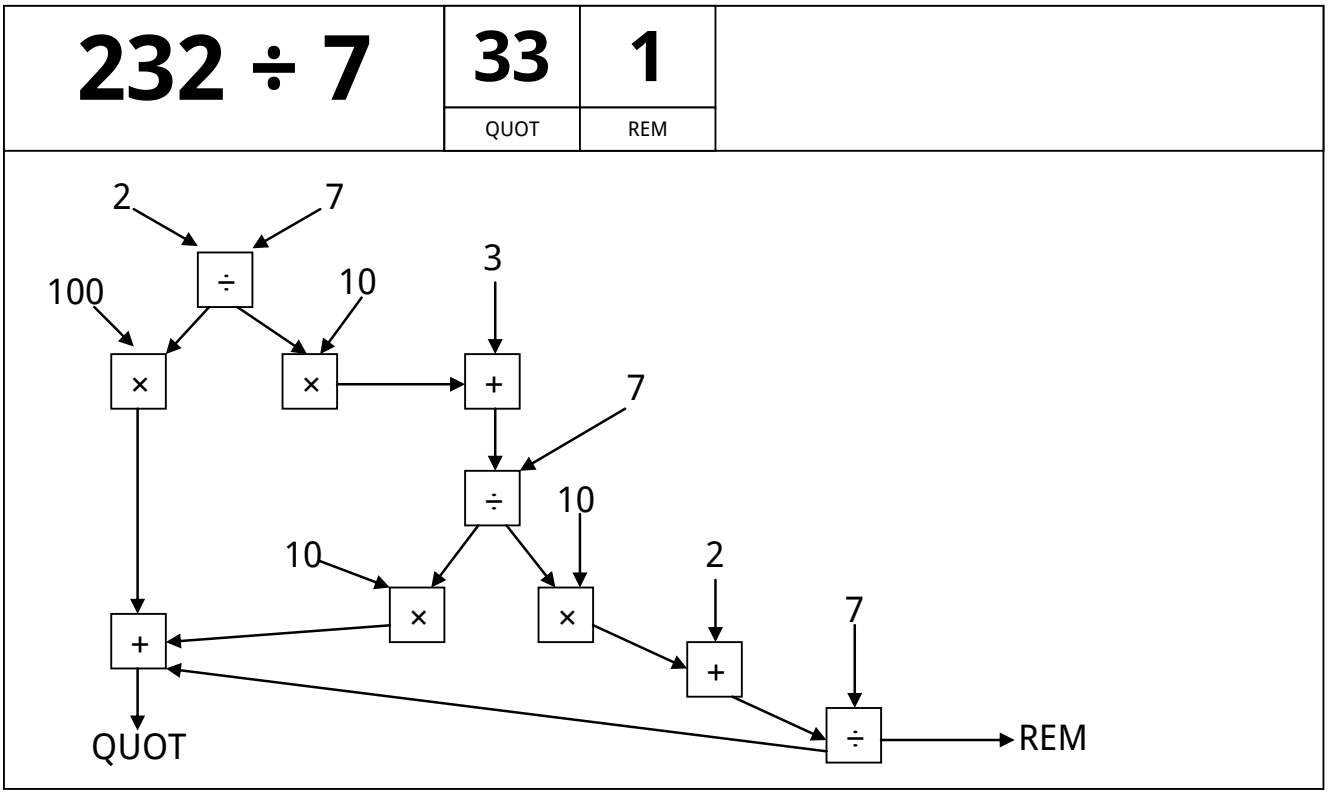
The following main learning tasks will have to be undertaken before the project can be started:

- 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.