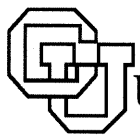
Building Curricula to Shape Cognitive Models: A Case Study of Higher Order Procedures

Julie DiBiase

CU-CS-808-96



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

BUILDING CURRICULA TO SHAPE COGNITIVE MODELS: A CASE STUDY OF HIGHER ORDER PROCEDURES

by

JULIE DiBIASE

B.A., Smith College, 1990

M.S., University of Colorado, Boulder, 1993

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirement for the degree of

Doctor of Philosophy

Department of Computer Science

1995

This thesis for the Doctor of Philosophy degree by

Julie DiBiase

has been approved for the

Department of Computer Science

by

_____

Michael Eisenberg

_____

Jim Martin

Date _____

Abstract

DiBiase, Julie (Ph.D., Computer Science)

Building Curricula to Shape Cognitive Models: A Case Study of Higher Order Procedures

Thesis directed by Assistant Professor Michael Eisenberg

In computer science, functional programming is traditionally taught in an axiomatic style that discourages the use of visual intuition. This paper, in contrast, describes a formula for designing curricula based on multi-modal imagery building. The work presents a series of case studies in which imaging theory is applied to the purely abstract concept of functional data objects.

These investigations have provided insight into the historically troubling pedagogical puzzles presented by the abstract nature of mathematics and related disciplines. It is the claim of this work that facilitating the teaching and learning of many "higher-order" mathematical concepts lies in demystifying this notion of abstraction. By using imaging tools to unravel complex concepts, students of the curriculum are encouraged to mature into experienced and expert abstracters.

Acknowledgments

I would like to thank the following groups of people for their contributions: the children who enthusiastically and (usually) patiently became SchemePaint aficionados; the computer and cognitive science graduates and undergraduates who worked out many a Scheme problem for the purposes of data collection; the professors and graduate students in computer science, psychology, math and physics who shared their insights on the matter of this work, including Eric Blough, Mike Doherty, David Grant, Judy Gurka, Missy Schreiner, Eric Stade, and Pearl Tesler.

Thanks to my committee -- Mike Eisenberg, Dirk Grunwald, Clayton Lewis, Jim Martin, and Nancy Songer -- for helping to bring this interdisciplinary work together.

Especially, I would like to thank my advisor, Mike Eisenberg, for his adept and patient editting of what seemed like innumerable drafts.

Support for this work was provided by National Science Foundation grant (IRI-9258684).

Table of Contents

List of Tables

List of Figures

1 Introduction

1.1 A Puzzle for Computer Science Educators

```
>>> (define (double x)
        (* x 2))
```

```
>>> (define (apply-to-5 f)
        (f 5))
```

The above examples present two seemingly parallel segments of Scheme code. First, the function `double` is defined to take a single input, `x`, and return the result of multiplying that argument by the number `2`. In the second example, the function `apply-to-5` is defined to take a single argument, `f`, and return the result of calling that argument on the number `5`. Despite the symmetry of these two types of functions, a surprising number of students (over 50% in one study to be reported here) have difficulty with the `apply-to-5` function. While students readily accept the idea that *numbers* can be arguments to functions they do not naturally extrapolate to conceive that *functions* can likewise behave as *data objects*. Determining exactly what accounts for this conceptual resistance was the original problem we sought to solve.

As it turns out, this general concept -- the notion of functional data objects -- appears in a number of important ways in other disciplines, including math and physics; educators in these areas report analogous scenarios. So, what began as a small educational dilemma in computer science has emerged as a window onto a much broader cognitive puzzle: *the notion of abstraction.* Solutions to such a problem must match its intimidating scope by transcending parochial methodologies.

1.2 Overview: Why an Imaging Curriculum?

> "In mathematics... we find two tendencies present. On the one hand, the tendency toward *abstraction* seeks to crystallize the *logical* relations inherent in the maze of material that is being studied.... On the other hand, the tendency toward *intuitive understanding* fosters a more immediate grasp of the objects one studies, a live *rapport* with them, so to speak, which stresses the concrete meaning of their relations...."
> [Hilbert and Cohn-Vossen 1932]

Historically, functional programming and higher mathematics have been characterized by their almost relentlessly textual and axiomatic style of teaching; the educational tradition for these topics is largely devoid of the rich "manipulatives" and visual images that characterize much of the most creative work in basic mathematics education. This work focuses on developing students' intuitions about abstractions through the use of imaging tools.

Abstraction is difficult: this is hardly a revelation in science education. However, the idea that manipulative-supported mental and motor imagery can be used as a "cognitive place holder" to navigate purely abstract ontologies is a non-standard application of skilled visualization. This study focuses on one particular concept in computer science and higher mathematics, that is, the idea of *functional data objects*. It extensively surveys students' misconceptions about this notion, creating an actual taxonomy of errors along the way. As a result of this analysis, several difficulties with abstraction emerge as focal problems; imaging techniques are then applied as a cognitive aid. The work described herein is based on a number of case studies in which the imaging curriculum was run with subjects between the ages of ten and fourteen.

Research in cognitive science is typically characterized by the melding of disciplines; the work described here conforms to this philosophy, blending its ideas primarily from

computer science, education and psychology. In an attempt to provide background and vocabulary in all three areas, we begin with an overview of the separate facets of the problem as they present themselves within the different disciplines.

1.2.1 Computer Science: Functional Programming

Informally, the main idea examined by this work is that *functions*[1] *are manipulable as units of data.* This concept, central to the whole paradigm of functional programming, can be traced back far into the historical foundations of computer science; for example, it is at the heart of Turing's universal machine concept [Turing 1950]. As will be discussed, this research has indicated that students consistently encounter difficulty with the notion of functions as data objects (FD).

A formal definition of FD can be found in Stoy [Stoy 1977]. FD adheres to the following four properties, common to *any* "first-class" data object in a programming language. Command of these four properties is facilitated by conceptualizing functions as data. A first-class object is one that:

1. Can be named
2. Can be passed as argument to a function
3. Can be returned as the result of a function call
4. Can form complex data structures

[1]For the purposes of this work, *functions* are defined in a decidedly "procedural" manner. Although strictly speaking the terms "function" and "procedure" have a subtle semantic distinction, they are herein used interchangeably and indistinguishably. This representation is most appropriate in the turtle-graphics setting; other alternative representations of functions (e.g. graphs, sets of pairs) are less viable in the current context of this work. Open questions remain about whether students who learn through this curriculum would be more or less challenged by those somewhat "purer" representations of functions.

The research reported in this thesis has indicated that properties 1-3 prove increasingly difficult for students to grasp[2]. Property 1 is generally not difficult for students: procedures, like any other data object, can have an associated name. In the case of numbers, this is parallel to the notion that a number can be defined by a variable name.[3]

Property 2 represents more of a cognitive leap. Students in traditional computer science curricula understand and even generate the notion that numbers can be passed as arguments to functions; the analog -- functional arguments to functions -- is more elusive, despite the apparent symmetry:

```
>>> (define (apply-double-to x) (double x))
apply-double-to
>>> (apply-double-to 5)
10
>>> (define (apply-to-5 f) (f 5))
apply-to-5
>>> (apply-to-5 double)
10
```

Property 3 is the most difficult for students to grasp. As will be discussed later in this report, there are several causes for this, the most compelling of which is students' inability to deal with FD anonymity. The following is an example of a function that returns another function as its output. Note that the result object has no associated name:

```
>>> (define (create-subtracter n)
        (make-procedure-object (x) (- x n)))
create-subtracter
>>> (create-subtracter 3)
#<PROCEDURE>
```

---

[2]Property 4 will not be examined by this work.

[3]Note however that Scheme semantics are an exception to the rule: in most programming languages, the name of a function is inseparable from the function itself. For example, in LOGO typing the name of a function of no arguments returns the result of calling the function and not the object it is bound to.

Below is a Scheme expression which combines properties 2 and 3. In several separate studies (to be described later in this work) performed with both graduate and undergraduate student Scheme programmers, over 50% responded incorrectly as to the outcome of the expression:

```
>>> (apply-to-5 create-subtracter)
```

Chapter 2 will examine in detail the nature of the misconceptions that account for such high rates of error in regards to this and other FD problems.

1.2.2 Psychology: Mental Imagery

Geir Kaufmann [Kaufmann 1988] identifies the central educational dilemma in imaging theory:

> "Conditions where imagery is most readily available may not be the same as the conditions where it is most strongly needed. Imagery may be most readily available in concrete tasks, but conceivably is most highly needed in abstract task-environments."

It is precisely the intention of this work to apply imagery techniques to more abstract concepts. Since imagery enhances comprehension—and since abstract concepts seem somewhat less suited to imaging—the curricular theory described in this document is founded on the belief that if we can enhance the imaging of abstract concepts in computer science to have the same functional power as the imaging of concrete stimuli, then we can potentially increase comprehension.

The nature and utility of mental imagery in psychology and philosophy has historically been a matter of great controversy. A good summary is presented in Tye [Tye 1991]. The argument is fueled by indeterminacy: is imagery a purely introspective phenomenon or can we use objective theories to qualify it? If we accept that imagery is scientifically verifiable then what exactly is the nature of its representation in the mind? Are mental images pictorial or linguistic? Such questions only scratch the surface of this debate (which is described at greater length in Chapter 6).

In this work, imagery is treated as an introspective notion and as a pedagogical tool; its representation is finessed. In order to be able to apply imagery as a tool and build a curriculum around it, it is not necessary to make a commitment to one or another representational theory; we need not agree on the exact status of imagery but merely on the fact of its introspectively verifiable existence and accepted utility as a cognitive tool [Hadamard 1944].

1.2.3 Math Education - Concrete Manipulatives

The curricular plan outlined by this work encourages students to remodel their view of functions as purely active entities through the use of imaging techniques. It further encourages students to "see" functions as units of data, similar in spirit to how most people come to view numbers as data objects. Not surprisingly, this turns out to be a non-trivial task for students. Interestingly, there is much to be learned about how to rectify the dilemma by studying how the object concept of number has historically and pedagogically developed: it likewise presents itself as a challenging cognitive task in early stages of development.

This work encourages students to visualize functions as they would any other type of information unit. To bolster the desired ontology, we require students to handle manipulatives—physical instantiations of data objects of various types. Hence, students interactively experience the object nature of functional data. This unique synthesis of motor and visual imagery theory is supported both by the psychological imaging theories cited above and further by analogy to mathematics education, where physical manipulatives have been quite successful in fostering object concepts of number in basic mathematics [Montessori 1956].

1.3 Organization of this Document

The work described in this document takes on two characters. First and foremost, it reports on extensive and original empirical studies of students working with FD problems. However, in order to solidly found this execution stage, the work necessarily explores and integrates extensive background research. The presentation of these two veins of research is structured in the following way:

- Chapter 2 focuses on curriculum development. It begins with a skeletal description of the design process which was developed for building imaging curricula. It then details each step of the process as it was used to structure the FD imaging curriculum.

- Chapter 3 introduces the subjects who took part in the study, as well as the methods of data collection employed. It outlines the curriculum plan used for FD.

- Chapter 4 presents two complete case studies of students learning to use FD through the imaging curriculum. It also points to interesting individual results from other case studies and summarizes the data from all 18 subjects.

- Chapter 5 analyzes the results of the data presented in Chapter 4.

- Chapter 6 provides extensive background research connects this work to the areas of mathematical manipulatives, visual and motor imagery, history of mathematical thought, misconceptions of FD, and traditions of computer science instruction.

- Chapter 7 projects the results of this research onto future endeavors and proposes a movement towards emphasizing intuitive learning.

2 Designing an Imaging Curriculum

What is the appropriate application for an imaging curriculum? As Kaufmann notes (Section 1.2.2), situations in which the target concept to be learned is abstract seem particularly wanting of imagery enhancing tools. How can educators encourage the use of imagery in students? This chapter outlines the system I developed to create a "blueprint" that can be used to tailor an imaging curriculum in some given specific problem area . The chapter begins with a generic template for building curricula to teach any target concept; this template is then applied to the concept of FD. Table 1 show a basic outline of the building process.

Table 1: Steps to Building an Imaging Curriculum.
This general outline provides an overview of the curriculum building process.

Step 0: Identify a pedagogical problem **Step 1:** Research problem conception **Step 2:** Classify results of Step 1 **Step 3:** Build imagery tools

2.1 How to Build a Curriculum to Build a Mental Image

As noted above, the first task in curriculum building is problem identification:

> Step 0: Identify a particular problem whose pedagogy is in need of improvement.

The problem might be notoriously problematic for students or educators; it could be a new topic whose comprehensibility is questionable. It should, however, be a problem whose source is at least intuitively perceived to stem from difficulties with abstraction. We will refer to this problem as the *target concept.*

The next steps are designed to help sharpen the researcher's intuitions which led to selecting the target concept:

> Step 1: Research students' problems and educators' concerns surrounding the target concept.

There are any number of approaches at this level of discovery. Clearly, it is important to survey and test students who are at the point of having been recently introduced to the target concept. For example, if we wanted to understand students' misconceptions about organic chemistry, then we might test college students in their first two years of school. It is not so obvious, however, that there is much to gain from questioning students outside of the range of schooling which applies to the target concept. For example, what are the thoughts of advanced chemistry students: do they remember struggling with the subject? What do experts have to say about the target concept? How is it perceived by professors of organic chemistry? On the other end of the spectrum, we might try teaching the target concept to younger students with less background. Here, we can see if behavior of the advanced students is paralleled. This kind of knowledge will be useful later in answering questions about the source of the misconceptions: is difficulty with the target concept intuitive or brought about by opposition to some other more simple and persistent (yet incorrect) model? What kind of *cognitive interference* or *cognitive assistance* is created by the body of knowledge that students have acquired over time (see also [Chi, Slotta et al. 1994])?

This is also an appropriate time to draw upon data from other disciplines: is there an analog which is equally problematic? For example, imagine we have decided to study the concept of recursion as it troubles students of computer science. We might also want to examine how a particular recursive concept in, say, physics is perceived by students of that discipline. Lastly, it may prove useful to search in both the "home" discipline and outside disciplines for historical examples of confusion or interest in the target concept.

Now, presumably, we have solid data to back up the intuitions from Step 0. Before we can formulate solutions, we need to classify the results of Step 1. This phase is characterized by two distinct activities:

> Step 2a: Create a taxonomy (hierarchy of difficulty) of specific problems from Step 1.
>
> Step 2b: Represent cognitive ontologies that symbolically reflect the difference between a mature and naive impression of the target concept.

In Step 2a we want to pinpoint the specifics of the difficulty: the level of detail at which we examine errors should be fairly fine-grained. For example, if we were examining students' difficulty with long division, it would be useful to note that students repeatedly switch dividend and divisor when divisor > dividend. The goal of Step 2b is to target the essence of the problem. The granularity in this sub-step should be just finer than "because of abstraction." So in the case of fractal geometry we might note that the difference between an expert and novice conception hinges on the ability to visualize what a set of non-integer dimension might look like. In other words, the ontology should anticipate the final visual curriculum.

The final step is also the most challenging and creative phase of the process:

> Step 3: Build tools that suggest visual images aligned with the cognitive difficulty of the problem area.

Step 2a involved identifying specific problems associated with the target concept; Step 2b required the researcher to denote the underlying principles that emerge from those difficulties. In this last -- and most implementationally challenging step -- we create objects which will be used to persuade students to correctly intuit a concept. To do this, we take the cognitive models (from Step 2b) and extract properties which distinguish the complete from incomplete mental models. Then, when creating objects to enhance visualization, we make sure the objects accent these properties. For example, suppose that we are trying to image a recursive function. A naive

model views recursive functions as syntactically altered iterative functions; to de-emphasize this and to encourage a model in which the critical feature of a recursive function is its ability to "regenerate", we might build an imaging tool which is a miniature xerox box, churning out copies at the press of a button (recursive call). The tools one builds may vary in nature according to their overall harmony with the problem area: they could be computational objects, concrete objects, textual objects, pictorial objects, or any combination thereof. One property might be best reified with a computer simulation, another with a two dimensional picture, and so on. The overall goal, then, of the tool developer is to identify, capture and represent the salient distinguishable features in the correct ontology.

Immediately the question arises: how does one know if the tools are built "correctly"? While this matter will be discussed in detail later, it is worth touching upon a simplified answer. The goal of a real-world constructed object is to encourage comfort with some mental object which is both abstract and consistent. If the job is well done, then students' new and valid intuitions about the target concept should steer them away from the specific mistakes categorized in Step 2a without encouraging any new misconceptions.

2.2 Building an Imaging Curriculum for FD

The following sections describe how each phase of the building process was implemented for the target concept of functional data objects.

2.2.1 Step 0 : Identify a problem

I first became intrigued by the notion that a function could be a piece of data through the research of Eisenberg, Resnick and Turbak [1987]. Their interviews with 16 MIT undergraduates enrolled in an introductory Scheme programming course confirmed and explored the different ontologies of FD that students held. In fact, on the sample question presented in Chapter 1, `(apply-to-5 create-subtracter)`, only six of the sixteen students surveyed initially responded with a correct answer[4]. What was perhaps more intriguing, however, was the consistent nature of the misconception reported by those students who were in error. Students reported that the above example would result in an error message due to the fact that it was "missing" some required execution information.

As my research will reveal, the "missing variable" phenomenon turns out to be the most frequently recurring problem in FD conception. In the interest of not getting ahead of the curriculum design process, however, it is sufficient to note that at this stage we have two intriguing leads: (1) a particularly difficult, fairly abstract concept, and (2) a strangely consistent set of misconceptions around this concept.

2.2.2 Step 1 : Research Problem Conception

So far, we have a established a lead which points us to the problem of FD as a candidate for image learning. Before beginning the curriculum building process, it is

[4]This result is all the more startling, given that the students were using a highly FD centered curriculum based on the popular text *Structure and Interpretation of Computer Programming* [Abelson and Sussman 1985].

necessary to further verify our intuitions by more exhaustively examining current and historical perspectives.[5] The examination process for FD was four-fold:

- Test undergraduate and graduate students of functional programming languages for FD concept acquisition;
- Interview instructors of subject areas relevant to FD (such as programming languages, artificial intelligence, abstract math);
- Observe younger students engaged in learning about FD;
- Research historical perspectives of FD in computer science and mathematics.

2.2.2.1 Novice Scheme Programmers

I performed two distinct studies of graduate and undergraduate students who had had some introductory Scheme programming experience.

The first study was performed on a sample of 11 students from three different but comparable backgrounds. They were students who recently completed a unit on Scheme in an undergraduate Programming Language course (2), students of a graduate Artificial Intelligence class that used LISP as a programming language (5), and students of a graduate course in computer science for cognitive scientists that used Scheme as its language (4). A summary of results to selected questions is presented in Table 2. Each of the three questions summarized is respectively related to each of the Properties 1-3 of first-class objects (section 1.2.1). Individual responses are paraphrased where interesting.

---

[5] In fact, some of the research I present in this section took place during or after the design phase. While this is not inconsistent with the iterative design methodology I will endorse, it should also be apparent why I recommend that the bulk of this background investigation be done prior to the design phase.

Table 2: Novice Scheme Programmers
Summary of results from a survey of novice Scheme programmers' aptitude for properties 1-3.

Sample Composition:	exposure to some Scheme instruction and programming	
Sample Cardnality:	11	
Example 1:		
Given:	`>>> (define number 5)`	
	`>>> (define (subtract-3 n) (- n 3))`	
Asked:	`>>> number`	
Answered:	"5"	10
	"error: need parenthesis"	1
Total wrong:	9% (1/11)	
Asked:	`>>> subtract-3`	
Answered:	"error: no argument"	5
	"definition or function body or lambda expression"	4
	"error: need parenthesis"	1
	"`subtract-3` is a function"	1
Total wrong:	55% (6/11)	
Example 2:		
Total wrong:	0% (0/11)	
Example 3:		
Given:	`>>> (define (create-subtracter n) (lambda (x) (- x n)))`	
Asked:	`>>> (apply-to-5 create-subtracter)`	
Answered:	"error: no argument"	2
	"error: no second function call for `lambda(x)`"	1
	"`(lambda (x) (- x 5))`"	1
	"procedure, as in (create-subtracter 5)"	1
	"nameless function"	1
	"error: `apply-to-5` not defined"	1
	"returns a function that returns `x-5` given `x`"	1
	"error: need 'x' to complete `lambda(x)`"	1
	no answer or ??	2
Total wrong:	64% (7/11)	

Example 1 contrasts students' views of number and function as object. All students correctly answered that the interpreter, when asked to evaluate the name `number`, would respond with the number object `5`. On the other hand, three out of seven students responded that the interpreter would return an error when asked to evaluate the name `subtract-3`. In all three cases, the misconception resulted from the incorrect assumption that we were attempting to *call* the procedure and hence were missing the argument (this is not an unlikely misconception: it is in fact what would happen in some programming languages). What these students were missing was the notion, parallel to the case of `number`, that we were simply asking the interpreter to look up a name and return the associated object. In other words, students lacked an object concept of function.[6]

In the second example, corresponding to Stoy's property 2, all students answer correctly about the use of a functional argument to a function. Granted, this is a fairly trivial example, the likes of which students have extensively worked with in class. It is appropriate here to touch upon the difference between *identification* and *generation* of concept. Students were able to identify the use of a function as a data object and as such present the correct solution. However, I claim that this activity is significantly less cognitively advanced than either the case where students are able to generate this use for functions or even elegantly apply that use in novel situations. As this research will demonstrate, the latter forms of learning take place through the imaging curriculum.

The final example gives students the definition of a function which returns a function as its result. They are asked to predict the result of an expression that both takes a

[6]The "call" protocol common to some other programming languages is inconsistent with an object model.

function as an argument returns a function as its result. Half of the students correctly answered that the output would be a new function. The other half of the students noted, in one way or another, that the function call would return an error because something was missing; one student went so far as to note that it "need[s] 'x' to complete `lambda(x)`". This example illustrates a common error. Students who get this class of problems wrong uniformly explain their answers with some notion of incompletion: some missing piece of data prevents the call from completing execution.

In the second study, 28 undergraduate programming language students were asked to write a series of short Scheme functions as part of a homework assignment. The questions were handed out after I presented two 1.25 hour introductory Scheme lectures. During the lectures, I emphasized the object nature of function and presented illustrative examples. No imaging tools were used. Tables 3-5 outline some interesting results.

Example 1 asks the students to redefine the semantically obscure `car` and `cdr` functions to have the new names first and rest, respectively. The most succinct way to do this is to merely rename the functions:

```
>>> (define first car)
```

The majority of the students solved the problem with the following code segment:

```
>>> (define (first l) (car l))
```

The first solution implies that students understand the concept that a function is a data object which can merely be renamed (consider the parallel numerical example: given `a` bound to `10`, the way to equate `a` with `b` would be `(define b a))`. Instead, students were unable to separate the function to be defined from its argument. So,

students defined a new function called `first`, which takes a single argument and then returns the result of taking the `car` of that argument. In this sense, the property that a function *does something* (to an argument in this case) takes precedence over the notion that a function is an object to be manipulated.

Table 3: Programming Languages Undergraduates: Part I.
Summary of results for a Scheme homework assignment given to undergraduate students of programming languages. These two questions test properties 1 and 3.

Sample Composition:	Undergraduate Programming Language class: exposure to some Scheme instruction and programming under my instruction.	
Sample Cardnality:	28	
Example 1:		
Question:	Redefine `car/cdr` to `first/rest` in the most concise way possible.	
Answers:	re-named procedure	10
	re-wrote procedure	3
	re-defined using args	13
	no attempt	2
Total wrong:	64% (18/28)	
Example 2:		
Question:	Write `make-nth-getter`, a procedure which, given a referent, `n`, creates a procedure which returns the nth element of a list.	
Answers:	correct	15
	no use of lambda	5
	"not different from `get-nth-elt`"	2
	wrong use of lambda `(define (mng (lambda (n) (gne x n)))`	6
Total wrong:	46% (13/28)	

In Example 3 (Table 4), students were given an approximation function for finding first derivatives; they were asked to write a derivative function in Scheme which, given some function, returned the approximation function (literally, not symbolically) for the first derivative. Students were also provided with examples of how the function they were to write would work. For example:

Table 4: Programming Languages Undergraduates: Part II.
Summary of results for a Scheme homework assignment given to undergraduate students of programming languages. These questions test properties 2 and 3.

Example 3:

Question:	Write a function which returns the first derivative of a given function using the following approximation: D1(x) =~ [F(x+h) - F(x)]/h], where h small , e.g. .0001	
Answers:	correct	12
	no use of lambda	9
	`(define (derivative fun x) ...)`	
	no attempt	7
Total wrong:	64% (16/28)	

Example 4:

Question:	How could you write a second derivative procedure?	
Answers:	correct	7
	no use of lambda	3
	need another formula	1
	use `nth-derivative` function	1
	wrapped extra lambda	5
	no attempt	11
Total wrong:	71% (20/28)	

Example 5:

Question:	Write a procedure to create derivative procedures of any order.	
Answers:	correct	8
	no use of lambda	1
	wrong body	5
	use `function-applied-n-times`	1
	"can't pass in an equation"	1
	"don't understand the question"	1
	no attempt	11
Total wrong:	68% (19/28)	

```
>>> (define (cube x) (* x (* x x)))
cube

>>> (cube 4)
64

>>> ((derivative cube) 4)
48.0012
```

The derivative procedure provides an elegant example of functional objects in Scheme since the process of taking the derivative of a function involves both using functions as data and returning functions as results. The Scheme code for programming a derivative function is a direct translation of the approximation given in Example 3 of Table 2:

```
>>> (define (derivative f)
      (lambda (x) (/ (- (f (+ x 0.0001)) (f x)) 0.0001)))
```

Despite the sample executions that students were given, nine of twenty-eight students didn't think they needed to use lambda (i.e. generate a new function); despite the ease of translation from mathematical notation to Scheme, seven more students didn't even attempt the problem. In Example 4 where they were required to write a second derivative function (the solution to which is to doubly apply the first derivative function) a total of eleven students did not even attempt the problem. Example 5 is an order of magnitude more difficult than Example 3: it asks the student to write a procedure which creates derivative procedures of any order. So, `(derivative-maker 4)` would return a fourth-derivative procedure. Only nine of the twenty eight students correctly answered the question. One student even noted that the question could not be answered since "you can't pass in an equation."

Perhaps the most difficult problem in the set asked students to use `function-applied-n-times` (a function which repeatedly applies another function to an argument for a specified number of applications) to redefine the `derivative-maker` procedure from question 5. To correctly complete this question, students need to understand that it is possible to have a function, `function-applied-n-times`, which

takes another function (in this case, `derivative`) and a number, `n` (in this case, the order of the derivative), as its arguments and returns a function, `derivative-maker`, which takes a number, `n`, as its argument and returns an unnamed function which essentially takes the `nth-derivative`. The six students who attempted this problem answered it correctly. What about this problem scared 22 students away from even attempting it? This question begins to bring us around to the notion of abstraction. There is a sense in which Example 7 is exactly one *order of abstraction* more difficult than something like an ordinary derivative function (which is itself an order of abstraction more difficult than, for example, an ordinary cube function). The idea of orders of abstraction will be discussed further in section 7.3.

Table 5: Programming Languages Undergraduates: Part III.
Summary of results for a Scheme homework assignment given to undergraduate students of programming languages. These are the most difficult questions testing property 3.

Example 6:		
Question:	Write a function which applies another function to its argument n times (`function-applied-n-times`).	
Answers:	correct	4
	correct with helper	5
	bad body	3
	no lambda	3
	incorrect function call	3
	no attempt	10
Total wrong:	68% (19/28) [or 86% (24/28)]	
Example 7:		
Question:	Use `function-applied-n-times` to redefine `derivative-maker`.	
Answers:	correct	6
	no attempt	22
Total wrong:	79% (22/28)	

2.2.2.2 Instructors of FD

Folklore within the functional programming community resoundingly agrees: students have a hard time with the idea that functions can be manipulated as data objects. Specifically, students of functional programming have difficulty with lambda expressions. One AI instructor, when asked to characterize student misconceptions, simply had this to say:

> "Well... I haven't really ever pushed them to see how much they've learned because I figured they'd never really get it at a deep level anyway."

The concept of functions as data is also prevalent in much of higher mathematics from calculus to group theory. I interviewed professors of mathematics to see if I might be able to gain different insights on the matter through the perspective of the mathematical community. Interestingly, the rhetoric was almost identical. Like the functional programming community, mathematicians identified this as a problem both of particular difficulty and great import. One professor of calculus had the following to say about his personal and professional experience with functions as data objects:

> **I**nterviewer: Did you or do you now have the notion that it's a piece of data?
>
> **M**athematician**1**: *Oh yeah, I mean once you do group theory you get well used to the idea that you want to think of things abstractly in terms of objects and your objects are functions - that's dandy. You learn to combine them and multiply them and treat them just like your elementary objects.*
>
> **I**: You're teaching calculus now - do you feel like your students have a good concept of this?
>
> **M1**: *No. They don't have a prayer.*

It is perhaps interesting to note that even as M1 speaks of his own impressions of functions he only ascribes to them properties which are "like" those of elementary objects; he never says that they *are* elementary objects. In a separate interview with a

professor of group theory (M2), the same kinds of personal struggles were made even clearer:

> "I can understand why this is hard because sometimes I have a problem with this myself... I came face-to-face with the problems people have seeing functions as input, or output, in teaching Fourier analysis where functions are *both*. Students *uniformly* get confused because you're not processing *numbers*, but *functions*."

Even as an instructor of group theory, which hinges on the notion that functions are data, M2 admits his own confusion with the concept. Further, when the definition of the create-subtracter procedure[7] was presented to M2, his behavior modeled that of the novice Scheme programmer exactly:

> "Well, you couldn't use `create-subtracter` because you don't know what `x` is yet. "

This represents an instantiation of a dilemma that was previously alluded to: the missing variable problem. In this case, the subject naively notes that a variable has been used in the body of the procedure which has not been declared as an argument. In fact, this variable is the argument to the result procedure which is bound at execution time. Students lack the familiarity with abstraction which is necessary to treat that variable as merely a symbolic place holder until after execution time: if it is without concrete value at run time, then the function is not executable.

What did these investigations reveal?

- Conjectures that this was an area of great difficulty for students were confirmed.
- Suspicions that this was an area where pedagogy was not sufficient to match the difficulty of the task were confirmed.
- Even the behavior of experts sometimes still supports a naive model of functional objects.

---

[7]Recall: `(define (create-subtracter n) (lambda (x) (- x n)))`

2.2.2.3 Younger students

Interviews with instructors of functional data indicated that even at their advanced level of knowledge there was some confusion and misconception about functional data. One question which immediately arises is: how much of one's ability to conceptualize functions as data is hindered by other contradictory information which has "cognitively accumulated" over time? For example, is the average student of functional programming at a disadvantage for already having, most likely, become proficient at programming in another paradigm which neglects constructs for functional data objects?

In order to answer this question, we turn our study to the unfettered experiences of pre-high school students. The bulk of this work involves repeated execution of the imaging curriculum with this type of subject. However, a few case studies were performed prior to introducing the notion of imaging in order to judge younger students' impressions and abilities. Subjects worked with SchemePaint [Eisenberg 1991], a graphics enhanced version of Scheme.

I first explored this concept with three 7th graders. In the initial run, I worked with two girls who had no prior programming experience. They had great difficulty picking up the skill of programming, hence we did not comfortably approach the subject of higher-order functions. Perhaps the most salient observation that could be made about the experience was this: a major road block in their learning came with the notion of arguments to procedures. They first had difficulty understanding that the formal name of an argument was merely a place holder until run time when an actual value could be associated. This misunderstanding is the first in a series of related misconceptions centered on the notion of abstraction. To understand formal

parameters, one has to be comfortable with the idea of abstracting a set of potential actual values into an abstract symbolic value. This is a milder instantiation of problems seen in students working with lambda expressions.

In the second scenario I worked with one male student, Hector[8], who had some limited experience with the C programming language. He learned basic Scheme in a more accelerated fashion than the previous two subjects, perhaps due to his familiarity with basic programming concepts. Within about two hours of work, I was able to introduce functions as arguments to other functions. He seemed to understand my explanation. However, when he was asked to apply the concept, he was initially unsuccessful, perhaps in part due to his lack of experience manipulating procedures in this fashion in a programming context.

In one example scenario, he starts by writing the two procedures shown below:

```
>>> (define (spike length)
       (rt 20)
       (fd length)
       (rt 140)
       (fd length)
       (rt 20))

>>> (define (hex-w-mover mover length)
        (repeat 6
            (mover length)
            (rt 60)))
```

Immediately after composing these two procedures, he was asked to make a six sided star. Instead of simply calling `hex-w-mover` using `spike` as an argument, i.e. `(hex-w-mover spike)`, he wrote the following code from scratch:

[8]All students' names have been changed to preserve anonymity.

```
>>> (define (star length)
       (rt 30)
       (fd length)
       (rt 120)
       (fd length)
       (rt 150)
       (fd (+ 10 length))
       (rt 150)
       (fd (+ 10 length))
       (rt 150)
       (fd (+ 10 length))
```

Eventually, Hector became very comfortable with functions as arguments. In one session, he was given a "mystery" procedure, along with several examples of how it could be used:

```
>>> (mystery fd)                    {move turtle forward 90 units}
90
>>> (mystery rt)                    {turn turtle 90 degrees}
90
>>> (mystery square)                {make a square with sides of size 90}
8100
```

At this point, he stopped me from demonstrating and began to work himself. The following represents an actual transcript of his work.

```
>>> (mystery fd)
90

>>> (mystery rt)
180

>>> (define (it mover)
       (mover 90))
it

>>> (it fd)
90

>>> (it rt)
270
```

When asked to describe in words what the `it` procedure does he replies "It does what you tell it to to 90." Note that his description is still very much based on the numerical component, not the procedural one.

In his next phase of work, Hector learned to use and create a new kind of data object called a color object. This introduced many interesting phenomena which do not come up in non-graphics Scheme. Color objects are complex data structures composed of three integer values, 0-65000, each representing the relative intensity of red, green and blue in a particular color. Color objects can be created through the SchemePaint `make-color-object` procedure. This procedure returns an anonymous color object, similar in vein to the way that `lambda` returns an anonymous procedure object:

```
>>> (make-color-object 60000 0 60000)
(60000 0 60000)
```

Students initially believe that the above statement would result in an error message from the interpreter. On the other hand, students are in agreement that the following code should work:

```
>>> (define purple (make-color-object 60000 0 60000))
purple
```

The difference between these two segments of code is anonymity: in the first statement, the result object has no name associated with it. Since standard Scheme really only has one commonly used anonymous object, the procedure object returned by a `lambda` expression, it had not previously occurred to me that the difficulty with procedures as objects might really be related to a more general problem with object anonymity.

Shortly after the introduction to color objects there was a three month hiatus in my work with Hector. Prior to this break, Hector took a written test in which he demonstrated working knowledge of properties 1 and 2, and some insight into property 3 (at the time of testing, this was brand new material); when we resumed,

Hector retook the identical test. His knowledge of properties 1 and 2 remained intact while his insight into property 3 seemed lost.

Hector was informed that his next project was to build a "library" of SchemePaint functions which performed manipulations on color-objects. He was first asked to write a series of color transformation procedures which, when given a particular color-object, make a new color-object with increased or decreased amounts of red, green, or blue. The following is an example of two of the six procedures that he wrote:

```
>>> (define (darken-red color-object)
       (make-color-object
          (+ (get-red color-object) 1000)
          (get-green color-object)
          (get-blue color-object)))

>>> (define (lighten-red color-object)
       (make-color-object
          (- (get-red color-object) 1000)
          (get-green color-object)
          (get-blue color-object)))
```

Hector wrote four other similar procedures: `darken-green`, `lighten-green`, `darken-blue`, `lighten-blue`. He was then asked to note that `{darken,lighten}-{red,green,blue}` were nearly identical; based on this observation he was asked to change the six procedures into three procedures. This task entails turning the {+,-} operator into a procedural argument (property 2). He wrote three new procedures, `change-{red,green,blue}`, without difficulty:

```
>>> (define (change-red direction color-object)
       (make-color-object
          (direction (get-red color-object) 1000)
          (get-green color-object)
          (get-blue color-object)))
```

Hector was then asked to write a number of other functions which operated in a similar fashion, that is, given a color-object as input, perform some manipulation on the color and return a new and different result color object. After completing this

task, he was asked to write a program which, when given a color-object and any two of his 15 transforms, sequentially applies the two transforms to the old object and returns a new color. This task also involves correct understanding of property 2. The following is his initial attempt:

```
>>> (define (apply-transforms color-object transform1 transform2)
       (transform1)
       (transform2))
```

This represents an incorrect use of procedural arguments. Although Hector correctly reasoned that he needed to pass in two transforms and a color-object as arguments, the body of the code is non-functional: he forgets to indicate that the transforms must themselves take the color-object as an argument when invoked, i.e `(transform1 color-object)`. This represents an instantiation of a common confusion about the role played by functional arguments to functions. After running the procedure, Hector quickly realizes what he did wrong, but it is interesting to note his initial instincts nevertheless.

His next task required a conception of property 3. Outside of the two exams he has been given, this is the first time Hector is required to generate such ideas. He is asked to write a procedure called `compose` which, given two transforms, creates a new procedure which, when applied to a color-object, returns the result of applying those two transforms to the given object. Upon contemplating the task, he incrementally exhibits three stereotypical misconceptions:

1. *"That is the procedure we just wrote."* This statement indicates a misconception about the object-ness of procedures. Hector does not conceptualize the procedure's ability to be returned as an object, hence he misconstrues this as identical to the previous task (`apply-transforms`) which emphasized knowledge of the more common ontology of procedures as active, applicative entities.
2. *"You can't do it; the [result] function wouldn't have a name."* This statement is illustrative of the common difficulty that students have with anonymous objects. The function Hector was asked to write would return a

nameless object, unusable without other context. Recall that earlier in the curriculum he had the same difficulty with anonymous color-objects (the result of all the transform procedures he wrote). This reaffirms the theory that the difficulty is not just with anonymous procedure-objects, but anonymous objects in general. As stated earlier, this distinction was previously unobserved since most functional languages have only one type of anonymous object - functional. The use of SchemePaint (and hence different sorts of anonymous objects) has pointed to a more global misunderstanding.

3. *"You can't do that; its missing a variable."* This statement is identically uttered by nearly all students attempting to grasp property 3. He is referring to the fact that the new procedure will require a color-object as an argument, and we have not provided this data anywhere when we create the procedure. This is what we have previously referred to as "the missing variable phenomenon."

Lastly, a somewhat more general problem was pointed out by Hector. He noted an inconsistency between the way the Scheme interpreter seems to view color-objects and procedure objects. When one asks the Scheme interpreter what a particular color-object is, the contents of the object are returned; when one asks the Scheme interpreter about a procedure object, the name is simply echoed, indicating that the procedure exists; no indication of content is given.

```
>>> purple
(60000 0 60000)

>>> darken-red
#<PROCEDURE darken-red>
```

We had already noted the potential confusion with `lambda` expressions and accordingly renamed the `lambda` special form to the more semantically accurate `make-procedure`. However, Hector's observation about the inconsistent display of procedure and color result objects brings attention to just how sensitive students are to syntactic and semantic details which cannot be reasoned from past knowledge.

To summarize, several new details about students' misconceptions of functions as objects have been learned through this case study of a novice programmer;

additionally, this study has hinted that the behavior of younger students more or less parallels that of adult students, making them good candidates for future study.

2.2.2.4 Historical Perspectives

By now, there may be an emerging dilemma for the reader: if, as we have shown, this is such a systematic problem for students in both mathematics and computer science, then why hasn't it been previously addressed? Is the concept that a function can be a data object necessary or important? Is it perhaps so narrow in focus as to constitute generally useless information? In order to address these concerns and confirm that this is an important concept of broad utility, we turn to the writings of some of the major figures in mathematics and computer science history.

As early as 1836, in his notes, Charles Babbage pondered blurring the distinction between data and operations in this early passage about the possibility of an Analytical Engine:

> "This day I had for the first time a general but very indistinct conception of the possibility of making an engine work out algebraic developments. I mean without any reference to the value of the letters. My notion is that as the cards (Jacquards) of the Calc. engine direct a series of operations and then recommence with the first so it might to punch others equivalent to any given number of repetitions. But [their holes] might perhaps be small pieces of formulae previously made by the first cards." (quoted in [Randell 1973])

In this selection, Babbage seems to be hinting that cards (operations) might in fact direct (generate) certain other operations.

Several years later, Ada Augusta considers Babbage's theoretical machine. By clearly demarcating operations and data, she exhibits the same naive understanding about operations (functions) which we saw in present day students of function:

> "In studying the action of the Analytical Engine, we find that the peculiar and independent nature of the considerations which in all mathematical

> analysis belong to *operations*, as distinguished from the *objects* operated upon and from the *results* of the operations performed upon those objects, is very strikingly defined and separated." [Babbage 1842]

Babbage returns to the issue, admitting its importance yet acknowledging that, for him, it presented an unconquerable challenge:

> "I am unwilling to terminate this chapter without reference to another difficulty now arising.... The extension of analysis is so rapid, its domain so unlimited, and so many inquirers are entering into its fields, that a variety of new symbols have been introduced, formed on no common principles....
>
> A few months ago I turned back to a paper in the Philosophical Transactions, 1844, to examine some analytical investigations of great interest by an author who has thought deeply on the subject. It related to the separation of symbols of operation from those of quantity, a question peculiarly interesting to me, since the Analytical Engine contains the embodiment of that method. There was no ready, sufficient and simple mode of distinguishing letters which represented quantity from those which indicated operation....
>
> Although deeply interested in the subject, I was obliged, with great regret, to give up the attempt; for it not only occupied much time, but placed too great a strain on the memory." [Babbage 1842]

A century after Babbage's initial musing about the self-modifying engine, Turing outlines the specifics for a machine which uses the specification of some other machine as data:

> "Let us first suppose that we have a machine M' which will write down... the successive complete configurations of M.... It is not difficult to see that if M can be constructed, then so can M'. The manner of operation of M' could be made to depend on having the rules of operation (i.e., the standard description) of M written somewhere within itself (i.e., within M'); each step could be carried out by referring to these rules." [Turing 1937]

Turing's ideas are in fact an elaboration of the same function-versus-object theme that is identifiable in the great mathematician Gödel's technique of representing proof sequences as numerals [Gödel 1931]. By 1945, Von Neumann constructs the machine which will use calculation specifications (functions) as input. In Von Neumann's system, functions take on the same form as any other data object:

"Any device which is to carry out long and complicated sequences of operations (specifically of calculations) must have a considerable memory. At least the four following phases of its operation require a memory:

> (a) Even in the process of carrying out a multiplication or division, a series of intermediate (partial) results must be remembered....
>
> (b) The instructions which govern a complicated problem may constitute a considerable material, particularly so, if the code is circumstantial (which it is in most arrangements). This material must be remembered....

To sum up... The device requires a considerable memory. While it appeared, that various parts of this memory have to perform functions which differ somewhat in their nature and considerably in their purpose, it is nevertheless tempting to treat the entire memory as one organ, and to have its parts even as interchangeable as possible..." [Von-Neumann 1945]

At the very least, these quotations tag functional data as central to the very core of computer science. Further, we have confirmed that this is a concept which is stimulating, perplexing and important to great as well as novice thinkers.

2.2.2.5 Summary

The goal of this initial phase of evaluation was to develop a comprehensive understanding of the target problem. Our varied investigations have revealed that throughout history and across the different disciplines which utilize functional data objects, students and scholars encounter cognitive barriers at exactly the points where higher-order procedure processing is necessary.

2.2.3 Step 2: Classify Results

2.2.3.1 Step 2a: Outline a Taxonomy of Error

So far, then, in the attempt to apply the imaging curriculum to FD, we have completed two step of the process:

Step 0: Identify a pedagogical problem.
Step 1: Research problem conception.

The goal of Step 2 is to classify the results from Step 1. Historically and currently, research from students and educators indicates the following principle of general categorization:

Properties 1-3 [cf. Stoy] of functional data objects represent strictly increasing levels of difficulty for students. Specifically:

i. Property 1 is not generally problematic.
Students, if anything, have more difficulty assigning names to numerical data than procedural data. In fact, functional data "feels" like it is required to have a name in order to have meaning whereas students are comfortable with the independent (and notably abstract) existence of numbers. Still, understanding that the name `subtract-3` would represent a function just like the name `number` represents a number was not completely transparent for students (Table 2, pg. 17).

ii. Property 2 is more difficult.
The concept of arguments of any sort is itself abstract and difficult for students. Beyond that, without the same practice with operating on functions that students possess for operating on numbers, the quandary of functions as arguments reduces to an experiential one. Both linguistically and mathematically, subjects are more accustomed to treating numbers as the default computational unit.

iii. Property 3 is most difficult.
Lambda expressions, functions which return other functions as result objects, are hard because they require an extra level of abstraction beyond that which is demonstrated by an argument to a procedure. One side effect of this is that students perceive that these expressions are incompletely specified (missing variable phenomenon). Lambda expressions are also hard because (1) they can employ the concept of object anonymity and, (2) because statement semantics are obscure and suggest few fruitful analogies.

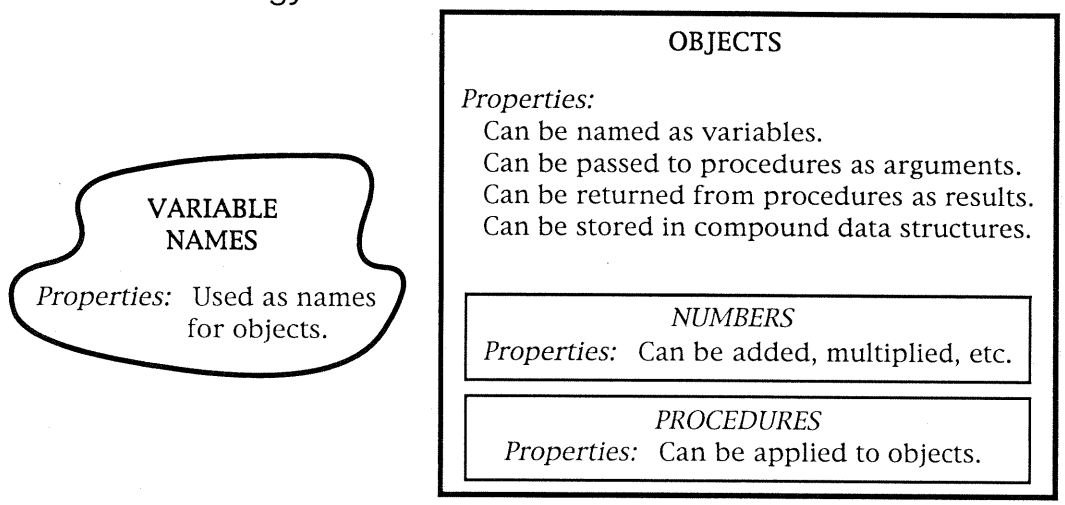
2.2.3.2 Step 2b: Build Cognitive Ontologies

We would like to suggest that the details of misconception outlined in the previous section represent pieces of a puzzle. All the pieces of the puzzle can be fit together to produce various models of conception (similar in vein to [Chi, Slotta et al. 1994]; see also section 6.3 of this document). Many projects have undertaken the task of specifying mental models, particularly in relation to concepts in the physical sciences [Gentner and Stevens 1983]. This work takes a similar approach to "mental modeling".

Figure 1 exemplifies an ontological picture, borrowed from [Eisenberg, Resnick et al. 1987]. The complete ontology of data stresses the notion that there are two kinds of things: objects and names for objects. Objects come in different types; those types have certain shared and different properties. The naive model paints a different picture: procedures are their own sort of entity, different and removed from data objects. They are characterized solely by their property of "activity"; they are not independent units but rather they are incomplete without numbers to manipulate. These models are consistent, respectively, with students who answer accurately and inaccurately to applicative questions about functional objects in Scheme.

It is claimed that while the incomplete ontology prevails, so do naive biases about the role that functions can play in a language. Thorough and complete understanding of the specific errors outlined in the previous section depends upon a incorruptible cognitive ontology. Figure 2 illustrates the difference in status for functions under both a complete and a naive cognitive model. In Figure 2a, the correct model, the world is made up of data objects; data objects can be of many different types. In the naive model (Figure 2b), the world is made up of data and procedures. In the latter

"correct" ontology:



"naive" ontology:

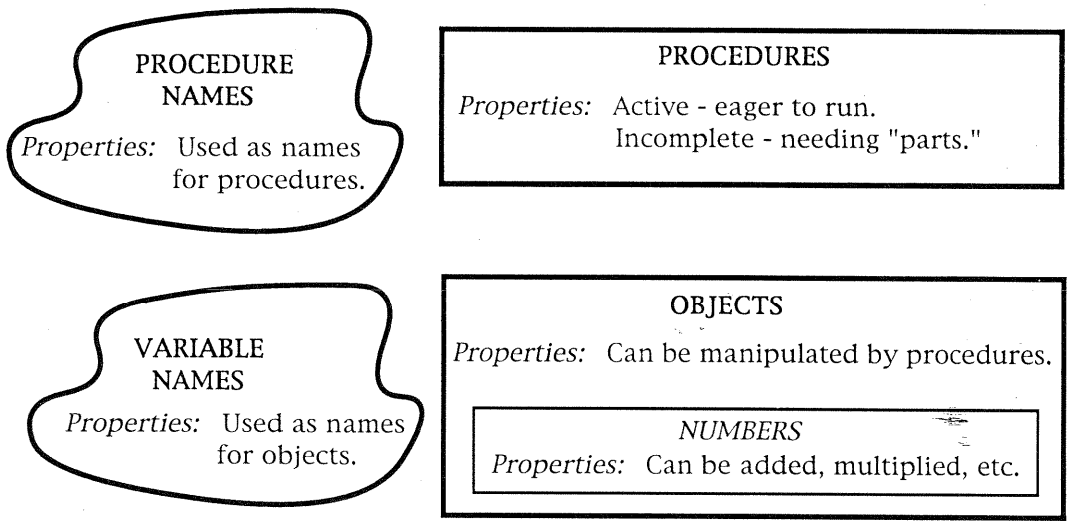


Figure 1: Correct and Naive Ontologies of Functional Data
This diagram, borrowed from [Eisenberg, Resnick et al. 1987], describes students' naive and expert concepts of function in Scheme.

case, procedures are assigned some special status such that they are not candidates to be data objects.

This naive ontology is preconditioned in a number of ways. For example, most imperative computer languages -- the paradigm used most often to introduce students to programming -- lack the constructs to support functions as first-class objects. Even natural language makes a clear distinction between nouns ("person, place or thing") and verbs ("action words"). Chi points out that conceptual category change is most difficult when the naive categorizations are deeply rooted through this type of persistent, consistent and recapitulated support system [Chi, Slotta et al. 1994].

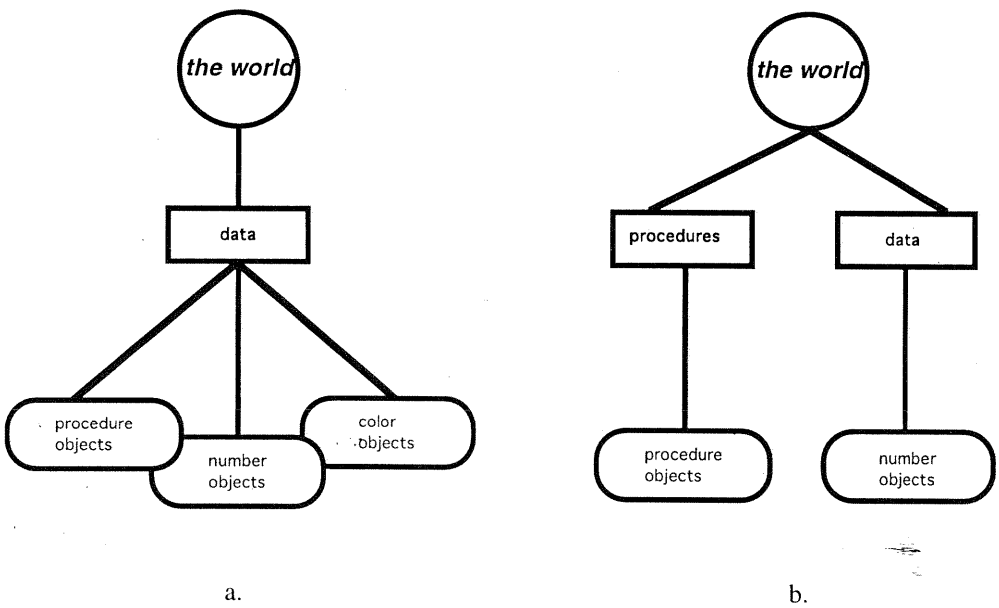


Figure 2: High Level Model of Functional Data
This figure describes the difference in status for functions under a correct (a.), and a naive (b.) cognitive model.

2.2.4 Step 3 : Building Imaging Tools

This last step of the design process is by far the most challenging and creative. It is at this stage that the researcher must, in fact, abstract certain key principles which are

mandatory for a non-naive understanding of the target concept. If the tools are designed correctly, they will encourage a complete and consistent ontology of the target concept; students can then draw on this ontology to correctly intuit solutions and simultaneously avoid the standard errors described by the taxonomy (Step 2a).

Based on the models from the previous section, we would like students to develop a comfortable ontology of data objects which is consistent with Stoy's properties 1-3. Note that there is no specific mention of functional data objects in this initial goal. Rather, the idea is for students to develop a rich object concept of all data, procedures being one of several classes of data objects. In order to foster this concept, we will try to enhance skills of mental imagery and visualization by matching an appropriate modeling system to the target concept. The tools of the imagery enhancing modeling system may be *multi-modal* in nature; that is, we do not limit students to one class of tools for visualization. Research has shown that some students have a predisposition towards one or another style of learning [Johassen and Grabowski 1993]. Similarly, students may favor one class of visualization tool over another. Additionally, different tools may be more or less pedagogically expressive for purveying different aspects or qualities of the target concept. For the purposes of this work, we use three modes of imaging tools:

- Computational - The standard programming environment. It is often claimed that computers allow students more tangible access to abstract concepts [Papert 1980]. However, as the data clearly indicates, programming languages have proved insufficient in the case of functional data.
- Textual / Visual - A two dimensional pictorial view of objects in Scheme written in large scale on a dry erase board.
- Concrete - Three dimensional tangible objects which students can hold and manipulate.

We now turn to a more detailed description of how the three properties interact with the three modes of imaging.

2.2.4.1 Property 1: Objects Can Be Named

A combination of all the imagery enhancing techniques are used to enforce the fact that procedures, like all objects, can be named. The curriculum begins with a discussion and analogy forming session about how this property is applicable to everyday objects. Students are asked to reason about why the following three statements may or may not be true:

- Everything is an object.
- Objects can be named.
- Objects have properties: some are shared among objects, some are different.

Students are asked to challenge preconceptions, particularly about what qualifies as an object. For example one student, when thinking about the first principle, responds that everything is an object "except thoughts". In this case, the discussion turns to a challenge of this bias: thoughts can be remembered, forgotten, verbalized, written, etc.; thoughts too are things we manipulate. When discussing the naming of objects, students are asked to think about the names of objects in terms of the roles they play in situations. For example, students might agree that a paperweight is a name for an object. They must further reason, however, that many objects can perform the role of a paperweight (a book, a rock, a cup). They are encouraged to note that these objects can have different and shared properties: all of the aforementioned objects can be paperweights; only a book can be read; only a cup can hold water, etc. Lastly, students are asked to think about what it means for objects to be *active* or *passive*, and whether the computer or its programs should be classified as one or the other.

Immediately following this, the curriculum commences with a discussion of objects in the computational (Scheme) world. Students are introduced to the 2-D visual representation of the *name-object table* (see Figure 3). This representation

encourages the idea of "bindings" between names and objects. The drawing reduces the whole processing system to table additions and inquiries. This helps reinforce the notion of naming as being an equal status property since there is a uniformity of structure and processing for all the different types of objects. However, since the 2-D drawn objects also each contain different sorts of information relative to their type (e.g. a value for number objects, an input list and body for procedure objects), it allows the student to likewise comprehend that objects can have unique properties.

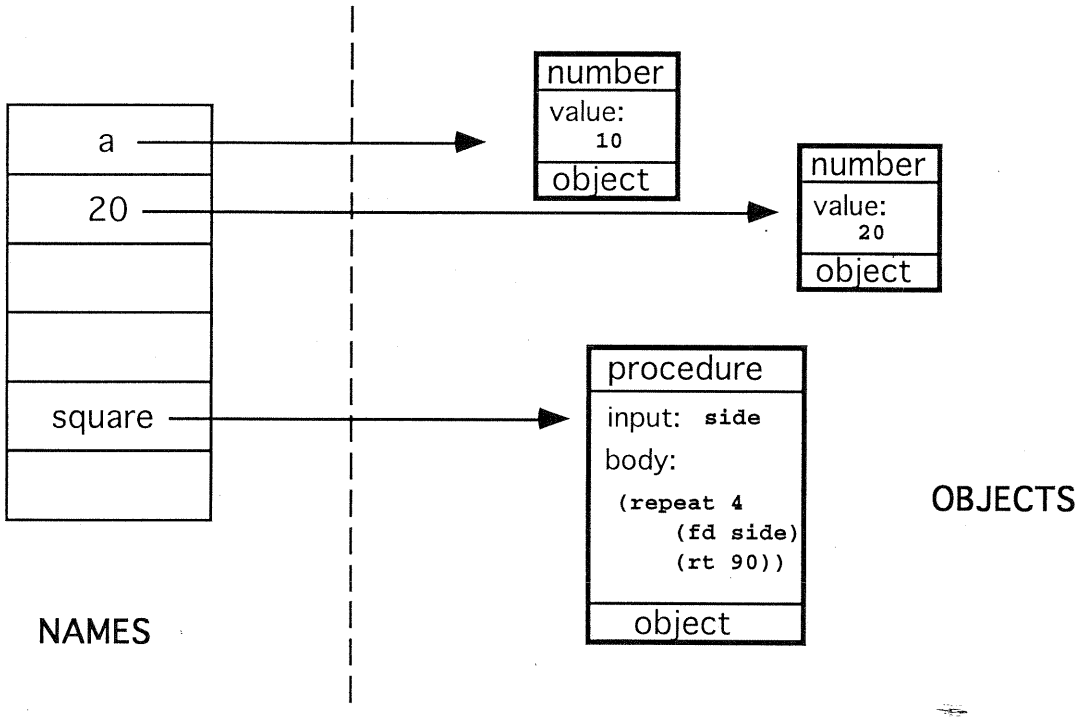


Figure 3: The Name-Object Table.
Students are first introduced to the behavior of the Scheme interpreter through the *Name-Object* table. All transactions in Scheme are said to take place through table "look-ups" or insertions.

The 3-D manipulatives (pictured in Appendix A) are most effective for reifying principle 1: everything is an object. The manipulatives are objects in the most literal of senses: small felt pillows each with its own textual information inside (miniature white boards with a small copy of the 2-D representation), relevant to the type of the object. The objects all share certain physical properties (they are all felt pillows, for

example) but they differ in others (color and size). This helps foster principle 3, again, at a very high but concrete level. The objects are less useful for formulating the notion that computational objects may have names; rather, due in part to their "container-like" independence, we will see that they are a good model for object anonymity.

The name-object binding expressions need to be as consistent as the physical representations of the objects. Note the syntactic difference between naming number, color, and procedure objects in standard SchemePaint:

```
(define x 5)
(define (double x) (* x 2))
(define purple
   (make-color-object
      (40000
      0
      40000))
```

In order to avoid confusion about syntax interacting with confusion about meaning, we adopt a universal name-object binding form: `(define` *name object-expression*`)`. In order to semantically separate the procedure object name from its argument (again, to encourage an independent view of the object) we parallel the `make-color-object` syntax:

```
(define double
   (make-procedure
      (x)
      (* x x)))
```

Note that replacing the inarguably obscure `lambda` expression with `make-procedure` also averts confusion that commonly arises from the semantically irrelevant nomenclature.

2.2.4.2 Property 2: Objects Can Be Arguments to Functions

2-D textual drawings are useful for helping students work through the details of how arguments to functions, and hence procedural arguments, work. Through the white board representation, students are able to follow execution rules (Figure 4) and perform direct name/object substitutions by merely wiping out a name on a white board and replacing it with the actual value of the argument (Figure 5). Still, this representation is only mildly imagery-enhancing for procedural arguments.

Scheme Expression Evaluation Rules:

Rule 1: *The Number Rule*
Numbers evaluate to themselves.

Rule 2: *The Procedure Rule*
To evaluate a procedure call:
(1) Look up the inputs in the name table.
(2) Look up the procedure object in the name table.
(3a) Associate real objects with input names.
(3b) Execute the body using the new input values.

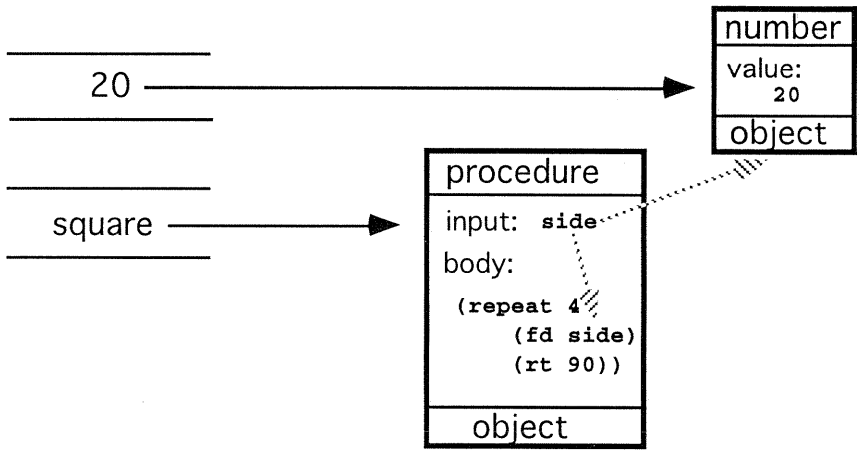
Figure 4: Scheme Expression Evaluation Rules
Students use physical and visual manipulative representations to work through the evaluation of expressions according to certain interpreter rules. Two of those rules, applying to number and procedure objects, are listed.

3-D representation is most helpful at this stage when thinking about procedures as arguments. Physical objects strongly encourage the notion of "passing" inputs to functions as the manipulatives themselves can be, literally, passed around. One student notes that when a procedure executes, we "throw" the interpreter a name and it "throws" back an object for use. Students come to think of the object itself as being part of the procedure, without common confusion about actual vs. formal parameter names. Further, the idea that we can "throw" a procedure object in the same way that

we can throw a number object further fosters a uniform view of objects in the language.

Question: *How does Scheme interpret the command:* `(square 20)` ?

Step 1:



Step 2:

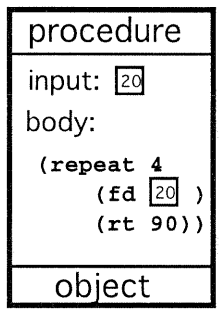


Figure 5: Visually Interpreted Command
Students use the visual representation of objects to work through the evaluation of Scheme expressions. Above, the command `(square 20)` is evaluated. In Step 1, the student looks up the value of the names `20` and `square` in the name/object table. With both these pieces of information available, the student can associate the number object `20` with the symbolic name `side`. By Step 2 the procedure body is fully specified and the student can sequentially execute the commands.

2.2.4.3 Property 3: Objects Can Be Returned Values of Functions

For property 1, the textual representation helped focus attention on the textual nature or "guts" of the object that we were defining or looking up in the table. For property 2, concentrating on the physical manipulatives helped accentuate the embodiment of the data: the fact that the data object could be passed around as a unit. In order for students to thoroughly understand property 3, we need to draw upon both these views.

Recall that one of the major sub-problems with procedure objects was their ability to exist without names. The 3-D object representation lends plausibility to the idea of anonymity. Students speak of -- and synonymously gesture with their hands -- the idea that anonymous "floating" objects can exist in the Scheme environment. The understanding that a procedure can be the result returned from a function call is intertwined with the understanding that procedure objects have the property that they can be "freshly" generated (through the use of `lambda`). Procedure objects are containers for their own specifications. As we noted earlier, the concrete procedure objects encourage the idea that procedure objects are capsules of information. The white board representation also helps students visualize how procedures are "born" and what information they contain (note that the idea of "containing" is consistent with "objectness") in virtue of the rectangular box structure of the objects. The fact that objects contain something similar to the pure programming language specification (Figure 6) meaningfully links the idea that a procedure is an active specification (better portrayed by the computational representation) with the idea that it is a passive object (better represented by the 2-D and 3-D descriptions). A new procedure is a "blank slate": literally, a blank white board instead of the "pre-packaged" object one might see for a built-in Scheme program. This allows students to see how it is possible that the specification for a procedure are not worked out (in this case, "written-in") until execution time.

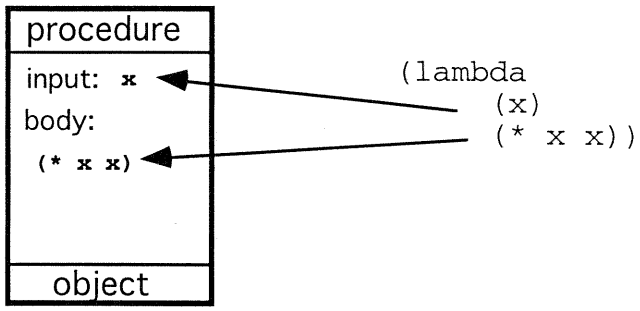


Figure 6: Visual Specification of a Scheme Code
The visual representation of a procedure object is a direct translation from the associated `lambda` expression. Arguments which are specified within an initial set of parenthesis in a `lambda` expression are written into the input section of a procedure object. All other sub-expressions in the `lambda` expression are translated to thebody of a procedure object.

Misunderstandings about "missing variables" from lambda expressions are hindered by the process of using object manipulations. Valueless variables in a procedure-generating procedure are consistent with the real object generation process. The visual aid of the literal procedure and number objects demonstrates that the procedure-generating procedure can legally execute without the information filled into those slots until a later time (Figure 7).

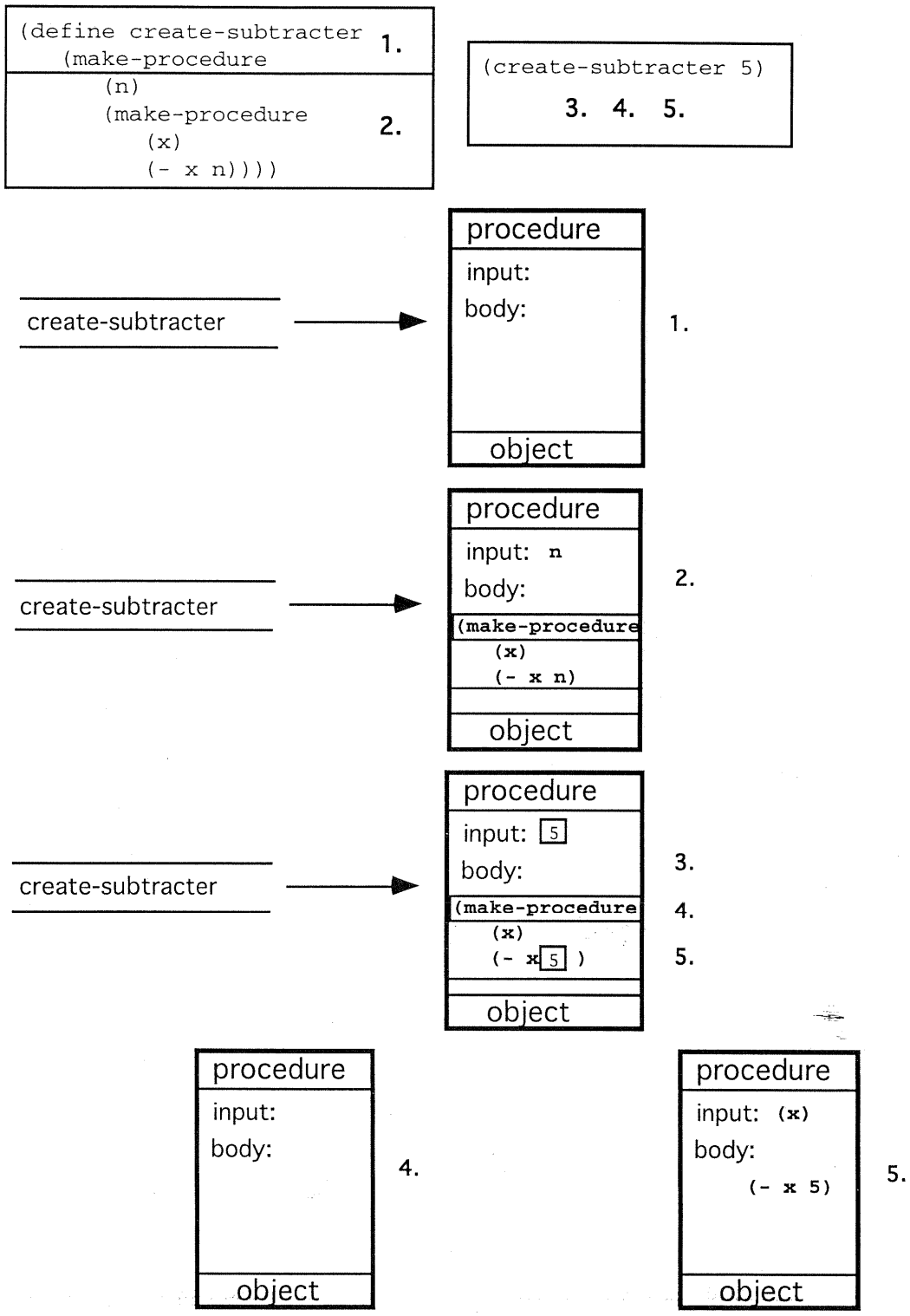


Figure 7: Visual Aid for a Procedure-Generating Procedure
In this example, the `create-subtracter` function and invocation are translated into a visual object representation. Steps 1-5 correspond to important stages where actions are performed on the visual representation. Note that the process emphasizes the birth of new objects.

3 Subjects and Methodology

3.1 Style: Case-Study Based

The research presented in this work was performed in a case-study style. Extensive time was spent with each student (compared to a standard classroom environment) permitting the researcher to have a familiar rapport with the student. In general, this style of data collection is founded on the principle that a student's particular situation -- their collection of experience over the years -- is relevant to data collection and analysis. The results of the preliminary study described earlier (with Hector: section 2.2.3) as well as the diverse and consistent array of errors illuminated by the background research, served as the basis of comparison for the imaging curriculum.

3.2 Setting

The researcher worked on an in-depth level with subjects in a mutual learning environment: students were advised that they were "teachers of learning," instructing the researcher as they proceeded as to areas of difficulty, ease, and interest. Conversely, students were treated as explorers -- they were encouraged to be inquisitive and develop and pursue their own lines of interest.

The setting for the curriculum was a small research office at the University of Colorado, Boulder containing:

- 1-3 Macintosh computers running SchemePaint
- 1-2 large dry-erase boards
- Concrete manipulative objects
- Students and researcher

Students worked one-on-one or two-on-one with the instructor, as it was determined in the pre-design phase that this student to instructor ratio worked best for minimilizing student distraction.

3.3 Subjects and Instructor

All sessions of the imaging curriculum were taught exclusively by me. Prior to this time, my only teaching experience consisted of weekly recitation lectures in an introductory programming course.

A total of 18 case studies were performed over a two and a half year period.[9] Students were recruited from a number of sources, summarized in Table 6. Recall that information from the pre-design phase indicated that the behavior of younger students more or less paralleled that of older students, making them excellent test subjects.

Table 6: Summary of Subjects
Students of the imaging curriculum were recruited from three major sources.

Children of fellow graduate students.....5
High School students working on independent projects.....9
Undergraduates.....1
Middle School students working after school.....3

[9]Three of the case studies, previously mentioned in section 2.2.2.1, took place in the pre-design phase, without the aid of the imaging curriculum.

The children of the graduate students were the youngest in the study, ranging in age from 9-12 years; the middle school students were in 6th and 7th grade (11-12 years of age). High School students came from an "alternative" public school which required the students to participate in "workshops" taught by members of the community. They were in 9th and 10th grade (14-15 years old). One undergraduate participated in a case study as part of a special project on learning programming languages.

6 of the subjects were female and twelve of the subjects were male. All students attended the local Boulder public school system; one student was a non-white, non-native English speaker; 17 were white middle class. Eight of the students had little or no experience working with computers; six were well versed at using applications, like games or word processors; four had some prior programming experience in BASIC, Pascal, or C. None of the students were previously proficient programmers.

All students participated in the project of their own free will. Students were given an introductory explanation of the program. If interest persisted, they were invited to an introductory session which demonstrated the SchemePaint application. At this time, students were also informed what would be expected of them including: (1) small pen and paper homework assignments, (2) attentiveness, (3) introspective reports on curriculum effectiveness and subject matter difficulty, (4) occasional written questionnaires.

In all of the case studies within this document the names of the participants have been altered.

3.4 Curriculum Outline

The curriculum was developed in a series of iterative steps. The method of iterative design was chosen so that information learned in early runs of the curriculum could be applied to later instantiations. A format similar to that used in the pre-design case studies[10] was used as a starting point with the following constraints:

- Properties 1-3 should be addressed in sequential order. This is consistent with the hierarchy of difficulty presented in section 2.2.3.1.

- A pre-test should be administered to gauge the student's background and interest. This stage of testing was intended to provide pointers to a student's interests, abilities and pre-conceptions, since it was determined in the pre-design phase that resilient misconceptions accounted for some of students' difficulty.

- Written questionnaires should be administered shortly after the introduction of properties 2 and 3 in order to monitor concept acquisition.

- Each property should be introduced in conjunction with all of the imagery tools. This is consistent with a theory of multi-modal imagery building. Literature in mental imagery (see also chapter 6) has supported the idea that exposure to a variety of representations results in a more complete mental model of a target concept (also [Polya 1945]).

Addressing properties 1-3 in order is roughly consistent with the standard order of introduction of material in a Scheme or LISP programming language course (see for example [Eisenberg 1990]; [Touretzky 1990]; [Koschmann 1990]; [Winston, 1989 #106]; [Wilensky 1986]). However, many of the other topics that would normally be intermittently addressed (control structures, recursion, etc.) were reserved until after a mastery of functional objects was achieved, or were introduced on demand according to student's interests.

[10]The specific course materials for these first few case studies was derived by a trial and error system.

As mentioned earlier, as far back as the first run of the curriculum the name `make-procedure` was used in place of the `lambda` special form (see [Soloway and Ehrlich 1984] for a study of novice programmers and the effects of non-mnemonic nomenclature). In earlier runs of the curriculum, students mentioned that `make-procedure` would have been easier if it was introduced earlier; additionally, there was some confusion over the difference in syntax for defining objects. Compare:

```
(define x 11)        (define x                    (define (x y)
                        (make-color-object           (* 5 y))
                           30000 0 20000))
```

In the latter example, the name of the object being defined is specified differently, using surrounding parenthesis. In order to eliminate this inconsistency and hence foster a uniform object model, the syntax of the procedure definition was changed in latter instantiations of the curriculum to:

```
(define x
   (make-procedure
      (y)
      (* 5 y)))[11]
```

My initial work with Maureen and Betty alerted me to the fact that some students might not be completely comfortable using the computer as a tool: intimidation, both from peers and the computer medium itself, proved a huge stumbling block for the girls. To counteract this effect, I was careful, on principle, to weave manipulative use into all stages of the curriculum. Manipulatives accompanied every introduction of a new topic in order to buffer insecurities about computational media and also to "put the student at ease" by providing links to the real world.

---

[11] There is in fact another difference which for complete consistency should have been altered. To be fair, the definition of all objects should have been specified as `make-<type>-object`. Although this is more uniform, it is misleading in the case of number objects. Note also that this system is a move towards introducing typing into the language. This issue will be discussed further in Chapter 7.

The curriculum was executed in multiple time intervals, some of which overlapped due to the time constraints of the students[12]. Therefore, at any one time I might have been working with one or more students at different stages of different iterations of the curriculum. After completing work with each student, details of the curriculum were re-evaluated and revised (again, in the spirit of iterative design).

I was able to learn new information about what sorts of things kids enjoyed programming and creating through the different iterations. I began with a small set of functions that I wanted students to compose (see the work of Hector from section 2.2.2.3) which I felt were graphically interesting and also represented each of the three properties of objects. As time went on, the students initiated their own interesting applications for the principles they were learning. I was able to create an informal library of functions that served the underlying curricular goals and were simultaneously interesting to students. This introduced some extra degrees of freedom into the curriculum so that students could choose what they were most interested in working on.

Each student worked with the curriculum for a total of between 10-20 hours. The duration of a session varied with the individual -- sessions ranged from 30-120 minutes. Hence, the number of sessions also varied with the individual--as did the pace of progression through the material. Students were encouraged to develop their own ideas resulting in variation of the specific problem set that each student worked on. The general format of the curriculum remained the same throughout the iterations. It was divided up into six basic stages:

---

[12]With the exception of the final iteration, which was run individually but simultaneously on 6 students.

0. Introduction, discussion & basic Scheme commands
I. Defining new functions (property 1)
II. Working with functional arguments (property 2)
III. Working with functional results (property 3)
IV. Interesting applications of property 3 (slope)
V. Recursion and higher order procedures

The stages 0-V are sequentially elaborated upon in Figure 8. The numbers in parenthesis are page numbers of this document which refer the reader to places in the case studies where a particular topic is addressed. The preceding letter is the first initial of the student.

3.5 Data Collection

Data was collected from the students in three ways:

1. Written questionnaires. Students were periodically required to complete questionnaires that tested their acquisition of certain concepts. Students were tested at least twice: once after working with properties 1 and 2 and again after working with property 3. Appendix B shows an actual questionnaire that was completed by a student.

2. Session transcripts. SchemePaint allows one to save a computerized transcript of a user's interaction with the interpreter. During each session, this transcript was saved. Additionally, written notes were taken on non-computational interactions[13]. At the end of a session, written notes were integrated into session transcripts.

3. Interviews. In audio taped interviews, students were directly asked to reflect on their impressions of the curriculum, what they had learned, and how useful the imaging tools were. This took place somewhere in the last third of the curriculum.

[13]Video and audio taping was determined to be too intrusive on a daily basis for this age range of students.

0. **Introduction, discussion and basic turtle commands**
- interested student interview
- written questionnaire
- discussion about the nature of objects (38-40)
- introduction to built-in procedure and number objects in Scheme
- introduction to name/object table
- introduction to Scheme interpreter rules (41-42)
- introduction to Basic turtle commands
- computational work with basic turtle commands (H:25-26)

I. **Defining new procedures** (B:60-61)
- animated review of rules using imagery tools
- computational work with defining new procedures
- hands-on "quiz" for explaining evaluation using imagery tools
- introduction to arguments
- hands-on work with imagery tools and arguments
- computational work with arguments

II. **Functions as arguments** (B:61-64)
- computational work with functional arguments
- hands-on work with imagery tools and functional arguments
- written questionnaire
- computational work with functional arguments
- introduction to color objects
- hands-on work with color objects and manipulatives
- computational work with color objects and manipulatives (B:66)
- [optional: introduction to lists, conditionals, and booleans in Scheme]

III. **Functions that return functions** (A:74-80)
- hands-on work with manipulatives
- computational work with functions as returned values
- written questionnaire (Appendix B)

IV. **Applications**
- slope function (A:83-84; Appendix D)

V. **Recursion and higher-order procedures**
- nested polygons

Figure 8: The Stages of the Curriculum
Levels I-III roughly correspond with Stoy's properties 1-3 of first-class objects. Each of these levels involves work with all types of representation (computational, visual, and physical). Levels IV-V are reserved for advanced topics in math and programming, respectively.

4 Case Studies

The following two sections present detailed case studies of subjects' experience with the imaging curriculum; the third section of this chapter contains noteworthy excerpts from the other 15 case studies; the last section summarizes the progress of all 18 participants. In the interest of brevity, and to avoid biased interpretation of "fuzzy" situations, only selections that overtly relate to issues of interest will be included.

4.1 Brooke

This is a report of work with the first student ever exposed to the imaging curriculum. During this first iteration, when the tools were fresh to both student and researcher, they were used sparsely. The subject, Brooke, was a 10 year old female in 5th grade. She had had no prior programming experience.

Brooke and I began the project with a discussion of computers and active and passive entities in the non-computational world. Her prior experience with computers was limited to word processing and games. She viewed the computer as both an active and passive entity. We went on to convince ourselves that everything in the world is an object, and that objects can be named and have different properties of an active and/or passive nature. I also explained the mathematical concept of a variable, with which she was previously unfamiliar.

Brooke retained this conception of objects in the world through to the next session, when we extended the notion to computational objects. I introduced her to number-objects and procedure-objects in SchemePaint, and explained how the "Scheme-

engine" (i.e. interpreter) maintained and utilized a huge table full of information about these objects. In order to explain the implementation details about how information gets deposited into and retrieved from the table, I used concrete procedure and number objects.

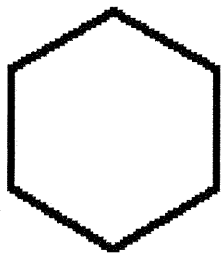
In the following session, Brooke was asked to explain the invocation of a procedure call. Although the concrete objects were not present for this latter session, she motioned throughout her explanation as if she were holding objects in her hands. She consistently spoke about number-objects and procedure-objects as "floating around".

In this second session, Brooke wrote her first procedure to draw a square. She suggested the concept of a parameter without instructor prodding. After an explanation, she wrote:

```
(define (square side)
   (repeat 4
      (fd side)
      (rt 90)))
```

During the third meeting Brooke wrote her second procedure to make a rectangle. She was able to recognize that it needed an argument but several of her attempts inserted an actual parameter in the procedure definition instead of a symbolic name. We went over the execution of such a procedure using the manipulatives. She noted and fixed her error and was able to then write the following procedure on her own:

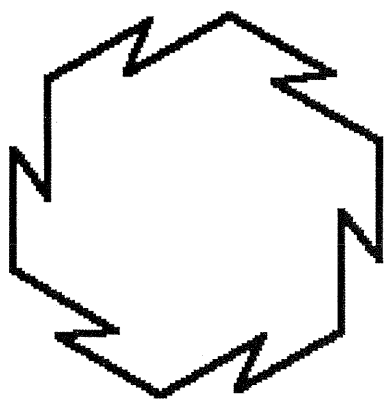
```
(define (hexagon side)
   (repeat 6
      (fd side)
      (rt 60)))
```



```
(hexagon 20)
```

Figure 9: Hexagon
The standard turtle graphics six-sided polygon.

During the fourth session, I proposed to Brooke that we might like to modify the sides of the hexagon to do things other than just go forward. I drew examples of six- sided figures that had varying shapes on each side. For example:



```
(hexagon zig)
```

Figure 10: Hexa-zig
In this polygon, the standard `hexagon` from Figure 9 has been permuted through the addition of a functional argument (in this case, `zig`) which specifies the movement of the turtle as it draws the sides of the shape.

I asked her to write a procedure that could achieve this or any number of other similar effects, keeping the side length constant. I told her that she could use any tools that she wanted to help her out. She chose to write it at the board. She worked out the problem almost entirely on her own. After five minutes of deliberating and

modifying, she had correctly completed the problem. Below is a transcript of her modification process, with student-teacher interaction included:

STEP 1:

```
(define (hexagon 20)
   (repeat 6
      (fd 20)
      (rt 60)))
```

[B: "This isn't right. I'm just thinking"]

STEP 2: [working on line 3 above for STEP 2-4]

```
(? 20)
```

STEP 3:

```
(fd pro 20)
```

STEP 4:

```
(pro 20)
```

[At this point I informed her that she was doing well. She seemed stuck for a few minutes, so I also told her she could ask for a hint. She did so, and I told her to think about how the computer would know what "pro" was.]

STEP 5: [line 1]

```
(define (hexagon pro 20)
```

[I told her she was close, and asked her how she would use the procedure. In trying to come up with a correct invocation, she realized her mistake and completed the correct version.]

FINAL:

```
(define (hexagon pro)
   (repeat 6
      (pro 20)
      (rt 60)))
```

She was then able to explain and use the procedure correctly. So, after 3-4 hours of instruction on SchemePaint, Brooke was able to successfully and independently make use of a procedure as an argument.

In the following session, Brooke was asked to explain the implementation details of all the procedures she had written. She was given the option to use the manipulatives

to explain the invocations. She did not opt to do so, but rather motioned again as if she were holding objects as she explained the actions of the interpreter. At one point, she even used a marker pressed up against the white board to denote objects floating around in 3-space. Brooke's explanations were correct to a sufficient level of detail. When asked to repeat the explanation using the concrete object, it was again correct. Later in the same session, Brooke was asked to write a new procedure. She walked to the board, drew a large rectangle, labeled it "procedure object" and began filling in the details.

At this point Brooke's mental model of procedures seemed to support a high-level object concept. This became clear through observing her use of the concretizations. When Brooke was asked to write a new procedure she did not just write the Scheme code: she began by drawing a box on the board and labeling it a procedure-object, then inserting the code in the box. Further, she motioned with her hands as if the concrete objects were present. Brooke chose to reiterate at least two different imaging modes for explaining the same concept. At no time did she choose to use the computer to work out a problem.

After only five 1-hour Scheme sessions, Brooke appeared to have developed some object concept of data. To test her skills at applying this knowledge, Brooke filled out a written questionnaire. There were several interesting results.

1.) Brooke was still confused about symbolic names for objects. Given the following expression, `(define side 90)`, she explained to me that this expression redefined the value of `side` "like in the hexagon procedure". In the curriculum, young students are first introduced to the notion of defining a procedure. They are shown that number objects also can be given names but they do not have cause to use this principle. Thus, when presented with the above item they exhibit two

misconceptions. First, students believe that because the variable is called `side` (this is the name they have most commonly seen given to an argument) it has something to do with a procedure. They do not understand the purely symbolic nature of names. Secondly, students are not able to extend the property of naming from procedure to number objects, again, indicating that the object types retain unequal status. These misconceptions were exhibited by the pre-design students as well.

2.) Prior to the test, Brooke had written two hexagon procedures, one with a numerical argument and one with a procedural argument. When Brooke was asked to combine these two procedures (the first time she would have ever seen a procedure with two arguments), she reverted to the same error we saw in her original "struggle" with a functional argument to a function. Her answer to the test question follows:

```
(define (hexagon-3 side side-shaper)
   (repeat 6
      (fd side side-shaper)
      (rt 60)))
```

When we went over the test together, I suggested she try typing it in to see if it worked. When it did not, she changed the third line to: `(side side-shaper)`. Interestingly, this implies exactly the opposite of her original answer. Here, the strong similarity between objects overrides her knowledge of syntax and leads her to believe that function and argument order are irrelevant.

An inconsistency was flushed out by Brooke on her questionnaire. It regards the disparity in syntax between defining a procedure-object and a number-object. Asked to define a particular number object, `11`, to have the name `age` Brooke wrote the following Scheme code:

```
(define (age)
   (11))
```

This is the correct syntax for the definition of a procedure-object. Brooke notes that the difference in syntax is confusing. I agreed: in a world where objects are supposed to have equal status, they should be uniformly definable. Again, students seem to be very sensitive to syntactic inconsistencies, pointing to the utility of an iterative curriculum design process.

In the second half of the test review session (#6) I showed Brooke the results of a procedure called `mystery-1`, in similar vein to the procedure I used with Hector in the pre-design phase. This `mystery-1` took a single number argument and created triangles of that size. After three examples of use, she was able to write the following:

```
(define (triangle side)
   (repeat 3
      (fd side)
      (rt 120)))
```

I then demonstrated the output of a `mystery-2` procedure with three examples: `(mystery-2 fd)`, `(mystery-2 zig)`, and `(mystery-2 hexagon)`. She requested at this point that I stop so that she could write the following:

```
(define (triangle side-shaper)
   (repeat 3
      (side-shaper 20)
      (rt 120)))
```

Note that this is somewhat more advanced than the mystery procedure solved by Hector[14] in the pre-design phase.

During session #7 we began working with color objects. Brooke was very unhappy with the name of the standard SchemePaint `set-default-color` procedure. We renamed it to `set-current-color`. She accepted without question the notion that we

[14]Recall: `(define (mystery-1 f) (f 90))`

could merely rename an object: `(define set-current-color set-default-color)`.

After a brief introduction to the `make-color-object` and `set-current-color` commands, Brooke was required to write a procedure that makes purple circles (purple was the test color-object that I used to explain `make-color-object`). After quickly and successfully completing this task she decided to define her own color object, `aqua`. In reply, I asked her to make a procedure which made `aqua` circles:

```
(define (aquacircle size)
   (set-default-color aqua)
   (repeat  18
      (fd size)
      (rt 20)))
```

At this point, after writing two procedures that differed only in one word -- i.e., the name of the color object -- I asked her to write a procedure that could produce hexagons of *any* color. Her first reaction showed some pattern tendencies:

```
(define (hexagon color side)
   (repeat 6
      (fd color side)
      (rt 60)))
```

We broke here for the day, but resumed session #8 with a review of color objects and another attempt at this procedure. This time, I used the manipulatives to explain the difference between procedures whose result is interesting (`make-color-object`) and procedures whose side-effect is interesting (`set-current-color`). I then asked her again to write the colored hexagon procedure. She assumed her favorite position of creative reflection at the white board. She asked if the procedure needed to make hexagons of any size. I said no. She started writing, commenting that it was easier this time because "you

made me make them any size last time." She was half finished with the procedure when she paused for a second, said "oh...", and added the size argument anyway. It was as if she just needed the problem broken down for the slightest time quantum before she realized the additional size argument was an easy extension. Note that she did not add a third argument, e.g. `side-shaper`, on her own. The first revision looked like this:

```
(define (hexagon color side)
   (color)
   (repeat 6
      (fd side)
      (rt 60)))
```

There is an interesting observation here. It is as if, for her, objects should embody their executability. Hence, she has taken one of the properties of procedure objects and extended it to color objects: that is, putting `color` in parenthesis should affect a change in the pen color in the Scheme environment. We went through an invocation of her new procedure using the concrete manipulatives. At this point, she said "Oh, I need to do something more with the color." and she added the correct procedure invocation. During the following session, she was asked to explain three statements as a review:

```
(define (hexagon color-o)
   (set-current-color color-o)
   (repeat 6
      (fd 10)
      (rt 60))))

(define purple (make-color-object 60000 0 60000))

(hexagon purple)
```

First, she drew a 2-D color object for purple and put it in the name-object table. Then she performed a similar operation for hexagon. To describe the invocation, she first

got a physical manipulative for a color object and said "Scheme goes out and grabs the color object for purple." She waved it around a bit and set it on the dry erase sill near the 2-D drawing. She did the same for the `hexagon` procedure object, then said "Once it has these two things it can do it." She grabbed both objects and pressed them together.

During the next three sessions we took a break. We resumed with a task that involved combining Brooke's knowledge of color objects with some new knowledge about procedures which generate procedures as their result (property 3). I found teaching this to be far more difficult than any of the previous tasks. The first observation I made was that this task required a good deal more "forward reasoning" than the others we had encountered so far. For example, the following was the first procedure that Brooke was assigned to write:

```
(define (color-version-maker col proc)
   (make-procedure
      (size)
      (set-current-color col)
      (proc size)))
```

It is hard for a "novice abstracter" to comprehend this procedure. First, she must look ahead to the time when the *result* of running color-version-maker *is itself run* in order to "fill-in" a value for size. Understanding that `size` has a meaning that is two levels of abstraction removed from the procedure in which it first appears requires a certain "cognitive patience" since the time at which the value of size is determined occurs after the time at which it first appears as an unknown to the user.

Brooke and I wrote this procedure together, using the manipulatives as we developed the code[15]. Brooke seemed to understand the procedure as we wrote it, and likewise

[15]From a teacher's perspective, I found that the manipulatives facilitated explanation. Whether this perceived ease of instructing is accompanied by an increased ease in understanding on the part of the students is an topic which will be returned to later.

she did not seem confused that a call to the procedure would return a procedure object as its result. However, she demonstrated no real ability to generate an example of use on her own. Even when she finally came up with the following call to the procedure:

```
(define cygon (color-version-maker cyan hexagon))
```

she still tried to use it incorrectly:

```
(cygon)
```

I asked her to explain the procedure call just like she had in the past for less complicated procedure calls, like `(hexagon 20)`. At first, she got it wrong, tending to want to break down the `hexagon` procedure right away and begin executing its commands instead of viewing it as a unit. After coming to an understanding about treating procedures as units, she generated the following code on her own:

```
(map cygon '(10 20 30 40))
```

```
(map (color-version-maker blue square) '(10 20 30 40))
```

Although I was impressed with her initial comprehension in these tasks, Brooke left feeling uncomfortable and insecure with her performance. Hence, we spent the next two weeks repeatedly going over this procedure and examples. She was asked to write the `color-version-maker` procedure over again at the beginning of each session without my help. Two interesting observations: (1) whenever she needed to write a procedure, either `color-version-maker` or the procedure that results from executing it, she always started by drawing a large rectangle on the board and labeling it with the procedure object structure, and (2) when she went to fill in that structure, she always started with specific code and then changed it to more general symbolic code, i.e. `(set-current-color cyan)` in her working version would change to `(set-current-color color-object)` in the final version. She was also asked to

explain a call to `set-current-color` using the Scheme evaluation rules. She twice attempted to first do so without the objects and failed. She followed each of those failures with successful attempts using the concrete manipulatives.

During the last session Brooke was asked to write a property 3 procedure on her own. Her only problem was one of syntax -- she still had problems with the `make-procedure` syntax in the absence of any visualization tools. With syntactic details aside, however, she was able to correctly generate the code:

```
(define (shape-maker sides)
   (make-procedure
      (size)
      (repeat sides
         (fd size)
         (rt (/ 360 sides)))))
```

I asked her to explain: `(define pentagon (shape-maker 5))`. She was able to explain the command correctly but, interestingly, she was bothered by the name `pentagon`. She thought that the procedure should somehow reflect that it was born from `shape-maker` and therefore she called it `pentasm`.

4.1.1 Summary

Brooke was the first student to use a variety of imaging tools in the curriculum. She made ample use of both visual and physical representations in problem solving, shying away from the computer until a time when she was satisfied with the correctness of her work. She was able to work through problems at level III aided by lengthy interaction with manipulatives. In a disappointing end to this case study, Brooke lost her final questionnaire and was reluctant to complete a second copy.

4.2 Aaron

Aaron, the subject of the second case study to be described here, participated in the final execution of the curriculum. Aaron was a 10th grade male with some prior programming experience in BASIC. He did not have a good enough working knowledge of the BASIC language to program on command. Aaron participated in the project through an apprenticeship program at his school. Aaron worked with the imaging curriculum for seven two hour sessions.

During our discussion of objects in the world and active and passive entities, Aaron made the following observations:

> "Everything is an object except thoughts."
> "All that thinking about thinking tells me is that I am actively thinking about an active thing."

Aaron's assertions would seem to indicate that he has a fairly strong bias against stereotypically active things having any sort of object-like nature.

After this discussion Aaron was introduced to the basic turtle commands and given an explanation of the Scheme rules for evaluation using manipulatives; he was then given freedom to explore with the computer. Aaron began by first writing the following code:

```
(repeat 360
   (fd 1)
   (rt 1)
```

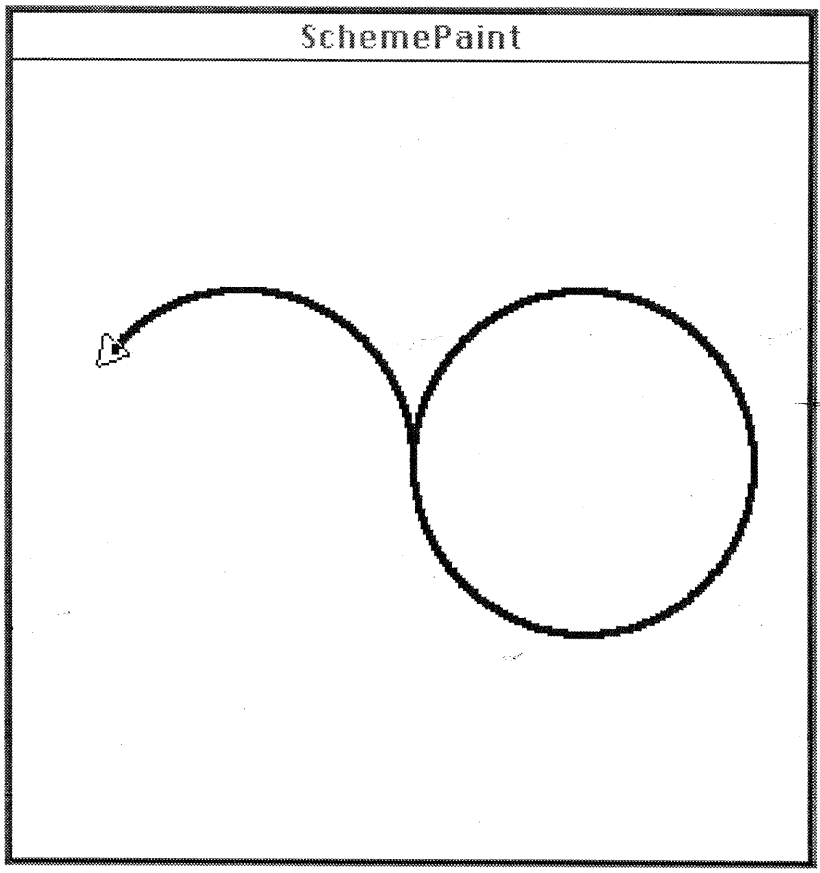
He was continually executing this segment of code and rotating the turtle 180 degrees to make two "eye-like" tangent circles. I explained to him how to make this code into a procedure (note the early introduction of `make-procedure` in this iteration of the curriculum):

```
(define circle
   (make-procedure
      ()
      (repeat 360
         (fd 1)
         (rt 1))))
```

Shortly after this introduction, he asked how to add an argument to change the size of the circle. Armed only with that information, I left him to work for five minutes. When I returned, he had written the following code:

```
(define circle
   (make-procedure
      (size dir)
      (repeat 360
         (fd size)
         (dir 1))))
```

He was using this code to generate either side of his "eyes":




```
(circle 1 rt)
(circle 1 lt)
```

Figure 11: Eyes
A favorite introductory exercise had students creating circle procedures. In this case, the student decides on his own to vary the direction of turn by adding a functional argument.

After this, I asked him to write the most general procedure for making polygons that he could think of. He wrote:

```
(define polygon
  (make-procedure
     (sides direction turn incA incB)
     (repeat sides
        (direction incA)
        (turn incB))))
```

However, he informed me after writing this that the following version was more readable:

```
(define polygon
   (make-procedure
      (A B C D E)
      (repeat A
         (B C)
         (D E))))
```

Clearly, Aaron had a very uniform concept of data objects. At this point, Aaron had only limited exposure to the imaging curriculum. One might suppose that Aaron already possessed a predisposition towards an object ontology of data; as we shall see later, Aaron, in an interview, admits to using his own visualization skills to help him program even prior to this time. Note that Aaron wrote the above code after 1.5 hours of working with SchemePaint: this fact coupled with his self-admitted tendency towards visualization supports the assertion that imaging techniques are one method of successful concept acquisition for functional data.

During the second session, after a review of command evaluation using visual and motor imaging manipulatives, I explained to him that we wanted to be able to make polygons which had polygons as sides. For example, contrast a five-sided polygon

with straight lines (`fd`) for sides with a five-sided polygon with ten-sided polygons for sides:

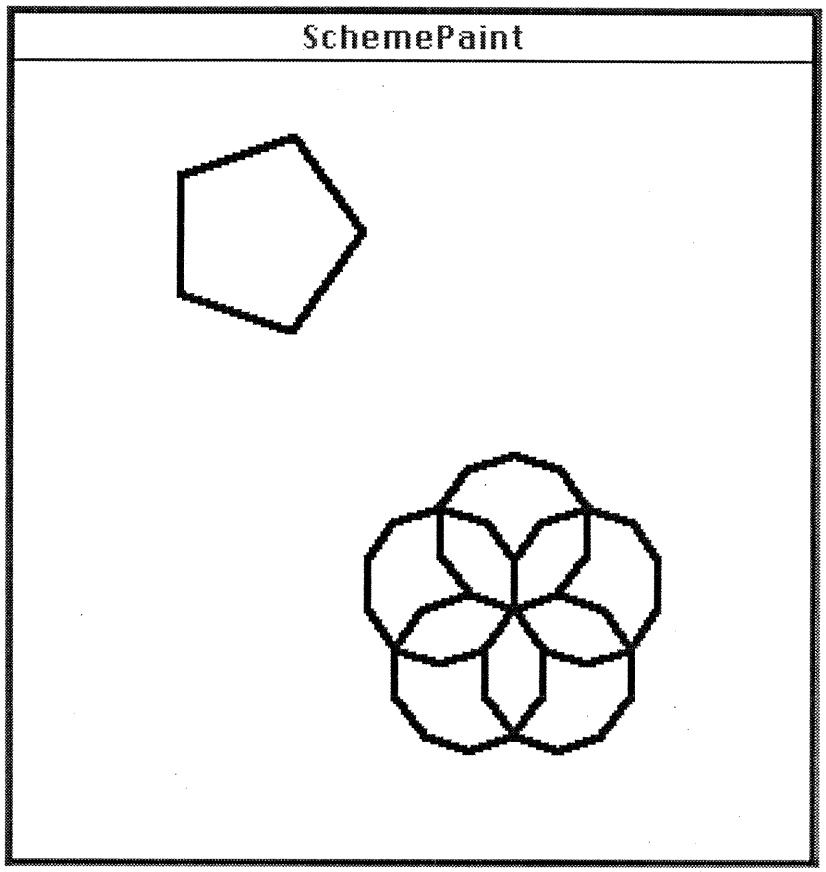


Figure 12: 5-Sided Polygons
The top polygon shows a five-sided figure with straight lines for sides. Below that is a five-sided polygon with ten-sided polygons for sides. These drawings are both accomplished using the same procedure with varying functional arguments.

As a first attempt at a solution, Aaron wrote the following:

```
(polygon 5 polygon 10)
```

This solution ignores the syntax problem presented by passing the `polygon` argument into the `polygon` procedure; on the other hand, it reflects an instinctive notion that procedures are just objects to be manipulated. Innocently, it correctly portrays a tension that can be seen between the object model and the standard environment model [Eisenberg 1990]; chapter 7 will further elaborate on this topic. We went through this expression, using the manipulatives to realize why it would not work

(most simply, the `polygon` procedure, which takes four arguments, is substituted into a slot where the expected procedure takes one argument).

I then explained how it could be useful to write a `polygon-maker-procedure` (abbreviated `ppm`). I told him that when you typed `(ppm 4)` then it should give you back a square procedure. Aaron seemed to have a bit of trouble grasping the fact that it would not actually *make* the square. Regardless, he was set to the task of writing the `ppm` procedure with just this limited information. The following code is taken from an actual transcript of Aaron's work on this problem. Instructor interjections are noted in brackets:

```
(define polygon-maker
   (make-procedure
      (sides, size)
      (repeat sides
         (fd size)
         (rt angle)
         (angle = / 360 sides))))
```

[I explained the concept of nesting procedure calls using mathematical examples]

```
(define polygon-maker
   (make-procedure
      (sides size)
      (repeat sides
         (fd size)
         (rt (/ 360 sides)))))

>>> (polygon-maker 4 5)
```

[I explained the difference between result objects and side-effects]

```
(define square
   (make-procedure
      (size)
      (repeat 4
         (fd size)
         (rt 90))))
```

[I explained the difference between `(ppm 4)` and `((ppm 4) 10)` and `(define square (ppm 4))`; in other words, I vaguely treated the subject of anonymous objects through example. This was the final intervention by the instructor.]

```
(define PM                                    [A]
   (make-procedure
    (procedure)))
```

```
(define PM                                    [A]
   (shape)
   (shape))

(define PPM                                   [B]
   (make-procedure
    (side)
    (define 4
      (make-procedure
       ()
       (fd 10)
       (rt 90)))))

(define PPM                                   [B]
   (make-procedure
    (side)
    (rt side)
    (define square
      (make-procedure
       ()
       (fd 10)
       (rt 90)))))

(make-procedure                               [A]
     (x)
     (+ x 1))

(define ppm                                   [B]
   (make-procedure
    (sides)
    (define sides
      (make-procedure
       (sides)
       (rt (/ 360 sides))))))

(define ppm                                   [B]
   (make-procedure
    (sides length)
    (define sides
      (make-procedure
       (sides)
       (rt (/ 360 sides))))))

(define ppm                                   [C]
   (make-procedure
    (NumSides SideLen)
    (repeat NumSides
            (rt (/ 360 NumSides))
            (fd SideLen))))

(define polygon                               [C]
  (make-procedure
   (sides direction turn incA incB)
   (repeat sides
           (direction incA)
```

In the latter part of the examples above, Aaron seems to remind himself of the use of `make-procedure` by practicing with a few trivial and non-related examples [A]. He appears to conceive that he needs to use a nested `make-procedure` call, but he still cannot divorce this idea from the notion that procedures need names [B]. Unsuccessful, he begins to abandon the idea of nesting [C].

At this point, we began to work through the problem together. We drew a step-by-step diagram on the board (Figure 13).

This translated into an instant solution for Aaron:

```
(define ppm
   (make-procedure
      (NumSides)
      (make-procedure
         (SideLen)
         (repeat NumSides)
            (rt (/ 360 NumSides))
            (fd  SideLen))))
```

After writing the correct procedure, I asked him to use it. It took him three expressions (shown below) to work out the correct syntax for what he wanted to do. Note that his first instinct was to include an extra numerical argument alongside the call to `ppm` (in actuality that numerical argument was a specification for the `polygon` procedure).

```
>>> (polygon 6 fd ((ppm 4) 10) rt 60)
>>> ((ppm 4) 10)
>>> (polygon 6 (ppm 4) 20 rt)
```

For homework, Aaron successfully completed an `inc-maker` function (takes a single numerical argument, `n`, and returns a function which increments its numerical argument, `x`, by `n`).

During the third session I asked him to write me a new square procedure that would work for the following invocation: `(square (ppm 6))`.

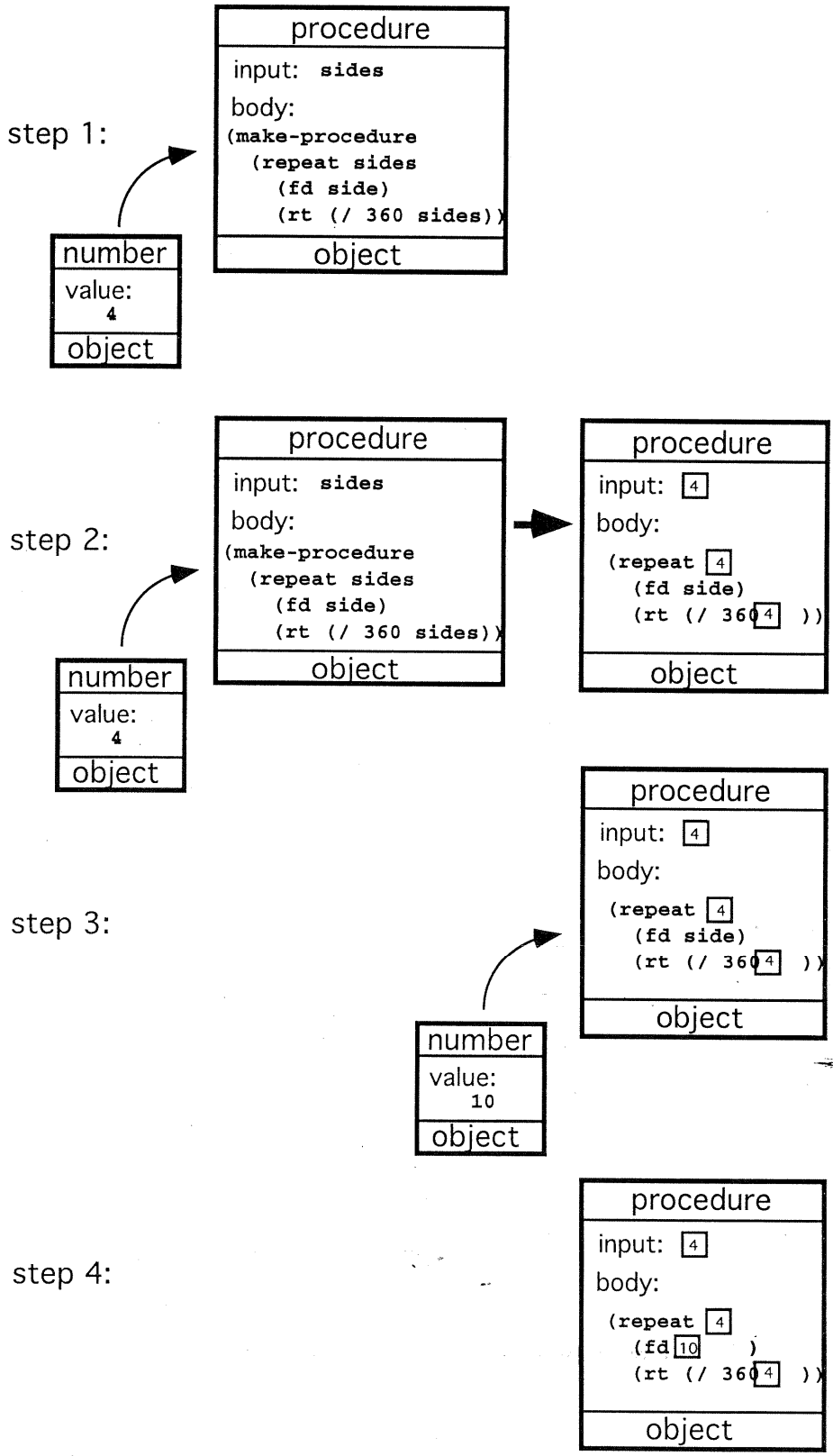


Figure 13: Visual of `polygon-procedure-maker`
An example of how manipulatives can be used to work out expressions: in this diagram, the student is able to verify the sanity of anonymous objects and "still missing" variables.

At first, he wrote a "normal" square procedure using the line `(fd size)`. Then, when he tried to run the original command I had asked him to mimic how he got an error. I had him work out the entire invocation of the command using the evaluation rules and creating new procedure objects with the manipulatives as he proceeded. When he got to the stage of invoking `(ppm 6)`, he did not think it would work because the computer would not know what `SideLen` was (missing variable phenomenon):

```
(define ppm
   (make-procedure
      (NumSides)
      (make-procedure
         (SideLen)
         (repeat NumSides
            (rt (/ 360 NumSides))
            (fd SideLen)))))
```

While Aaron realized that the missing variable explanation was not in line with the original error message that he had gotten, he said that he could come up with no other explanation. I encouraged him to plod along in the rule execution. He got to the point where he had to substitute in the anonymous `(ppm 6)` procedure object that he had created in place of the variable `size` in `(fd size)`. He realized his error at that point and fixed it. We went back over it a second time and I asked Aaron if he now understood why we didn't need to know about `SideLen`; he did. I also directly asked whether he had a problem with the idea of anonymity for the result of a call like `(ppm 6)`; he claimed that he did not.

While observing the above sequence it became obvious to me how critical the process of detailed rule evaluation using manipulatives had been to his debugging process. In this case, the objects served more usefully as "tools of rehearsal" rather than "instigators of intuition." So far, the manipulatives had most apparently been useful

at a very cognitively high level -- to foster an overall intuition about the object nature of function. Now, the concrete tools were being utilized beyond the student's initial flash of insight -- on a more long term basis as objects to think with. In either scenario, however, it is noteworthy that the tools played a critical role in ushering the student into a correct problem solution.

By this time, it was clear that Aaron was capable of and interested in learning more about the Scheme language than is typically presented in the standard curriculum. Accordingly, we went over lists and list manipulation procedures, predicates, conditionals, and Boolean objects. One of the tasks he was initially asked to do was to redefine `car` and `cdr` to `first` and `rest` (recall that half of the undergraduates previously surveyed did not complete this question in the most concise manner). Aaron redefined `car` and `cdr` in the simple object-oriented way:

```
(define first car)
(define rest cdr)
```

Here, we also began working with color objects. The anonymous object concept was completely clear to Aaron; one could surmise that this was due to his previous introduction to anonymous procedure objects. Aaron's task, similar to that of Hector (from the pre-design phase), was to write a series of color transformation procedures. He had no difficulty with this; he even, on his own, came up with the idea of redefining the "-" procedure to the name "`lighten`". For example, Aaron wrote the following color change function:

```
(define change-r
   (make-procedure
      (dir color)
      (make-color-object
         (dir (get-red color) 5000)
         (get-green color)
         (get-blue color))))
```

Typically, one would call `change-red` with the - and + signs as arguments: `(change-r - cyan)`. However, Aaron's change makes the code nicely readable: `(change-r lighten cyan)`. His next task was to write the `compose` function[16], again as in the pre-design phase. His first attempt was not too conceptually far off. Note, however, that it lacks the notion of an argument to operate on:

```
(define compose
   (make-procedure
      (function1)
      (make-procedure
         (function2)
         (function1 (function2)))))
```

For homework, he revised the function into a technically sound version:

```
(define compose
   (make-procedure
      (function1 function2)
      (make-procedure
         (color)
         (function2
         (function1 color)))))
```

During the following session, Aaron took a written questionnaire. This questionnaire contained never before seen problems. It was designed to test whether Aaron could take the concepts that he learned in a graphics context (SchemePaint) and transfer them over to a more mathematical pure Scheme environment. Aaron's complete test is shown in Appendix B. To summarize, Aaron retained a good concept of functions as arguments into the mathematical domain and is able to successfully apply what he has learned. Note also that he successfully answered the problematic `(apply-to-5 create-subtracter)` example.

Immediately after the written questionnaire, Aaron participated in an interview (Appendix C). In this interview, Aaron was directly asked to self-assess how he

[16]The idea of composition of functions was completely new to Aaron and required extensive explanation. While that was not surprising, it was also apparent that he had little or no mathematical experience with any sort of function manipulation, e.g. graphing or f(x) notation.

thought the imaging curriculum had affected him. Aaron does not espouse the usefulness of the physical manipulatives, but rather prefers the visual representation:

> *"I think having it on the board was the most useful because... I think putting it inside of the little pouches and stuff its harder for me to see how it goes together and how it works."*

Still, Aaron's work with physical manipulatives has left a mark on him by coloring both his language and his gestures. He refers to a functional argument to a function as "a box inside of a box". Throughout the interview -- even as he denies the usefulness of the objects -- Aaron simultaneously motions with his hands to make a box shape as he talks about functions.

For the following two weeks we worked on integrating the concepts of recursion, graphics, and higher-order procedures. Aaron's success with these ideas was uninterestingly precise.

During the last session Aaron wrote a function to find the slope of mathematical functions. He used a worksheet to guide him through the process (Appendix D). Unfortunately, the worksheet turned out to be insufficient because he did not have proper background in mathematics (for example, he did not even know how to graph the function y=2x). He was unfamiliar with f(x) notation. We took a 20 minute break from the worksheet so that I might present a brief explanation of slopes and functions in mathematics.

While he seemed to understand how to calculate a slope, he was still too uncomfortable to tackle question 4 of the worksheet which asks that he write the `slope` function in Scheme. As a sub-problem, I asked that he write a slope function for only the `double` procedure (as opposed to one which calculated slopes for any function). He wrote:

```
(define slope
   (make-procedure
      (function)
      (- (function 2) (function 1))))
```

I clarified that I wanted it to work for *any* interval, but only the `double` function. He wrote:

```
(define double-slope
   (make-procedure
      (x)
      (- (double (+ x 1)) (double x))))
```

Then I asked that he abstract it to work for any function. He did so, with just a minor glitch:

```
(define slope
   (make-procedure
   (function)
   (define
```

[A five second pause: "Hmpf..." and then he changes `define` line:]

```
  (make-procedure
     (x)
     (- (function (+ 1 x)) (function x)))))
```

At first, he incorrectly used the function:

```
(slope (double 3))
```

but quickly fixed his mistake:

```
((slope double) 3)
```

He was then able to complete the worksheet and make observations about the behavior of functions. We concluded with a discussion of how slopes relate velocity and acceleration. Aaron was thoroughly proud of himself when he realized what he had just accomplished.

4.2.1 Summary

Aaron was able to rapidly progress through the curriculum, mastering properties 1-3 and some advanced topics in Scheme programming. His use of the physical manipulatives was sparse since he preferred the visual representation; he claimed to have an object concept of function prior to beginning the curriculum. Aaron was able to apply his knowledge of functions within the domain of graphics to a purely mathematical problem involving slope functions (property 3).

4.3 Other Interesting Case Pieces

4.3.1 Gretta

Gretta was a 14 year old high school student with little interest or background in mathematics. She chose to do the SchemePaint project to fulfill a workshop requirement at school and because her father wanted her to learn more about computers. She was quiet and introverted. We met for eight weeks in two half hour sessions per week. Gretta participated in the final iteration of the curriculum.

During the fourth session with Gretta we went over adding arguments to procedures. She had been working on generating circles since the beginning of our work together. Gretta quickly grasped the idea of an argument, and added one to change the size of her circle. Then, she summoned me and asked if it was "ok" to replace the `right` procedure with an input so that she could make the circle turn right or left as it was drawing.

Later in the curriculum when we discussed property 3, Gretta exhibited the same instincts about property 3 as Aaron did. She was asked to write a procedure called `make-colored-hexagon-procedure` (`mchp`) which took a color object as input and

generated a "hexagon making procedure" which could then be used to generate hexagons of that color of any size. Her first attempt was just shy of correct:

```
(define mchp
   (make-procedure
   (col)
   (define col-hex
      (make-procedure
      (size)
      (repeat 6
         (fd size)
         (rt 60))))
```

She made her first mistake in forgetting to actually call the `set-default-color` procedure. More interestingly, however, she tried to define and assign a name to the generated procedure, indicating that she had a problem with its anonymity (recall that Aaron exhibited the same tendency). After we fixed both problems, a run through with the manipulatives allowed her to reconcile the validity of an anonymous object.

At the end of nine weeks Gretta was interviewed about her opinions regarding the work we had done. Some interesting excerpts follow:

I: Does it seem any different to use procedure objects as inputs than it does to use number objects as inputs?
G: *Kind of the same but there is a bit of a difference. Both can be variable.*

I: How vivid is the image of a procedure? Do you have an image?
G: *Kind of since I have been shown a table or a square object. It makes it easier to think about it.*

I: Do you see procedures as single units? Do they have any parts?
G: *Single units unless they've got input. Then there is another part you need to stick in to make it work.*

I: Sometimes we represent objects as pictures on the board in the framework of the name/object table. Sometimes we represent them with the puffy objects. Do you think either of these representations has helped you to learn about objects in SchemePaint?
G: *Yeah. It helps to show what's going on so that we don't always think of it as a big mass of numbers. It helps you understand what's going on so that you don't just think of the computer as this big massive thing that shoots something out at you.*

I: So, I'll make a statement based on what you just said and you tell me if you think I've gotten the idea that you were talking about: "Objects help build bridges between thinking in the everyday world and thinking in the computer world." Is that right?
G: *Yeah.*

I: Is the computer world real?
G: *Yeah. Its different from ours but its real because its still happening... still going on.*

I: Its real, so, is it more of an abstract thing or a physical thing?
G: *Its more like abstract things than physical things.*

I: So, do the objects help make the abstract more understandable?
G: *Yah. Humans make minds think they should see everything concretely so it helps when you can see it concretely.*

I: Is one representation better than the other? Are there times when you mentally refer to one representation or the other, depending on the circumstance?
G: *[Writing on the board] shows the table better. Both show what's happening, just in a different way.*

I: What do you think the objects are good at showing?
G: *How it goes and gets them, has to look them up and put them together. It shows it in a more solid way.*

4.3.1.1 Summary

Gretta had no prior programming experience but was able to produce the idea of property 2 all on her own. Although she progressed to property 3, she was able to perform such calculations only with the aid of the physical manipulatives. In an interview, she supports the utility of the manipulatives for bridging abstract and concrete thinking.

4.3.2 Nate

Nate was a 13 year old high school student. He had programmed extensively before in BASIC and was currently learning C. He was a difficult student with a short attention span. His programming was careless and not well thought out.

About half way through the curriculum (after functional arguments had been introduced and mastered) I began to introduce color object with the physical manipulatives. Nate's response was, "Oh, not the puffy objects. I hate the puffy objects. Put them away and I will image that they are there." The first time he went to the computer to explore some programming with color objects, he typed in:

```
(make-color-object 50000 20000 30000)
```

He immediately paused and said "Oh, but that just makes a lost object that we can't access. We have to use `define` to name it." Later on, when he learned about list objects he thought that there should be such a thing as a list generating procedure. This procedure, he thought should take as input some kind of specification for the list you would want to generate and then create it. You could then use a call to this procedure as test input to other list manipulating procedures. This idea shows a solid object concept of data. He later demonstrated this further by writing a procedure to generate procedures that returned the nth element in a list. With only that introduction, he was able to write the following on his own[17]:

```
(define make-nth-elt-getter
   (make-procedure
      (input)
      (make-procedure
         (input2)
         (list-ref input2 (- input 1)))))
```

In the following passage, Nate comments about his concept of data and the utility of the curriculum for developing those intuitions:

I: Are there any kind of pictures in your head around that thing, any kind of visual image of the idea of passing a procedure -

[17]There was one glitch; he too initially added a `define` line in the second call to `make-procedure`. However, he realized and repaired his error before he was even done with the procedure.

N: *I mean I guess there's the idea of the little thingy and you take the head off and you take another little thingy out and you take the head off and you take another little thingy out...*

I: Wait, what's a little thingy?
N: *You know like those little clay people inside the people and the last one is like a little wad of clay?*

I: Now, in this particular procedure does it seem any different to you, the idea that we would pass in a number as a piece of data to this procedure or that we would pass in a procedure? Or do they seem pretty parallel?
N: *No, it seems slightly different cause when you've got a procedure you've got something that still is doing something whereas if you have a number you're having something tell something to do something. Its not really doing something on its own.*

I: The number is not doing anything on its own?
N: *Right. Its just telling things what to do.*

I: Ok, so the number is telling things what to do?
N: *Yah. The number is telling the procedures what to do.*

I: We've done a lot of stuff talking about passing procedures as arguments and returning procedures as results of other functions. I want you to describe sort of generally any imagery that you have around procedures. So when you think "I have to write a procedure" or "I have to pass this procedure as an argument" what image do you have in your head of a procedure? Are procedures units? Do they have parts? What do they look like?
N: *Yah, I mean to me it seems like a procedure is a sort of a unit and a block of data.*

I: Along the course of our 10 weeks of work, I've presented two different pictures of things. My question for you is opinion based: do you think that either of these representations has helped you to learn in any way? And if the two different representations have helped in different ways, how?
N: *I'm not sure that either has helped a lot but I think that having a little object definitely helps you think of something as an object.*

I: Are there times when you're sitting down to write a program when you would conjure up either of these representations like you would see the table in your head or -
N: *- No, I never lean back and close my eyes and see little pink things floating in my head.*

I: Does returning procedures as results of other procedures seem different or weird to you or does it seem pretty natural?
N: *It seems pretty natural.*

I: So here, the fact that I have a call to increment-maker with an argument of 10, and what do I get back when I just do that?
N: *Junk.*

I: Junk?
N: *Basically, a procedure that's nameless and so its junk 'cause you can't access it.*

I: Does that seem like a weird thing?

N: *Seems weird that you can't go and fry your nameless procedures. Like, get rid of them.*

I: So it doesn't seem weird to you that procedures can be returned as results of other procedures?
N: *It's different, but it makes sense.*

4.3.2.1 Summary

Nate, who had some programming experience prior to exposure to the imaging curriculum, expressed a particularly strong dislike for the physical manipulatives. Despite this, his discourse about and around functions advances a very strong object concept of function. Nate had the best working knowledge of anonymous objects of any of the students of the curriculum. Like Aaron, he was able to apply his knowledge across domains to purely mathematical problems.

4.4 Summary of Results

A total of 18 students began the imaging curriculum. Each student attained a different level of achievement within the program. Table 7 summarizes the progress of each student: AGE in years; TIME in the curriculum in hours, number of COMPANIONS with whom the student was simultaneously tutored; overall ATTITUDE towards the experience; and the LEVEL of achievement completed, based on the following division:

0 Basic scheme commands
I Defining new functions (property 1)
II Working with functional arguments (property 2)
III Working with functional results (property 3)
IV Interesting applications of property 3 (slope)
V Recursion and higher order procedures

Table 7: Student Achievement Summary
Student's performance in the imaging curriculum is recapped. The level that each student reached is considered alongside of information about age, duration, working partners, and overall attitude.

ALIAS	AGE	LEVEL	TIME	COMPANIONS	ATTITUDE
maureen	12	I	7	1	insecure
betty	12	I	7	1	insecure
hector	11	III	12	0	enthusiastic
brooke	10	III	14	0	enthusiastic
lydia	21	III	9	0	enthusiastic
sonny	18	O	3	0	neglectful
laurence	9	I	10	1	energetic
jonathan	11	I	10	1	energetic
patty	13	I	5	2	hostile
david	13	0	5	2	hostile
joseph	13	I	5	2	hostile
nate	15	IV	21	1(->0)	arrogant
donald	14	I	3	1	enthusiastic
aaron	15	V	14	0	enthusiastic
samuel	14	I	8	1	insecure
gretta	14	III	8	1	enthusiastic
duncan	10	II	5	0	enthusiastic
slone	12	II	8	0	enthusiastic

The next chapter will further analyze the particulars of each student's experience with the curriculum and discuss the implication of the various successes and failures.

5 Analysis and Discussion

This chapter addresses several pertinent issues regarding the imaging curriculum. First, it presents a brief description and discussion of each individual student's experience with the imaging curriculum and also highlights consistent observations. Next, the chapter reflects on what kinds of learning might take place for students of the imaging curriculum. This is followed by the linking of salient observations together in order to pinpoint the specific ways in which the imaging tools catalyzed the learning effort. The last section attempts, in a theoretical sense, to critique the consistency of the object design.

5.1 Individual Experiences

The following descriptions of students' experiences are grouped as the students were themselves grouped when they worked with the curriculum. The students are presented in an order relative to when each of them began the curriculum. Each description includes notes on:

- **grade & gender**; **source:** how did each student come to be involved?
- **interests:** e.g. mathematics, art, etc.
- **iteration**: approximately how many times had this material already been taught to someone else before the case study in question?
- **setting:** what was the working environment?
- **time** (spent on the material)
- **attitude:** what was the student's attitude towards this learning experience?;
- **level** (of accomplishment)
- **notes**
- **selected interpretations** derived from the current description

5.1.1 Maureen and Betty

Grade & Gender: Maureen and Betty were two 7th grade females.

Source: Both girls were recommended to me by their Middle School computer instructor.

Interests: Both had a general interest in art (this was considered important, given the graphical nature of the SchemePaint application).

Iteration: 0 (pre-design phase).

Setting: We worked after school in their Middle School computer lab; each girl had her own computer. The screens were small and the computer memory was inadequate, resulting in frequent and frustrating crashes of the application.

Time: Seven weekly one hour sessions; no homework.

Attitude: Both girls were insecure about their ability to learn the material. They often approached tasks murmuring about their inability to complete them. They were very concerned about the possibility of other students from the school (passers-by) observing their work.

Level: Neither student became fluent with the idea of parameter passing. They often made the mistake of using the actual value of an argument in the procedure definition instead of a formal symbolic representation.

Notes: Maureen and Betty were very worried about the social ramifications of working after school, which severely curtailed their attention span. They came into the program leery of computers and were never able to move beyond this fear. Insufficient computing resources exacerbated an already frustrating experience.

Selected Interpretations: a) Setting and resources are critical to student performance; b) The computer was a difficult artifact for these girls to relate to.

5.1.2 Hector

Grade & Gender: Hector was a 6th grade male.

Source: Hector frequented the Middle School computer lab. He saw Maureen and Betty working with the curriculum and asked to join.

Interests: Hector was interested in computers. He had some programming experience and was a member of the computer club.

Iteration: 1. Although this was the pre-design phase, there was enough of a time lapse between his sessions and those of the previous two girls such that the curriculum was modified.

Setting: We worked after school in a small faculty office with adequate computing resources.

Time: 12 weekly one hour sessions broken approximately in half by a summertime recess; some homework.

Attitude: Hector was enthusiastic and bright. He worked on novel tasks with pleasure. Towards the end of the curriculum, his interest waned. He explained that he was more interested in working on C programming with his friends.

Level: Hector became very comfortable with procedural arguments (property 2) and marginally comfortable with property 3.

Notes: Hector was a self-motivated computer programmer. Upon entering the program, he had many standard programming concepts mastered. Property 3 was the most troubling concept for Hector. He also showed some confusion with color objects.

Selected Interpretations: a) Property 3 is the most difficult for students to master; b) Experience with imperative programming may have accounted for some of the difficulty with functional programming concepts; c) Anonymous objects of any sort present difficulty.

5.1.3 Brooke

Grade & Gender: Brooke was a 5th grade female.

Source: Brooke was the daughter of a local high school teacher (a classmate of mine).

Interests: Brooke was interested in the performing arts. She expressed an explicit distaste for mathematics. She had no programming experience.

Iteration: 2. This was the first study which used physical manipulatives.

Setting: We worked after school in a spacious room with a large computer and dry-erase board.

Time: 14 weekly one hour sessions; no homework.

Attitude: Brooke was enthusiastic and meticulous; she was a self-proclaimed perfectionist.

Level: Brooke became comfortable with procedural arguments (property 2) and marginally comfortable with property 3. By the end, she was able to describe the execution of a property 3 function only with the assistance of the manipulatives.

Notes: Brooke was not an experienced programmer or natural mathematician. Perhaps because of her interest in performance, she hooked onto the manipulatives as a form of representation and *always* used them to work out problems.

Selected Interpretations: a) Manipulatives helped Brooke solve problems using a tangible method that the computer did not provide; b) Manipulatives helped Brooke

work with functions that returned other functions (property 3) by making all of the pieces of the process accessible and sensible.

5.1.4 Lydia

Grade & Gender: Lydia was a 22 year old female undergraduate.

Source: Lydia was working with SchemePaint as part of a class project about how people learn programming.

Interests: Lydia was interested in computer tutoring and graphic arts.

Iteration: 2. This occurred essentially simultaneously with Brooke.

Setting: We worked in a spacious room with a large computer and dry-erase board.

Time: Nine weekly one hour sessions; homework involved reflecting on her experience with the curriculum and keeping a journal.

Attitude: Lydia was interested and enthusiastic. She described this as the best educational experience she had ever had.

Level: Lydia was able to use property 3.

Notes: Lydia was in favor of the use of manipulatives as well as one-on-one educational experiences. She expressed a level of discomfort with instantiations of property 3 because "you don't define everything that you use in the input." While she was able to use property 3, she never described herself as comfortable with it.

Selected Interpretations: a) One-on-one experiences are unrealistically favorable; b) Comfort (which was very important to the student) is a less common feeling with higher-order procedure processing because of a lack of analogous experience with it in other domains.

5.1.5 Sonny

Grade & Gender: Sonny was an 18 year old 10th grade male from Nepal.

Source: Sonny was the friend of a fellow graduate student.

Interests: Sonny was interested in running.

Iteration: 2. This occurred essentially simultaneously with the last two case studies.

Setting: We worked after school in a spacious room with a large computer and dry-erase board.

Time: Three weekly one hour sessions; no homework.

Attitude: Sonny was always enthusiastic when present; however, he rarely attended scheduled meetings. He began to forfeit sessions of the curriculum in favor of track practice and eventually formally dropped out. His English skills were poor.

Level: Sonny mastered the use of basic turtle commands.

5.1.6 Laurence and Jonathan

Grade & Gender: Laurence was a 4th grade male. Jonathan was a 6th grade male.

Source: Laurence and Jonathan were the sons of a computer science graduate student.

Interests: Unknown.

Iteration: 3.

Setting: We worked in a small room with two large computers and dry-erase board after school.

Time: 10 weekly one hour sessions after school; no homework.

Attitude: Both brothers were energetic. Laurence was more outgoing and hasty; Jonathan was quieter and tentative.

Level: Neither boy progressed beyond property I (although we did touch upon functional arguments in the final session).

Notes: Having both boys in a small room created a behavior problem. They would frequently bang on the keyboard until it beeped. Neither boy was prone to reflection about the problems that they worked on, resulting in a trial and error style of learning. Both boys had trouble with parameterizing functions.

Selected Interpretations: a) Two students in a small room is inefficient for social interaction reasons; b) The curriculum is not good at motivating or drawing the non-self-directed student; c) Trial-and-error does little in this case to foster overall concept acquisition; d) Trouble with parameters to functions could be age related.

5.1.7 Patty, David, and Joseph

Grade & Gender: Patty was a 9th grade female; David and Joseph were 9th grade males.

Source: All three students signed up to do the project as part of a workshop requirement for their high school.

Interests: No academic interests were expressed by the students.

Iteration: 3.

Setting: We worked during the middle of the school day in a small room with three large computers and dry-erase board.

Time: 10 semi-weekly half-hour sessions; no homework.

Attitude: All three students were hostile to the project and thought it was a waste of their time.

Level: None of the students progressed beyond property 1. One student never mastered property 1.

Notes: The student's were unwilling to work with the manipulatives. Students were much more interested in the direct manipulation aspect of SchemePaint (and other applications on the computer) than in programming. The graphics in SchemePaint were not perceived to be dynamic enough.

Selected Interpretations: a) Interesting applications of SchemePaint and higher-order procedures should be introduced early in the curriculum to pique the interest of non-intrinsically motivated students; b) Physical manipulatives are perceived as dumb by students of this age range and motivation level.

5.1.8 Nate and Donald

Grade & Gender: Nate was a 10th grade male; Donald was a 9th grade male.

Source: Both students signed up to do the project as part of a workshop requirement for their high school.

Interests: Nate is an interested and experienced computer programmer (of the "hacker" variety). Donald is computer illiterate.

Iteration: 4 (final).

Setting: We worked during the middle of the school day in two small rooms each with a large computer and dry-erase board; the instructor moved from room to room.

Time: Donald stayed in the program for six semi-weekly half hour sessions before dropping out in order to run for student government; Nate stayed with the program through the duration of his workshop session (12 semi-weekly half hour sessions) and then re-joined for an independent study (nine additional hours in six one and a half hour blocks); some homework.

Attitude: Nate is arrogant and confident. Donald is quiet and has a learning disability. Both are enthusiastic.

Level: Donald dropped after an introduction to procedural arguments, never completing a homework assignment on the subject. Nate became proficient at properties 1-3.

Notes: Nate thought that, by comparison to BASIC, functional programming was dumb. He had some initial problems with moving beyond assertive-style

programming semantics. Overall, Nate had very few serious problems in the curriculum. Nate was able to trivially write the slope function. He had problems with the introduction of recursion and dropped the curriculum there.

Selected Interpretations: a) Imperative programming ideals provided an irritation if not a serious obstacle to higher-order programming concept acquisition; b) Imperative programming did seem to create problems once recursion was introduced; c) Recursion is perhaps another abstract concept whose pedagogy is a needy candidate for imagery based instruction.

5.1.9 Aaron

Grade & Gender: Aaron was a 10th grade male.

Source: Aaron signed up to do the project as part of community apprenticeship requirement for his high school.

Interests: Aaron has done some programming and is generally interested in academics.

Iteration: 4 (final).

Setting: We worked during the middle of the school day in a large room with a large computer and dry-erase board.

Time: Seven two hour weekly sessions; weekly homework.

Attitude: Aaron was calm, persistent, and goal-oriented.

Level: Aaron mastered properties 1-3 as well as recursion and recursive functions using property 2.

Notes: Aaron was a generally fast and insightful learner. He seemed motivated by a desire for knowledge. He claimed in an interview that he found physical manipulatives unhelpful. He always completed his (always challenging) homework correctly.

Selected Interpretations: a) This case study raises issues about the utility of the manipulatives for different sorts of learners.

5.1.10 Samuel and Gretta

Grade & Gender: Samuel (male) and Gretta (female) were in 9th grade.

Source: Both students signed up to do the project as part of a workshop requirement for their high school.

Interests: Gretta hates math but her father makes her do it; Samuel thinks computer graphics are neat and he wants to author a computer game eventually.

Iteration: 4 (final).

Setting: We worked during the middle of the school day in two small rooms each with a large computer and dry-erase board; the instructor moved from room to room.

Time: Eight semi-weekly half hour sessions; no homework.

Attitude: Gretta is quiet and amenable. Samuel is quiet and interested in flashy graphics. He is interested not in thoughtfully doing the work but merely in the final product.

Level: Samuel only reached level I and his comfort with parameters was never high; Gretta used the manipulatives to do level III problems.

Notes: Samuel was slow with the curriculum because of a general lack of interest in thinking about or solving problems himself; he was perfectly content to let me work through a problem for him. He never tried solutions on his own, rather, he always gave up with little or no attempt and asked for my assistance. He was not enthused about using the manipulatives. On the other hand, Gretta was comfortable with the manipulatives and seemed quite motivated by the pride of finishing a task on her own. Even on tasks that were well beyond her expected ability, she was able to make very educated guesses. She is one of the students who generated property 2 on her own. Gretta never "graduated" from manipulative use to a point where she could generate property 3 functions only with the computer and without the use of the manipulatives.

Selected Interpretations: a) The curriculum does little to improve the motivation of students; b) A direct correlation between students' motivation, success with functions as objects, and willingness to use the manipulatives seems to be emerging.

5.1.11 Duncan

Grade & Gender: Duncan was a 5th grade male.

Source: Duncan was the son of a personal friend.

Interests: Duncan liked art, but not math. He had no experience with computer use.

Iteration: 4 (final).

Setting: We worked after school in a large room with a large computer and dry-erase board.

Time: Five weekly one hour sessions; no homework.

Attitude: Duncan is excitable but able to concentrate when he tries.

Level: Duncan was able to do some work with property 2.

Notes: Couched in a decidedly jovial manner, Duncan was successful at using property 2. He named his functions things such that expressions might look like, for example, `(frig frog)`. He seemed to be more interested in the work when it was a game. He was able to use the manipulatives, but was not particularly detailed in his use of them; he enjoyed picking them up and throwing them around but not composing them.

Selected Interpretations: a) The game-like element of the manipulatives and Duncan's naming schema points to a tendency to prefer games which may prevail among younger students.

5.1.12 Slone

Grade & Gender: Slone was a 7th grade male.

Source: Slone was the son of a computer science graduate student.

Interests: Slone liked computers and was just becoming familiar with them.

Iteration: 4 (final).

Setting: We worked after school in a large room with a large computer and dry-erase board.

Time: Five bi-weekly one and a half hour sessions; no homework.

Attitude: Slone is a "class-clown" but able to concentrate when he tries. He likes to talk about things other than SchemePaint, like Star Trek. His work is careless syntactically: he never gets it right the first time, but is always able to fix things upon second glance.

Level: Slone was able to do some brief work with property 2 (using `hexagon` and `zig`). He felt very comfortable with the `map` procedure and created interesting uses for it.

Notes: Slone was a slow learner, if only for his inability to concentrate on one subject. Had we had more time, I feel that he would have progressed further.

Selected Interpretations: a) Two weeks between sessions is too much time for students to recall the material. b) One and a half hours seemed to be the ideal session length for covering material and simultaneously maintaining students' interest.

5.2 General Discussion

All students who reached level III of the curriculum initially exhibited symptoms of the missing variable problem. However, 4 out of 5 students[18] surveyed answered `(apply-to-five create-subtracter)` correctly on a written test. In all but one of these cases, this was also the student's first introduction to non-graphics Scheme code.

The two students who spent ample time (Nate: 21 hours, Aaron:16 hours) in an advanced stage of the curriculum were able to correctly apply their knowledge about functions as objects from the domain of graphics to the domain of mathematics by writing a `slope` function.

In the final iteration (4), 4 of the 5 students reached level II within five hours of instruction. These were both students with prior programming experiences as well as students with little or no computer experience at all.

It is worth noting that 10 out of 18 students did not progress beyond level I of the curriculum. This figure is not significantly different than the results obtained by [Eisenberg, Resnick et al. 1987] in their study of MIT students. However, it is important to note the differences in setting between the two situations. The subjects in the imaging curriculum had no special mathematical training and, in most cases, no similar academic experiences or background in which to frame their learning. The "successful" students who reached levels II and III were *not* MIT students: they were as young as 10, 11, and 12 years of age.

[18]The 6th student who reached level III did not test on this problem.

5.2.1 Age and Gender

There are a number of sociological issues that come into play in an effective manipulative design. Both gender and age seemed to influence the type of reception that the concrete tools received.

Its interesting to note that the manipulatives were denounced by most of the male subjects as being too "girly", leading me to believe that had I decided to represent functions as dump trucks carrying procedure logs they might have been more successful at working with the representation. Perhaps their reluctance was due to the "soft" tactile nature of the objects. All the males that were questioned about the objects had a negative response to them; conversely, the female responses were all positive. Although the boys' introspective feedback regarding the physical manipulatives was largely negative, there were many ways in which *all* of the students' actions and language indicated that manipulative use had left an impression upon them. This will be discussed further in section 5.4.

Two of the five students[19] who got to level III were female; these two students were also novice programmers; they both were able to work at level III only in the presence of the manipulatives. The three male students who got to level III all had prior programming experience and all preferred to work without the manipulatives at this level of the curriculum. This perhaps implies that the manipulatives were better utilized by the males for general concept formation and better utilized by the females for careful and thoughtful explanation of and repetitious practice with certain phenomena[20]. This relates back to Lydia's comment that her main dilemma with property 3 was over a lack of "comfort" with the concepts. The experience of

---

[19] Discounting the undergraduate female who also reached level III.

[20] Klawe [Klawe 1995] has demonstrated that different computer interaction protocols result in statistically significant performance differences for males and females.

practicing with manipulatives seems to foster a level of familiarity for girls that they do not otherwise generate on their own. Recall the perceived lack of ability to master and unwillingness to attempt novel tasks expressed by Maureen and Betty during their work with SchemePaint in the pre-design phase. Klawe [Klawe 1995] has similarly reported that when children are presented with familiar and novel computational tools boys are equally likely to select novel tasks as they are to select familiar tasks. Girls, on the other hand, showed a marked tendency to choose familiar tasks.

Boys with no prior programming experience seemed to be the least successful as a whole (they were also the largest group). Of the ten novice males who began programming through the imaging curriculum, only two even reached level II. This is again consistent with the results of [Klawe 1995]: in a two month study of thousands of children using video and computer games it was observed that boys tended to seek out fast action and novelty. The tasks presented to them through the imaging curriculum required a duration of attention which may have been above their standards (see also section 5.2.2 on motivation).

Age also seemed to affect the students' perception of the tools. Younger students took pleasure in the use of the objects; middle school and high school students generally found them "stupid"; older students found them tedious but useful, exactly as might have been expected. In general, there seemed to be no significant correlation between age and success. However, it is interesting to note that the younger of the students were teetering at the age of what Piaget refers to as the *Formal Operations Stage* [Piaget 1966]: the point when children generally are able to begin to perform "operations on operations", or, when they begin meta-cognitive awareness.

That younger students were able to discipline themselves *at all* to learn the material may seem surprising. In fact, this result is consistent with the style of Montessori

education (see also section 6.4) which emphasizes that younger children can focus -- in fact, thrive -- on meticulous controlled tasks for long periods of time.

5.2.2 Interest and Motivation

Many of the students who did not reach level II in the curriculum were reported in section 5.1 to have some form of diminished interest. Although the imaging curriculum was a critical learning tool for some students, it did nothing to overcome a general lack of interest on the part of other students. All students who maintained interest in the subject matter were able to progress. It must be admitted, nonetheless, that there is a certain element of circularity in this argument: conceivably, those students who retained an interest in the subject did so in part because they were experiencing success with the curriculum.

Many factors seemed to influence students' performance. In the pre-design phase, it was clear that Betty and Maureen (the only female-female work pair) were too self-conscious about the opinions of their peers to concentrate particularly hard on their work. They also seemed concerned about failure relative to each other. Initially, their teacher had recommended them both because they were personal friends with similar interests. However, this created a dynamic of self-consciousness between them: neither girl wanted to fail in the eyes of her friend nor was she interested in "competing" to achieve something that her friend was possibly incapable of completing. While this is only one example of this behavior within the curriculum, it is also consistent with well known characteristics of learning for girls.

One of the male-male pairings, Laurence and Jonathan, were also unable to progress very far in the curriculum, but for different reasons. In this case, the boys were easily distracted by each other over disconnected activities, like changing the volume on the

computer or repetitiously punching keys on the keyboard to see what would happen. As mentioned before, the pace of and material in the curriculum did not seem to meet the boys expectations. They seemed unable to concentrate for long periods of time on details of the learning process. As mentioned earlier this was a behavior that was common to all novice males, although it was particularly severe in the male-male pairing. In general, students exhibiting a cognitive tempo [Johassen and Grabowski 1993] of reflectivity (as opposed to impulsivity) were more successful with the imaging curriculum.

All the successful students were tutored on a one-to-one basis except for Gretta who worked in an adjacent room, set off from her fellow male student. Are any of the successful results from these one-on-one tutoring experiences scalable to a classroom setting? Although success did not seem to scale to the one-on-"few" learning situations, a classroom setting may actually prove more optimistic. Students of this curriculum differed from classroom students in many ways, including a lack of extrinsic motivation. Students were not motivated by grades (they received none), prospects of advancement (in their eyes, the curriculum was terminal) or application (it was hard to see how the work they were doing was useful beyond mathematics which to most younger students is not useful in and of itself).

Based on observations, particularly those of the novice male programmers, the imaging curriculum did little to draw students who were not independently motivated or who were lacking concentration skills. One could imagine scaling the imaging curriculum to a classroom setting by dividing into groups of students working independently from the instructor in a collaborative effort. This task is left as an open research direction.

5.2.3 Multiple Representations

In this instantiation of the imaging curriculum, three distinct representations were used: computational (SchemePaint), visual (dry-erase board) and physical (felt pillows). It is the claim of this work that presenting the student with an array of views on an object (see also [Polya 1945]) helps to build a more persistent and complete mental representation. Gardner describes seven "intelligences" that learners tend to favor in their cognitive abilities. These are: logico-mathematical, linguistic, musical, spacial, bodily-kinesthetic, interpersonal, and intrapersonal [Gardner 1993].

Any one student of the imaging curriculum seemed to show an affinity for some subset of the representations: they were usually able to informally "rank" the tools from best to worst (see the interviews from sections 4.3.1, 4.3.2 and Appendix C) by stating a preference. Rankings varied from student to student and across gender.

Whether for the purpose of repeatedly encouraging visualization or for the purpose of catering to individual learning styles, the net result of a multiply represented system was more a more exhaustive style of concept formation than that of a curriculum founded on a single or even a dual representation. The example concept of FD had three representations: perhaps additional representations, covering a wider range of the cognitive abilities described by Gardner's seven intelligences, would have been all the more productive.

5.3 Critiquing the Learning

5.3.1 Quality

Arguably, there is a difference in the quality of the learning that takes place through the imaging curriculum. Imaging supports a correct high-level understanding which

is also a deeper understanding. It is a knowledge foundation from which students may build new and more advanced ideas. This type of learning is different but potentially complementary to other contemporary styles of learning (e.g.: situated learning, intentional learning, constructivist learning; see [diSessa 1993] for an elaborated discussion of issues related to learning style). While other methods may be useful for linking abstract knowledge with real world experience, they are not necessarily intended to make students comfortable with the abstraction itself. It is this latter goal that is the subject of this work: making students comfortable with the abstract concepts which underlie their experiences.

Comfort with abstracting can be gauged on three different levels, presented in increasing order of depth:

Recognition: The ability to recognize an example of the underlying target concept.
Application: The ability to apply the target concept to a novel situation.
Generation: The ability to generate the underlying abstract target concept.

Figure 14 compares perceived ease of cognition of the different properties of FD at the different levels of depth. A darker shading implies stronger support. The number in the internal white box indicates the percentage of successful students for each group of students that reached a certain level (I-III) of the curriculum.

By contrast with evidence from Section 2.2 of the background research, we can note where the imaging curriculum had its most vivid effects:

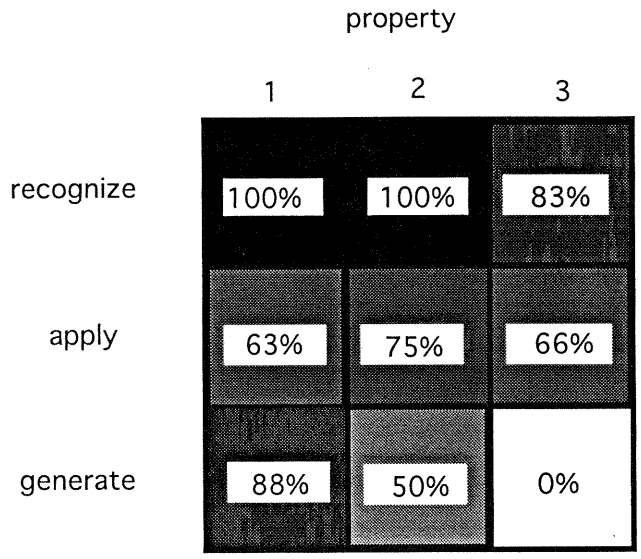


Figure 14: Effects of the Imaging Curriculum
For each group of students who reached a particular level I-III (directly corresponding to Stoy's properties 1-3) of the curriculum, this diagram indicates the percentage who were able to perform the cognitive tasks of recognition, application and generation.

• **1A**: Students of the curriculum are able to apply the idea that all objects have names to successfully and succinctly rename functions, e.g. `(define first car)`.

• **2G**: Some students generalize the concept that a function can be an argument to another function, e.g. `(define (circle dir) (repeat 36 (fd 1) (dir 10))`.

• **3A**: Some are able to generalize the concept of functions as returned values from the domain of graphics to that of mathematics, e.g. `(apply-to-5 create-subtracter)`.

5.3.2 Grain

There are two grains at which learning occurs in the imaging curriculum for the current target concept. Most obviously, object centered imagery tools foster general knowledge acquisition (coarse grain) of an object concept of data. The claim here is that it is with such general but thorough understanding that some students are able to apply (and even generate) property 2.

On a more observable level, manipulating 2 and 3-D objects is an effective way to verify the correctness of computer code. The objects provide a map through the debugging process (fine grain) that is absent from the debugging of straight text. Additionally, the act of (quite literally) "performing" the execution of commands puts the student more personally in touch with the process. By playing the role of the interpreter, students carefully learn what types of things are valid and conceivable. The process, though somewhat tedious, reconfirms the rules of evaluation for students, making them less likely to repeat their mistakes.

5.3.3 Quantity

Situations where the student-to-teacher ratio is low are often considered to be more pampered learning environments. This situation is no different: students are given an unrealistic[21] amount of instruction and learn a proportionately small amount of material (albeit, well-learned material). This is questionably practical for a real classroom setting. As mentioned previously, one could imagine modifying the curriculum for a classroom through the spirit of computer supported cooperative work. The question, then, is one of priorities: is time spent on learning general cognitive and higher mathematical skills worth the time required by the imaging curriculum?

5.4 Imaging "Fingerprints"

Upon completing the curriculum, children aged 10-13 demonstrated expertise in abstraction and the use of higher-order functions that rivaled that of undergraduate and graduate students studying the same topics in a classroom setting (from the

[21]Unrealistic, that is, for a real classroom setting.

preliminary research phase, Section 2.2). How do we know that the imaging curriculum has anything to do with this fact?

The answer to this question can be found by examining the actions and the language of the students who used the curriculum. Some of the cues may be given very explicitly by the students and others may be colored more by residual reflections of objects, for example:

- Students introspectively describe their own notions of functions as being very object-oriented.
- Even in the absence of the objects, students motion with their hands in a "container" shape as they describe the workings of the Scheme system.
- When students speak of procedure objects they use action-on-object verbs like throw, get, make-a-new, gives-back, etc.

These collections of identifying impressions which characterize student's behavior in the absence of the tools themselves will be referred to as *imaging fingerprints.*

But what of a student like Aaron who seems to have "graduated" from the manipulatives so shortly after his introduction to the curriculum? Or what about another student who claimed to have an object concept of functions before ever beginning the curriculum? One answer to the question of how this curriculum benefits someone like Aaron is to simply shrug: it does not matter how the student got the object concept of data; rather, what is important is the realization that Aaron's object knowledge of this concept has made him a creative programmer set apart from other novices. One of the goals of this work is to demonstrate how imaging helps to develop the abstract mind. Aaron's claim that he already thought in object images -- coupled with his stellar performance in the curriculum -- only reaffirms this claim.

Still, an answer like the one above dismisses many of the important benefits that a student -- particularly someone like Aaron -- can reap from learning about functional

data. To begin with, the imaging curriculum can only help reify existing notions of function. It provided a formal setting in which someone who was clearly ready to explore these issues could explore ideas to which he would otherwise not have explicit exposure for another five years of education, if at all. Early introduction to the concept of functional data -- a watershed issue in higher mathematics -- can only be beneficial.

Anomalies like Aaron aside, the curriculum is designed more for that majority of students who do not have *a priori* knowledge of abstraction. For many of these students, the curriculum is a stepping stone that helps the mind to extend beyond its initial reach. With the assistance of cognitive placeholders, students can practice at a level of programming for which they are unable to otherwise maintain an attention span. Statistics from Section 2.2 indicated that university students still have a hard time maturing to this level of mathematical thought; perhaps such cognitive blocks can be avoided if students are thoughtfully exposed to this material at an early age.

5.5 Critiquing the Design

For each of the three imaging tools used [**C**omputational, **V**isual, **M**otor] there are several questions that need to be addressed by an effective curriculum design:

1. Is the design consistent with correct ontology? [i.e. Does the design embody the necessary characteristics of FD under the correct world model? The representation must (a) be object-like in nature; (b) have some embodiment of its ability to execute; and (c) have similar construction to other objects in the language.]

C: No. (a) Pure programming environments do not support object representation. This is well supported by previous research [Eisenberg, Resnick et al. 1987] which confirmed that students favored an active model of functions. (b) Pure programming environments do, therefore, actually put disproportionate weight on the characteristic

of executibility. (c) Procedure objects are *constructed* differently than other objects in the language and therefore are assumed to have different *construction* [22]. So, for example, note the fact that in Scheme when one asks the interpreter to return the associated value of a named list we get back *the list itself* , a representation of the constructed object. However, when the interpreter returns the associated value of a named procedure object the result is devoid of any information about the procedure and is, in fact, useless. A seemingly more appropriate result might be the actual procedure -- the closure -- referred to by a particular procedure name. In fact, when questioned on a written test as to the result of a name reference to the `create-subtracter` procedure, one student incorrectly yet intuitively soundly responds that it "returns the body of the function `create-subtracter`."

V: (a) A 2-D drawing of a procedure object has more of an object-like appearance than its purely computational counterpart; still, it lacks the tangibility of a physical representation. Nevertheless, students explicitly note that they prefer this form of representation. Recall also that Brooke *always* chose to use this representation to work through her assigned tasks. (b) In the absence of a real interpreter, it becomes the student's responsibility to act as interpreter when working with any non-computational representation. Although the "acting out" process supports learning and memory (recall Aaron's experience debugging his code), it also somewhat de-emphasizes the execution powers of a function. This has not proved problematic, however, for the students owing perhaps to the very strong sense in which functions are active things, and (c) Objects are constructed uniformly, each with a surrounding box and each with its own content. When Brooke began to work on a section of code, she always started with a visual representation, drawing boxes for each object in an expression.

M: (a) Concrete representations strongly support an object concept of data in virtue of being able to be literally handled. Gretta made the association most tangible when asked if she has an image of a procedure: "Kind of since I have been shown a table or a square object. It makes it easier to think about it." (b) Again, the objects on their own are fairly static and de-emphasize their active quality. The object nature of

---

[22]The notion of construction in itself is potentially a strange one for functions since, in virtue of being non-object-like, they lack the concreteness implied by the notion of construction. This is, of course, exactly the problem at hand.

function is stressed to the student. On the initial questionnaire, when asked to complete the sentence "Procedures are _____", all students responded "objects." (c) The constructions of all objects share similar properties (e.g. they are all felt pillows). Slone expresses the effect well when asked if there is a difference between number and procedure objects "Well, they're different but its not like one is more important than the other."

2. Is the design inconsistent with the incorrect ontology?

C: No. As just one example, the standard syntax for naming procedure and number objects is quite different. Compare:

number:	procedure:
`(define test 10)`	`(define (test x) (+ x x))`

Recall from section 4.1 that Brooke expressed confusion over exactly this problem. Further, students are shown a description table on their first written test which contains names and associated objects for a number and a procedure object. They are then asked to come up with the command that would have resulted in that table configuration. Overwhelmingly, students are unable to define the number object but correctly define the procedure object.

V & M: Yes. The design does not support any kind of special status for procedure objects. After having his first experience parameterizing a function, Samuel [23] was asked if the use of procedure and number objects as arguments seemed symmetrical: "It seems like it made sense 'cause that's what I was thinking earlier when I was trying to make that zig thing. I was thinking you could do this sort of replacement." Note that the student's thought refers back to a time prior to the explicit introduction of functional arguments when he had considered the idea on his own.

3. Does the design support imaging?

C: Weakly. Exercising the powers of imagination, it is conceivable that procedures could be viewed as units defined by their syntactic delimiters (the outer defining

[23] Note that Samuel is not characterized as a graduate of level II because his only experience with the issue was when it was presented to him in a final interview.

parentheses in the case of Scheme). Recall that Aaron described having a "block" notion of function prior to his work with the curriculum. It is questionably imageable; it is unquestionably weak.

V: Somewhat. Again, objects drawn on a board are still abstractions. On the other hand, any picture is more imageable than a linguistic description. Students varied in their personal preference as to the use of visual or physical manipulatives. In general, males preferred the visual representation. According to Slone: "I like it better when its drawn [on the board]. These things mess with my mind."

M: Yes. Circular as this may seem, the design supports imagery of objects because the manipulatives *are* tangible objects. In general, females preferred the physical representation. When asked about the utility of the objects, Gretta responds: "Humans make minds think they should see everything concretely so it helps when you can see it concretely."

4. Is the design consistent with fine grained models? [e.g. Scheme Environments]

C: Yes. A really fine-grained ontology of how information is managed in Scheme environments can be found in [Eisenberg 1990]. Computational environments are consistent with this (although, it might also be argued that they do nothing to necessarily foster the Scheme environment model).

V: Potentially. A 2-D representation would seem to encourage the visualization of environments and data in addition to being consistent. However, due to the fine-grained nature of this system of representation it was impractical to use this representation (again, found in [Eisenberg 1990]) with the imaging curriculum. While the visual representation that was used in *this* curriculum was not particularly consistent with the environment model, the representation used by Eisenberg *is* a visual representation. Potentially, the current model could evolve into the environment model [Abelson and Sussman 1985].

M: No. This system is not supported by the imaging curriculum: it supports a much higher level concept of functions as objects. Although Aaron ascribed properties to functions which were consistent with the environment model the curriculum itself is admittedly too coarsely grained to explicitly support this view.

5. Is the design inconsistent w/ fine grained errors? [e.g. problems of anonymity]

C: Theoretically, yes. This may be a surprising answer. In fact, though, the user manipulates unnamed objects freely and commonly: number objects are, in a sense, anonymous. On the other hand, numbers are unique in that, unlike functions, they embody their own single chunk of information. Despite this inherent symmetry between unnamed number and procedure objects, the observations of section 2.2 indicate that this is a recurring problem in standard pure programming environment curricula. So in an empirical sense the answer would seem to be no.

V: At a low level, yes. The process of working through an example with the visual representation allows the student to find errors they would otherwise overlook. This technique was used effectively by every student who made it to level III in the curriculum. Two of the students, Brooke and Gretta, were not able to write code at level III without the use of the manipulatives. At a higher level, it is less clear how useful the objects are in forming a high level concept of anonymity. There were no instances, for example, where curriculum participants *generated* property 3 of functions. For more coarse grained concept acquisition, then, the visualization method seems less useful.

M: Yes. In addition to the above properties of visual objects which are shared, the manipulatives uniquely demonstrate how a new procedure object is "born". By analogy, consider the case of a new born child: although the child may be nameless, it still has a very tangible existence. Again in the cases of Brooke and Gretta, an increased understanding seemed to come from the motor process of retrieving a blank procedure object for a `make-procedure` call and then actually filling in its body and arguments. This seemed stronger, in contrast, to the case where they only used the pure visual representation to draw a new procedure object. In both cases they were able to write the correct visual representation of a function call but they were not able to *make novel use of* that function until after they had worked through the motor representation.

6 Related Research

The work presented here is a subset of no single traditional body of research; rather, it taps previously unrelated research from a variety of disciplines thereby filling an educational gap. So far, this research has presented several related views: current pedagogical dilemmas in mathematics and computer science have been linked to its historical roots; neglected problems in higher order thinking have been specifically addressed; basic and higher math education have been used as sounding boards for a unified method of teaching; theories of mental imagery typically used to foster concrete thinking have been used to address the teaching of abstraction.

To recap, the argument presented at the beginning of this work for validating its importance went something like this:

- The concept of function -- particularly higher-order function -- is of paramount importance in mathematics and computer science;
- Students learning about functions -- particularly higher-order functions -- chronically exhibit error-ridden learning behavior;
- The core difficulty with functions as data objects is a specific instantiation of a more general problem with abstraction;
- Imagery tools can be usefully employed to help students reify abstraction.

The following sections of this chapter will summarize the empirical foundations for these claims.

6.1 The Importance of Function

Both historically and currently, the notion of function pervades mathematics and the sciences (recall the ruminations of the founders of computer science related to the

notion of higher-order function [Babbage 1842]; [Lovelace 1842];[Turing 1937];[Von-Neumann 1945] as described in Chapter 2). In present day computer science, functions and procedures provide the underlying organizational structure for all programming languages [Sethi 1989]. On top of the already critical notion of function is the powerful notion of functional data. Sethi further describes this phenomenon:

> "The pure lambda calculus has just three constructs: variables, function application, and function creation. Nevertheless, it has had a profound influence on the design and analysis of programming languages. Its surprising richness comes from the freedom to create and apply functions, especially higher-order functions of functions." [Sethi 1989]

We find functions to be of equal import in mathematics as the foundation of calculus:

> "Roughly speaking, calculus is the mathematics of change. Particularly, calculus is a powerful tool for understanding change in physical quantities and phenomena that depend on or relate to each other. The dependence of a given quantity on another is often described mathematically by a function; thus, the heart of calculus is the study of functions and how they change. The differential calculus studies instantaneous change of a function as quantities vary and the integral calculus measures the cumulative effect of the change of a function." [Thomas and Finney 1993]

And in physics:

> "Similarly, the union of the concept of a variable function with the ideas of contemporary algebra and geometry produced the new functional analysis. Just as analysis was necessary for the development of the mechanics of time, so functional analysis provided new methods for the solution of present-day problems of mathematical physics and produced the mathematical apparatus for the new quantum mechanics of the atom. History repeats itself as usual, but in a new way, on a higher plane." [Aleksandrov 1963]

The power of higher-order function is further noted by Cajori: "Without a well-developed notation, the differential and integral calculus could not perform its great function in modern mathematics [Cajori 1928]."

To summarize, functions and higher order functions, both historically and currently, allow for the robust and elegant expression of ideas across many scientific disciplines.

6.2 The Difficulty with Function

Although clearly an important concept, the earlier quotes from the icons in computer science history tell us that functions were likewise perceived as notably difficult. This instinctual problem is perpetuated by current day programming paradigms which neglect constructs for supporting functional objects. Even some functional languages support a naive concept of function. For example, in Common LISP, the same symbol can be attached both to a function and a value, necessitating the use of the LISP `funcall` primitive. As a result, procedures can not be directly abstracted in the same way that numbers can indicating, in some sense, a different status [Winston and Horn 1989].

A recent anthology published by the Mathematical Association of America dedicated itself entirely to examining problems with the concept of function [Harel and Dubinsky 1992]. One selection from that volume [Sierpinska 1992] cites the "widely reported and well known" student difficulty with the concept. Sierpinska specifically notes how detrimental preconceptions of function can be for conceptual development. In fact, this problem is most exaggerated in the case where functions are operating as data objects.

It is prudent, before examining the genesis of the object concept of function, to probe the historical development of an even more common data object: the number.

> "We shall see... that the "abstraction" of the number sequence from the things counted created great difficulties for the human mind. We need only ask ourselves: how would we count if we did not possess this sequence of remarkable words, 'one,' 'two,' three,' and so on? ... [O]ne achievement of our number sequence is its independence of the things themselves. It can be used to count *anything*." [Menninger 1969]

As the above quote indicates, the challenges identified relative to the genesis of the number system are parallel to the problems we currently see from students of function; this may seem an unlikely assertion given that most adults take for granted the idea that a number can be a piece of data. Nonetheless, there is evidence that in early civilizations people indeed had difficulty with the transition from an "attribute" to an "object" concept of number—for instance, the separation of the concept of "fiveness" from its object of cardinality (5 oxen, 5 fingers, etc.) [Menninger 1969]. Moreover, early civilization's concept of number reappears developmentally in present-day children's initial concept of number [Hughes 1986]. The central challenge then, both for modern children and ancient adults, lies in separating the objective and abstract nature of number from the "thing to be counted" [Menninger 1969]. Aleksandrov points out that this difficulty with abstraction extended beyond the number system: "In a completely analogous way, certain peoples had no concept of 'black', 'hard', or 'circular'. In order to say that an object is black, they compared it with a crow for example, and to say that there were five objects, they directly compared these objects with a hand." [Aleksandrov 1963]

Unfortunately, while early civilizations outgrew (and children likewise outgrow) their misconceptions of number objects, the same is not true for the "objectification" or abstraction of processes. Menninger reports a similar historical difficulty in developing the notion of arithmetic function. He reports that there is a noticeable absence of symbolic representations for arithmetic operations despite the development of symbolic representations for quantities. The concepts were functionally utilized but not formally represented: "The idea that a purely abstract mark on paper can represent a change or alternation of some kind does not, it seems, come at all easily". In present day education this reluctance towards the symbolic recognition of "active entities" is reinforced in many basic ways, including language:

"...you might posit the class noun as those words that can be used to identify the basic type of object." [Allen 1987] One pertinent study of interviews with Argentinean children aged 4-6 who had not previously experienced written language revealed that the subjects intuitively believed that nouns could be described in written words (e.g.. the word "Daddy" in "Daddy kicks the ball."), but the same was not true for verbs (e.g. the word "kicks" in the same sentence) ([Ferreiro 1978], sited in [Hughes 1986]).

Recently, many math educators have begun focusing their research on the concept of function. Dubinsky and colleagues have postulated an epistemology of functions [Breidenbach, Dubinsky et al. 1992]. According to the theory, development of the concept of function occurs in three phases:

> 1. Action: the ability to plug numbers into an algebraic expression and calculate.
>
> 2. Process: dynamic transformation of quantities according to some repeatable means.
>
> 3. Object: the ability to perform actions on and transform the function itself. [Harel 1992]

Several studies have targeted computer programming models as aids in the development from action to process concept of function [Breidenbach, Dubinsky et al. 1992; Ayer, Davis et al. 1993; Cuoco 1993a; Cuoco 1993b]. Little work has been done, however, to trace the development from process to object concept of function. In one of the only such scenarios, a 14 year old child participated in a 12 week study where researchers attempted to use a computer environment (with which the child was already proficient) to examine the student's development from process to object concept of function [Kieran, Garaicon et al. 1993]. The study reports that the child did not acquire an object concept of function.

Based on Chi's theories of conceptual change [Chi, Slotta et al. 1994], students' misunderstandings about the object nature of function are not surprising. Chi

proposes that all entities in the world can be classified in one of three categories: Matter, Processes, and Mental States. She notes that "misconceptions that cross ontological boundaries can occur in various scientific disciplines at various levels of analysis" which is consistent with the uniform errors that were observed from students of both mathematics and computer science. Chi proposes an "incompatibility theory" to account for why students have trouble with certain science concepts: it lies in the difference between the categorical representation that students bring to an instructional context and the ontological category to which the science concept truly belongs. The more difficult entities to learn about are those that simultaneously embodies more than one category and hence require conceptual alternation , e.g. matter and process (or, if you will, object and function).

To summarize, the development of children's concept of number seems to parallel the evolution of the concept of number within the human species. The impetus behind both the understanding of the concept of number and function lies in the ability to mentally manipulate abstract data types as objects.

6.3 Abstraction and Mental Imagery

The key to unlocking problems with higher-order function lies in mastery of abstraction. Anna Sfard puts it too well to rephrase in her paper "Operational Origins of Mathematical Objects and the Quandary of Reification -- The Case of Function"[Sfard 1992]:

> *The Ideal mathematician*, according to Davis and Hersh (1983, p. 35), *studies objects whose existence is unsuspected by all except a handful of his fellows.* Even the strangest abstract entities, while scrutinized and manipulated, seem to the mathematician as unquestionably real as the pen with which he or she writes papers...

l . s u ı ılatively positive [Engelkamp and Zi n

> ...Shortly after being introduced to [the notion of function, the student] is expected to analyze and manipulate the new entity with a confidence which can only be achieved by those who can treat it as if it were a real thing. Many of our students, however, seem to be lacking this ability. Recent studies on learning mathematics abound in findings which can serve as evidence.
>
> Awareness of the long and painful process preceding the birth of a mathematical object may be the key to understanding some of the difficulties experienced by so many learners.

The difference, as noted by Sfard above, between an expert and a novice mathematician lies in the level of comfort with abstract objects. Indeed, the key to expertise rests in the ability to mentally manipulate abstract objects *as if they were real objects*. Quite simply put, the task of the imaging curriculum is to build concrete manipulatives which act as stepping stones, carrying the student from novice to expert abstracter.

The math education research of Dubinsky *et al.* cite Piaget's theory of *reflective abstraction* [Piaget 1966] which connects concrete activity with the development of abstract concepts as the roots of the present day epistemological analysis. The idea of this work is that the ability to mentally manipulate abstract data objects is directly related to one's expertise in mental imagery. The notion of using mental imagery in this fashion can be traced back far before Piaget to Plato: the Socratic method and the theory of forms are both philosophical examples stressing imagery of mathematical (and other) concepts [Plato 1974]. In his 1944 work *The Psychology of Invention in the Mathematical Field*, Jacques Hadamard [Hadamard 1944] wrote about introspective evaluation methods for invention in the mathematical field. He encouraged the use of concrete representations - images, drawings or linguistic artifacts - to catalyze the generation of logical and intuitive imagery.

It has been the task of this work to enhance that imagery ability and to justify the feasibility of this approach; thus we turn towards a more contemporary summary of some of the important empirical results in the study of mental imagery.

Imagery researcher David Marks claims that while the *ability* to generate and employ mental imagery varies across people, the *potential* to do so is universal. Results of a study on picture-memory task retrieval indicated that "poor" visualizers produced 36% more errors than good visualizers. Ability to visualize, Marks asserts, depends on situational variables which can be manipulated to enhance imagery ability [Marks 1972]. This is supported by [Marschark 1988] who claim that mediation (metacognitive instruction) is useful in fostering children's imaging ability. According to Marks, imagery is useful as an associative retrieval aid between different but related stimuli [Marks 1990].

In studies linking imagery and problem solving, Kaufmann found that imagery is most important in the initial phases of the task; with increasing familiarity, a purely linguistic representation was favorable. Novelty, complexity, and ambiguity all seem to place additional cognitive demands on a task [Kaufmann 1988]. In a second study, Kaufmann linked imagery specifically to the discovery stage of problem solving: utility of imagery increased systematically as the level of programming (familiarity) in the task to be solved decreased [Kaufmann 1990] [Kaufmann 1988]. Engelkamp found that the effect of added exposure to the recall stimulus (planning, executing, visualizing, *and* verbalizing as opposed to planning, executing, visualizing *or* verbalizing) is cumulatively positive [Engelkamp and Zimmer 1990]. [24]

[24] This supports the imaging curriculum philosophy of exposure to multiple imaging representations (computational, visual, motor).

In a study of imagery and unexpected learning, subjects instructed to image and not to learn were unable to prevent learning [Sheehan 1972]. Research has conclusively correlated imaging ability to high performance on memory tasks involving concrete stimuli [Paivio 1971]. Paivio also discovered that "high" imagers reacted significantly more quickly than low imagers when the stimulus words were abstract. He concludes that abstract language does not evoke imagery as readily as concrete. In another study, results indicate that differences in recall between concrete and abstract stimuli are reduced or even eliminated when rich contextual factors operate [Paivio 1988].

More recently, additional research on representation and action words has positively linked performance and motor imagery (imaging performance) to efficient verb encoding and comprehension [Engelkamp 1988]. This study found that verbs were learned much less easily than nouns under standard instruction (63 vs. 72 percent) but that this difference was eliminated when the learning included motor encoding (acting) (77 vs. 79 percent). They further discovered that elaboration (acting more, longer, or differently) had no effect on the initial success, indicating that motor encoding had a particularly strong initial effect which would not be easily improved with further processing.

To summarize, the above research concludes that imaging is positively linked to learning. Although less inherently developed for abstract tasks, this is nevertheless the domain in which imagery is most strongly needed. Imagery is particularly useful in the initial (novel) stages of problem solving. Motor imagery has also been shown to be a particularly strong method of encoding.

It would be negligent to leave this topic without acknowledging that the debate over mental imagery is far from resolved. The raging debate, tightly summarized in

Michael Tye's *The Imagery Debate*, is focused mainly on the exact *nature* of mental imagery. The issue, according to Tye, is representational: what is the composition of an image? He outlines a set of competing theories on the nature of imagery, including:

- the picture theory--mental images are pictures, similar to projections on a computer screen [Kosslyn 1980]
- structural descriptionalism--mental images are complex linguistic representations [Pylyshyn 1981]
- array theory--mental images are interpreted, symbol filled arrays [Tye 1991]

While the debate about representation rages on, the single denominator among theories seems to be the notion that, whatever its nature, imagery does exist as an introspective phenomenon. Work on imaging curricula does not rely on the resolution of the imagery debate: the empirical results summarized above are sufficiently promising indicators of the utility of imagery as a tool for learning.

6.4 Manipulatives and Education

Recall the words of Hilbert as introduced in the first pages of this text:

> "In mathematics... we find two tendencies present. On the one hand, the tendency toward *abstraction* seeks to crystallize the *logical* relations inherent in the maze of material that is being studied.... On the other hand, the tendency toward *intuitive understanding* fosters a more immediate grasp of the objects one studies, a live *rapport* with them, so to speak, which stresses the concrete meaning of their relations...." [Hilbert and Cohn-Vossen 1932]

This quote draws together the all facets of the problem to be solved. In the imaging curriculum, the embodiment of all the research, both historical and present, is the not the idea, but rather the fact, of the physical manipulative.

In this century, the idea of using concrete manipulatives was popularized through the Montessori school of learning [Montessori 1956]. Montessori's theories of "direct adaptation to the adult world" were born out of her observations of children's fascination with task repetition: "Thereafter, I set out to find experimental objects that would make this concentration possible, and carefully worked out an environment that would present the most favorable external conditions for this concentration."

The Montessori tradition has been carried on through the use of Cuisienaire rods [March 1977] for basic math education. Children use colored wooden rods, each of which uniquely represented a number 1-9, to learn simple algebraic calculations. The rods perform the same task for number that we hope the felt pillows achieve for functions: they allow the learner to make a smooth transition from expert manipulator of concrete objects to expert manipulator of abstract data objects.

This work uses not just physical manipulatives but a combination of multi-modal tools to evoke imagery. In another multi-modal study involving basic math skills, 130 different experiments were performed on young children using a combination of abacus, Cuisienaire rods, graphs, and Dienes logical blocks [Frederique 1971]. This study concludes in a section on reasoning and concrete imagery that:

> "The child's spontaneous tendency to live side by side with imaginary beings quite naturally leads to his ability to reason with the help of concrete imagery, a fundamental process in every science. In our teaching we show when seeing reveals or raises a point; we use manipulation when touch suggests, teaches or consolidates an idea. " [Frederique 1971]

Thyer and Maggs further reflect on the state of basic math education by pointing out that "it is now generally accepted that for mathematics teaching, surer foundations are laid when a child's thinking is closely linked with perceptual experiences acquired by doing things." [Thyer and Maggs 1971]

To summarize, recent history in the theory of education has placed some emphasis on developing links between physical learning tools and abstract concepts. In this educational paradigm, tangible objects help children to develop mental familiarity with a concept by bridging the gap between real and abstract artifacts. Manipulation of concrete objects attempts to address the issue pointed to by Locke in his *An Essay Concerning Human Understanding*: "Abstract ideas are not so obvious or easy to children or the yet unexercised mind as particular ones. If they seem so to grown men it is only because by constant and familiar use they are made so [Locke 1894]." Exercising the mind by linking ideas and objects hints at a suitable training program for the novice abstracter.

7 Research Projections

7.1 Methodological Analysis

Chapter 5 presented various perspectives for analyzing the learning which took place for students of the imaging curriculum. It is equally prudent to note what has been learned from the researcher's perspective. In particular, we would like to reflect on the overall utility of the various methods which were used for purposes of data collection.

Admittedly, the statistical data based on students' absolute success and failure in the curriculum is a useful set of results. However, it provides only one of many perspectives for analysis. This work is better interpreted not on the basis of any statistical argument but rather as a framework for a set of pedagogical issues. It enlightens the community of educators by illuminating a set of potentially superfluous (in view of their demonstrated resolvability) pedagogical problems. In its most humble interpretation, this work is useful as an existence proof: that one can design a curriculum with which to present especially important and difficult material to younger children.

7.2 The Power of Function

The idea of functions as data objects is a powerful and robust one which deserves more attention as a generally necessary and useful concept. As the history of mathematics points out, FD is a concept whose time has come: the current status of function can be compared to the early transition to an abstract number system.

Functions represent the next generation of higher order thinking. According to Aleksandrov:

> "Summing it up, it is possible to say that while elementary mathematics deals with constant magnitudes, and the next period with variable magnitudes, *contemporary mathematics is the mathematics of all possible (in general, variable) quantitative relations and interdependencies among magnitudes.*" [Aleksandrov 1963]

Aleksandrov's quote illuminates the importance of functions as data objects to the field of mathematics as a whole; this work raises the issue of precisely when such topics can and should be introduced into mathematics curricula. Potentially, having a cohesive ontology of the FD concept at an early age could translate to less confusion over concepts of higher math and science as they were introduced further on in a student's educational career.

The issue, then, boils down to a question of exactly what makes higher mathematics "higher". This work demonstrates that children as young as 10 years old are capable of comprehending higher order functions. If, as Aleksandrov alludes to, this is a cornerstone issue in the future of mathematics, shouldn't students begin practice with this topic as early as possible? Consider the analogy with the concept of number (cf. Menninger): 5,000 years ago, the concept of abstract numbers could hardly have been considered an appropriate topic for anyone but the intellectual elite. Today, the concept of number is the domain of a five year old. Extrapolating, based on analogy with the number system, the concept of function is perhaps destined to become equally as commonplace in the future of mathematics education.

7.3 Extending the Imaging Curriculum

This work has endeavored to provide a convincing argument for the utility of imaging as a pedagogical aid; it developed a curriculum template for doing so, founded on a

rich pool of research from different disciplines; it executed the curriculum, successfully, on the particular target concept of FD.

One future direction for this work might be to fit the curriculum to another abstract problem area and confirm its extendibility. Within mathematics alone, there are a multitude of candidates for study including complex numbers and *n*-dimensional spaces. Within the domain of physics, there is already much research indicating that the key to maturation from novice to expert lies in one's powers of abstraction. Larkin [Larkin 1983], for example, describes the problem representation of novice physicists as a naive composition of objects that exist in the real world (blocks, pulleys, springs) and characterizes experts by their ability to manipulate purely imagined (abstract) entities such as force and momentum. Here again is a situation in need of an explicit link between the concrete and abstract problem solution: the idea that not only can we manipulate blocks, pulleys and strings, but also we can manipulate as mental objects the abstract concepts, such as force and momentum, that apply to them.

Ultimately, this work should lead to a theoretically founded craft of manipulative design. To begin this process, there are many general questions which can be asked of the current work, for example: (1) does the 50% failure rate of the imaging curriculum reflect the efficacy of imaging as a whole or of this particular design; (2) are there universal design principles that can be extracted from this experience; (3) can any conclusions be drawn about the trainability of imagery?

In terms of the application described in this work -- that of imaging applied to functional data -- there are several specific questions to be asked, for example:

(1) How do motivational issues affect the results? Recall that all the unsuccessful students of the curriculum demonstrated a marked lack of interest. Should it be the task of such a curriculum to motivate the student?
(2) How important are a variety of tools in influencing imaging? Since the curriculum always used three types of imaging tools (computational, visual, motor) it could not support or refute the power of any one tool to enhance imagery. Sorting out the dynamics of this interaction seems like logical next step.
(3) How do issues of age and gender (section 5.4) influence the effectiveness of a tool? Results indicate that these two factors played a role, yet it would seem extremely inefficient to have to design a unique set of manipulatives for each age and gender group.

Further, any intelligent design should be capable of elegant extension. In other words, as theories and audiences change and grow, so should their pedagogical tools. This introduces the notion of *progressively scaffolded images*: developing a hierarchical set of tools. Collins, Brown and Holum have noted that apprenticeship of a novice learner often involves the expert performing portions of the task that the student is as yet unprepared to independently complete. This is referred to as *scaffolding*. *Fading*, then is the act of slowly removing the expert's support and gradually giving the apprentice more and more responsibility [Collins, Brown et al. 1991]. One could imagine analogously applying these concepts to the imaging curriculum: once a certain level of cognitive expertise is attained the user could abandon a primitive set of imaging tools in favor of a more advanced, yet consistent, set of tools. For example, in the imaging curriculum for functional data, one might want to develop another set of tools which were consistent with Ableson's and Sussman's environment model and which were ready for use after "graduation" from the basic object images that were already provided.

7.4 Codifying Orders of Abstraction

In the course of this research the notion of abstraction has blossomed into the critical issue. This work occasionally mentioned the idea of "orders of abstraction": that is,

some facets of an abstraction problem may be quantitatively more difficult than others. Classifying these differences could be the key to a new level of understanding about many of the problems that students encounter in any number of complicated higher learning tasks. To begin this process, we can examine the problems within the domain of functional programming which have been pinpointed by this study.

Recall that some younger, more inexperienced programmers hit their first major impediment when the idea of adding a parameter to a function definition was introduced (Maureen and Betty: section 2.2.2.3; Laurence and Jonathan: section 5.1.6). Recall also that `lambda` expressions present problems for programmers, specifically in the form of the missing variable phenomenon (Hector: section 2.2.3; Aaron: section 4.2; [Eisenberg, Resnick et al. 1987]). Lastly, half of students in an undergraduate programming language class were unable to answer the `derivative-maker` question on a homework assignment (Table 4). Based on these observations, Figure 15 shows a number of related Scheme problems organized in a rough hierarchy. Although it seems intuitively obvious that these problems are organized in an increasing order of complexity, extracting exactly why this is the case seems both critical and non-trivial.

Following up on this specific example from functional programming, consider the differences in abstraction between `slope` and `slope-maker` functions in Figure 15. We might classify the `slope` function as *abstraction order 1* because it requires that we wait for one abstraction unit before it can be fully specified. In other words, when this procedure is defined, it is missing one piece of information, `f`, which will be specified in a later stage of the processing, when we actually call the `slope` function

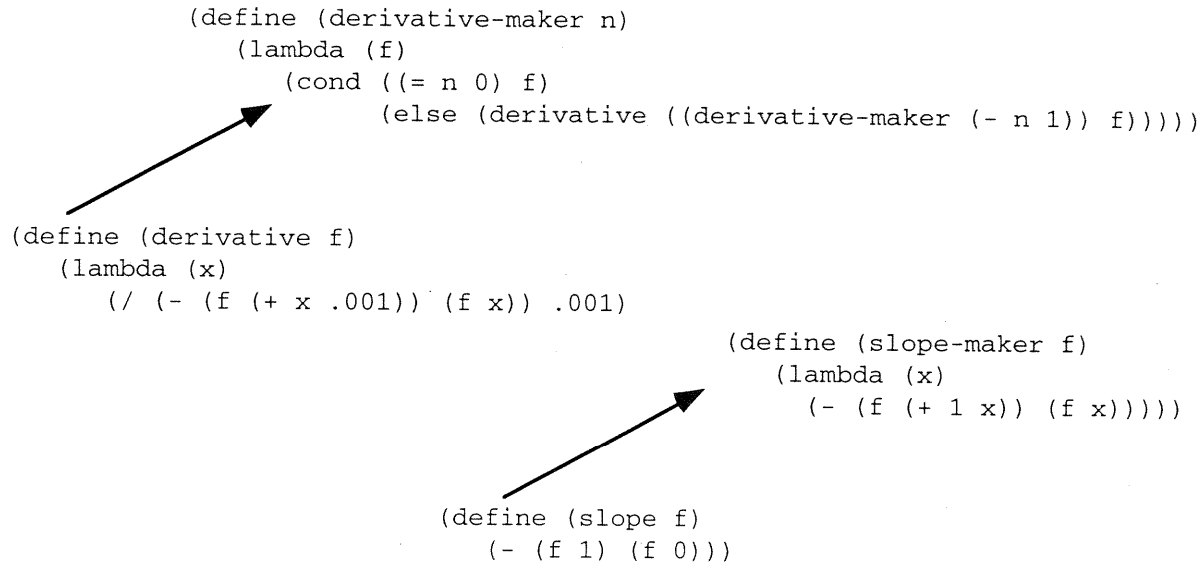


Figure 15: Orders of Abstraction
A rough abstraction hierarchy. In the lower example, the `slope` function is considered less abstract than the `slope-maker` function. Slope takes a single argument, `f`, specified at runtime. `Slope-maker` also takes as its only argument, but its output is also a function of one argument, `x`. In that unbound variables remain even after an initial evaluation, `slope-maker` is considered more complex than the `slope` function. The `derivative-maker` example is even more abstract. In this case, the result of a call to `derivative-maker` is a recursive property 2 function. The introduction of recursion seems to increase the complexity of this problem over its `derivative` ancestor.

with a specific functional parameter, e.g. `double`. The `slope-maker` function, then, would be classified as *abstraction order 2*. In this case, after the same amount of processing as was performed in the `slope` function (calling `slope` with the `f` argument specified) the result is a function which is still missing some specification, the numerical argument `x`. The process is not completed and fully specified until another abstraction unit has expired and the result procedure (e.g. the result of `(slope-maker double)`) is executed with a value for `x` inserted, e.g. `5`. An elaboration of the same argument could be used to explain why the `derivative-maker` example might be classified as *abstraction order 3*.

Note that we decided that the abstraction process for the `slope-maker` function had "bottomed-out" when we specified the numerical argument, `x`. Drawing again on what is known about the history of the formation of the abstract number concept, one might argue that the numerical value of `x` is *not* in fact the ultimate specification of data; rather, the number `5` is itself a generalization of all the sets of things whose cardinality is five.

Yet, on a day-to-day basis, we take the number `5` for granted: it has been completely internalized as a "thing in itself." There is a culture around numbers which trains their users to consider them as ordinary. By a very young age, for example, children have already had enough exposure to abstract numbers as to have reified them. *Abstract numbers are an ever present part of our culture.* Again, this was not always the case; for example, the ancient Greeks had no concept of zero. Today, ruminations about zero are relegated to Sesame Street. The concept of zero is no longer problematic because it is deeply embedded in our culture. Just as the abstraction of numbers has been demoted to trivial status over the centuries via cultural immersion, so might higher orders of abstraction be someday part of our cultural fabric.

It seemed that the level of abstraction presented by a Scheme expression related to when its unbound variables would be specified. However, this relationship may not be as simple as is suggested by the examples in Figure 16. A number of other factors may influence the perceived difficulty of the problem -- for example, the overall number of variables in an expression. Is an "abstraction order 1" function of six variables more problematic than the same function with only one parameter? How might adding multiple unbound variables to the result function in an "abstraction order 2" function influence its perception (e.g., adding more parameters to the `lambda` expression in the `derivative` function from Figure 16)? Even the

classification of the examples in Figure 16 is not clear cut. For example, how important is the inheritance of information from a workable problem at "abstraction order *x*" to the conception of a problem at "abstraction order *x+1*". In other words, given a student who has a complete understanding of either the `slope` or the `derivative` function, is it equally difficult to progress from this point to a complete understanding of the `derivative-maker` function?

This discussion certainly does not capture all that there is to be said about the notion of abstraction. However, it does indicate that there is more to be explored. Functional programming -- specifically, the generalization of data -- has provided but one specific example of how difficulties with abstraction can be stratified. Future research might probe for similar patterns within other domains.

7.5 Assessment Mechanisms

Mental imagery, however it may be represented, is an introspectively verifiable phenomenon. Recall that this work used several methods to assess the utility of the curriculum for enhancing learning: (1) student self-reports, (2) imaging "fingerprinting", and (3) written questionnaires. It verified that students were able to take their knowledge from one domain and apply it to another; it observed that students "graduated" from using concrete manipulatives to the independent and comfortable application of FD.

In this study, success was assessed based on transfer of skill from the domain of graphics programming to the domain of mathematical programming. Ideally, one would like to address not just the *content* of what was learned but also the *techniques* that were learned. So instead of studying students' aptitude for functional data across

two related domains using identical sets of tools, this study might have analyzed students' ability to draw on imagery skills to understand a completely novel area of abstraction.

The notion of imaging ability itself was also left as an unnamed variable in the current study. No significant attempt was made to gauge a student's predisposition for imaging prior to their beginning the study. For this reason, it is not clear whether this study targets those students with the most talent for imaging or those with the most difficulty. A student like Aaron might be tagged as a "good" imager and therefore without need for such a curriculum. Students like Patty, David, and Joseph may have been "bad" imagers who were failed by a curriculum which did not sufficiently inspire them. A more plausible explanation would be to conjecture than the majority of the students -- the ones who were in fact served by the imaging curriculum -- possessed a "standard" ability to image which was well toned by participation in this work.

In future studies, it would be possible to more precisely answer these questions by pre-testing students for imaging aptitude. There have been many different models proposed for classifying learning styles [Johassen and Grabowski 1993]. As an example, consider the Gregorc Style Delineator which measures on a bidimensional scale: concrete-abstract and random-sequential. Gregorc proposes that anyone can possess skills in any of the four manners defined by the scale. Subjects in an imaging curriculum might be tested before and after participating in the curriculum to determine if their aptitude in one or the other of Gregorc's manners had been altered.

7.6 Implications for the Functional Programming Community

Minimally, the results of this study should attract the attention of the functional programming community. For years, proponents of functional programming have seemed frustrated and confused about lack of acceptance and acknowledgment (of the greatness of functional programming) within the larger computer science community. Backus, an avid fan of functional programming, calls conventional programming languages "fat and flabby" [Backus 1978]. Yet, functional programming has never gained the momentum its supporters feel it deserves. This study of functions as objects can perhaps provide some pointers into the nature of this problem.

The novice programmer's standard introduction to functional programming is rife with contradiction and lacking familiarity. While it might not be viable to introduce something like a manipulative-based curriculum at the university level, other options for reifying the notion of functional objects include better interface support. A graphical user interface could succinctly capture the spirit of, at least, the visual imagery tools used in this curriculum therefore making comprehensive functional programming more accessible to students. Applications like STk [Gallesio] (an x-windows interface to Scheme which allows graphical display of built in data structures) are in fact pioneering this much needed movement. The shortcomings of such a move are the same as the shortcomings of the imaging curriculum for non-novice programmers, like Aaron. In general, however, the *novice* functional programmer should be well-served by such a trend.

On a similar note, submerging the learning of functional programming within a non-standard setting may be another useful method of introduction. In the study at hand, users were able to write independently interesting code in the domain of graphics. If

a user is immersed in a particular problem area of interest, the difficulties presented by functional programming languages -- indeed, by *any* programming language paradigm -- may be more easily overcome. This was so in the current study: SchemePaint was used to interest the users in the graphics as well as the pure programming. Additionally, the combination of programming and direct-manipulation [Eisenberg 1991] provided another invitation for the novice programmer.

Issues of interface also call into question how much of an effect learning one particular functional programming language might have over another. ML [Milner 1978] is a functional programming language whose power is roughly identical to that of Scheme. However, Scheme and ML differ in one potentially key area: ML, in contrast to Scheme, is strongly typed. An example of the standard `(apply-to-5 create-subtracter)` example is included in Table 8 for comparison:

Table 8: A Comparison of Scheme and ML.
The prototype create-subtracter example from this study of Scheme is presented alongside its ML counterpart. For some students, the strongly-typed nature of ML may make it syntactically and semantically richer for supporting the concept of functional objects.

Scheme: `(define (create-sub n) (lambda (x) (- x n)))`	**ML:** `fun create_sub(n) = let fun f(x:int) = x-n in f end`
Scheme: `(define (apply-to-5 f) (f 5))`	**ML:** `fun apply_to_5(f) = f(5);`
Scheme: `(apply-to-5 create-sub)`	**ML:** `apply_to_5(create_sub);`

Recall that students of Scheme complained about the inconsistency in syntax between defining different types of objects in Scheme. In addition to correcting the syntactic difference, the curriculum began to move towards a "pseudo-typing" system by using

declaration language of the form `make-<type>-object`. Although one might praise ML for its use of strong types there are several issues to be resolved still:

1. Merely announcing types still does nothing to encourage an object concept of function. For example, an imperative programming language like Pascal is typed but procedures are not first-class objects.

2. There are still syntactic inconsistencies between functions and other objects in ML. For example, for syntactic reasons, the symbol `fun` cannot appear to the right of an `=` sign. Hence, as in the above example, it is necessary to use the `let/in/end` environment to define a function the result of which is also a function. This is awkward.

3. One *could* argue that reasoning from types is a good way to solve problems like the `(apply-to-5 create-subtracter)` example. In the original study, [Eisenberg, Resnick et al. 1987], some of the students who answered correctly explained in an interview that they had reasoned from types. Yet, there is a sense in which this type of reasoning averts the underlying issues touched upon by functional data. *It is certainly possible to correctly reason from types while still holding a naive concept of function.*

Another issue worth exploring is the role of arguments in the concept of function. Recall from this research that young students initially had difficulty with the general notion of parameterization. Note also that the Eisenberg [Eisenberg, Resnick et al. 1987] study observed that some students who failed on `(apply-to-5 create-subtracter)` were in fact able to complete a similar problem of the form: `((apply-to-5 create-subtracter) 3)`. The study notes that reasoning about the expression was much easier once all of the "parts" were present. This would seem to refute the utility of pure functional system (e.g. FP, proposed in [Backus 1978]) where functions are, in a sense, "maximally curried" and non-functional data is transient. A system like FP creates more issues about the status of objects as well. In the imaging curriculum, we are trying to encourage a uniform concept of data. FP creates additional unjustified status differences among data objects. For example, FP would transform an inherently binary function, like addition, into a function applied to a single argument function. FP would necessarily imply unequal status of the two

addends based on their order.[25] On the other hand, to understand even the simplest of processes in FP (e.g. adding two numbers) is to understand that functions are objects (in virtue of the closure that is created by this process).

In summary, it is unclear from the current study how the interaction of factors like typing and argument structure might affect the learning process. These topics are certainly worthy of future exploration.

7.7 Towards an Intuitive Learning Movement

Ideally, the student of the imaging curriculum will carry with her the meta-cognitive skills [Yussen 1985] of visualization and abstraction through a lifetime of learning. It is the claim of this work that these "higher-order" thinking skills are far more pedagogically and practically advantageous to the lifetime learner than any disconnected collection of facts or theories. Ultimately, those skilled in thought have a larger window on learning than those who are simply skilled.

Unfortunately, the transfer of thinking skills between situations is uncommon. Research has shown that learning is fairly context dependent [Lave and Wenger 1991] and immune to meta-cognitive skill transfer. As with the acquisition of any other skill, imaging ability cannot be woven into the fiber of cognition without extensive practice; Benoit Mandelbrot acknowledges the need for cognitive exercise as it regards to pictorial images: "Intuition is not something that is given. I've trained my intuition to accept as obvious shapes which were initially rejected as absurd, and I find everyone else can do the same." (quoted in [Gleick 1987])

[25] The Scheme expression `(+ 4 5)` would be solved in FP by first creating an "add-4" closure and then applying it to 5.

If imaging tools are introduced early enough and frequently enough into the standard pre-college curriculum we can conjecture that learners *will* get better at knowing how and when to use their skills. The work that has been described here is a step toward intertwining a greater volume and variety of imaging tools into K-12 education: with careful and persistent attention to visualization techniques, students can learn to bridge the gap between concrete and abstract thinking.

References

Abelson, H. and G. J. Sussman (1985). *Structure and Interpretation of Computer Programs.* Cambridge, MA, The MIT Press.

Aleksandrov, K. &. L., Ed. (1963). *Mathematics: Its Content, Methods, and Meaning*. Cambridge, MIT Press.

Allen, J. (1987). *Natural Language Understanding*. Menlo Park, Benjamin-Cummings.

Ayer, T., G. Davis, et al. (1993). Computer Experiences in Learning Composition of Functions. Clarkson University.

Babbage, C. (1842). Notes on the Analytical Engine. *Charles Babbage and His Calculating Engines: Selected Writings by Charles Babbage and Others*. New York, Dover Publications, Inc. 225-295.

Backus, J. (1978). "Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs." *Communications of the ACM* **21**(August): 613-641.

Breidenbach, D., E. Dubinsky, et al. (1992). "Development of the Process Conception of Function." *Educational Studies in Mathematics* **23**: 247-285.

Cajori, F. (1928). *A History of Mathematical Notation*. La Salle, Illinois, Open Court.

Chi, M. T. H., J. D. Slotta, et al. (1994). "From Things to Processes: A Theory of Conceptual Change for Learning Science Concepts." *Learning and Instruction* **4**: 27-43.

Collins, A., J. S. Brown, et al. (1991). "Cognitive Apprenticeship: Making Thinking Visible." *American Educator* (Winter):

Cuoco, A. (1993a). Computational Media to Support the Learning and Use of Functions. Education Development Center.

Cuoco, A. (1993b). Constructing Functions from Algebra Word Problems. Education Development Center.

diSessa, A. A. (1993). Images in Learning. University of California at Berkeley.

Eisenberg, M. (1990). *Programming in MacScheme*. San Francisco, CA, The Scientific Press.

Eisenberg, M. (1991). Programmable Applications: Interpreter Meets Interface.

Eisenberg, M., M. Resnick, et al. (1987). Understanding Procedures as Objects. *Empirical Studies of Programmers: Second Workshop*. 14-32.

Engelkamp, J. (1988). Images and Actions. *Cognitive and Neuropsychological Approaches to Mental Imagery*. Dordrecht, Martinus Nijhoff.

Engelkamp, J. and H. Zimmer (1990). Imagery and Action: Differential Encoding of Verbs and Nouns. *Imagery: Current Developments*. London, Routledge.

Ferreiro, E. (1978). "What is in a Written Sentence? A Developmental Answer." *Journal of Education* **160**: 25-39.

Frederique (1971). *Mathematics and the Child*. Montreal, Algonquin.

Gallesio, E. STk Reference Manual. Universite de Nice.

Gardner, H. (1993). *Multiple Intelligences : the Theory in Practice*. New York, Basic Books.

Gentner, D. and A. Stevens (1983). *Mental Models*. Hillsdale, NJ, Lawrence Erlbaum Associates.

Gleick, J. (1987). *Chaos: Making a New Science*. New York, Penguin.

Gödel, K. (1931). *On Formally Undecidable Propositions of Principia Mathematica and Related Systems*. New York, Dover.

Hadamard, J. (1944). *The Psychology of Invention in the Mathematical Field*. New York, Dover Publications.

Harel, E. D. a. G. (1992). The Nature of the Process Concept of Function. *The Concept of Function*. Mathmatical Association of America.

Harel, G. and E. Dubinsky, Ed. (1992). *The Concept of Function: Aspects of Epistemology and Pedagogy*. Mathematical Association of America Notes and Reports Series.

Hilbert, D. and S. Cohn-Vossen (1932). *Geometry and the Imagination*. New York, Chelsea Publishing Company.

Hughes, M. (1986). *Children and Number: Difficulties in Learning Mathematics*. Oxford, Basil Blackwell.

Johassen, D. H. and B. L. Grabowski (1993). *Handbook of Individual Differences Learning and Instruction*. Hillsdale, Laurence Erlbaum.

Kaufmann, G. (1988). Mental Imagery and Problem Solving. *Cognitive and Neuropsychological Approaches to Mental Imagery*. Dordrecht, Martinus Nijhoff.

Kaufmann, G. (1988). Mental Imagery in Problem Solving. *International Review of Mental Imagery*. New York, Human Sciences Press.

Kaufmann, G. (1990). Imagery Effects on Problem Solving. *Imagery: Current Developments*. London, Routledge.

Kieran, C., M. Garaicon, et al. (1993). *Technology in the Learning of Functions: Process to Object*? Psychology of Mathematics Education, Asilomar Conference Center,

Klawe, M. (1995). *E-GEMS: Electronic Games for Education in Math and Science*. National Educational Computing Conference, Baltimore,

Koschmann, T. (1990). *The Common LISP Companion*. New York, Wiley.

Kosslyn, S. (1980). *Image and Mind*. Cambridge, MA, Harvard University Press.

Larkin, J. H. (1983). The Role of Problem Representation in Physics. *Mental Models*. Hillsdale, NJ, Lawrence Erlbaum Associates.

Lave, J. and E. Wenger (1991). *Situated Learning : Legitimate Peripheral Participation*. Cambridge, Cambridge University Press.

Locke, J. (1894). *An Essay Concerning Human Understanding*. Oxford, Clarendon Press.

Lovelace, A. (1842). Notes on the Analytical Engine. *Charles Babbage and His Calculating Engines: Selected Writings by Charles Babbage and Others*. New York, Dover Publications, Inc. 225-295.

March, R. (1977). "Georges Cuisienaire and his Rainbow Rods." *Learning* **6**(3): 85-86.

Marks, D. F. (1972). Individual differences in the vividness of visual imagery and their effect on function. *The Function and Nature of Imagery*. New York, Academic Press.

Marks, D. F. (1990). On the relationship between imagery, body, and mind. *Imagery: Current Developments*. London, Routledge.

Marschark, J. C. Y. a. M. (1988). Imagery and Children's Learning. *International Review of Mental Imagery*. New York, Human Sciences Press.

Menninger, K. (1969). *Number Words and Number Symbols: A Cultural History of Numbers*. New York, Dover Publications.

Milner, R. (1978). "A Theory of Type Polymorphism in Programming." *Journal of Computer and Science Systems* **17**(3): 348-375.

Montessori, M. (1956). *The Child in The Family*. New York, Avon Books.

Paivio, A. (1971). Imagery and Language. *Imagery: Current Cognitive Approaches*. New York, Academic Press.

Paivio, A. (1988). Basic Puzzles In Imagery Research. *Cognitive and Neuropsychological Approaches to Mental Imagery*. Dordrecht, Martinus Nijhoff.

Papert, S. (1980). *Mindstorms: Children, Computers and Powerful Ideas*. New York, Basic Books.

Piaget, E. W. B. a. J. (1966). *Mathematical epistemology and psychology*. New York, Gordon & Breach.

Polya, G. (1945). *How to Solve it*? Princeton, NJ, PUP.

Pylyshyn, Z. (1981). Imagery and Artificial Intelligence. *Readings in the Philosophy of Psychology*. Cambridge, MA, Harvard University Press.

Randell, B., Ed. (1973). *The Origins of Digital Computers: Selected Papers*. Berlin, Springer-Verlag.

Sethi, R. (1989). *Programming Languages: Concepts and Constructs*. New York, Addison-Wesley.

Sfard, A. (1992). The Case of Function. *The Concept of Function.* Mathmatical Association of America.

Sheehan, P. (1972). Imagery and Unexpected Recall. *The Function and Nature of Imagery*. New York, Academic Press.

Sierpinska, A. (1992). On Understanding the Notion of Function. *The Concept of Function.* Mathmatical Association of America.

Soloway, E. and K. Ehrlich (1984). "Empirical Studies of Programming Knowledge." *IEEE Transactions on Software Engineering* **SE-10**(5): 595-609.

Stoy, J. S. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. Cambridge, MIT Press.

Thomas, G. B. and R. C. Finney (1993). *Calculus and Analytic Geometry*. Reading, Addison Wesley.

Thyer, D. and J. Maggs (1971). *Teaching Mathematics to Young Children*. London, Holt, Reinhart and Winston.

Touretzky, D. (1990). *Common LISP: A Gentle Introduction to Symbolic Computation*. Redwood City, Benjamin/Cummings.

Turing, A. (1937). *On Computable Numbers, with an Application to the Entscheidungsproblem.* Proceedings of the London Mathematical Society,

Turing, A. M. (1950). "Computing Machinery and Intelligence." *Mind* **59**: 433-460.

Tye, M. (1991). *The Imagery Debate*. Cambridge, MIT Press.

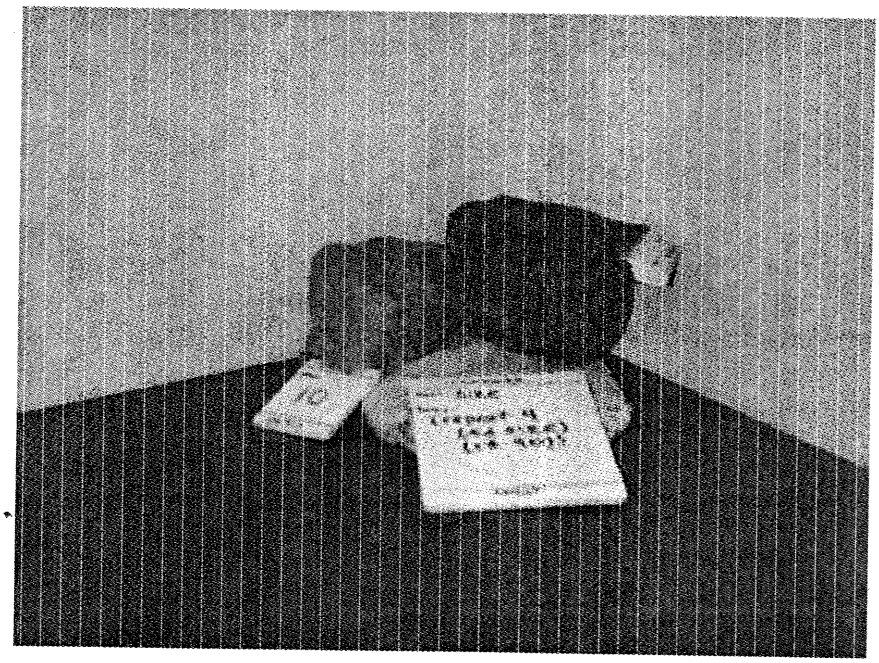
Von-Neumann, J., Ed. (1945). *First Draft of a Report on the EDVAC*. The Origins of Digital Computers: Selected Papers. New York, Springer-Verlag.

Wilensky, R. (1986). *Common LISPcraft*. New York, Norton.

Winston, P. H. and B. K. P. Horn (1989). *LISP*. New York, Addison-Wesley.

Yussen, S. R. (1985). The Role of Metacognition in Contemporary Theories of Cognitive Development. *Metacognition, Cognition, and Human Performance: Volume 1 -- Theoretical Perspectives*. New York, Academic Press.

Appendix A: Manipulatives



Appendix B: Aaron's First Test

The following pages contain a copy of the actual test completed by Aaron. The test was administered after Aaron completed four sessions in the curriculum. He had previously worked with Levels I-III in the domain of graphics only.

It would be posssible to write a procedure which squared numbers:

```
>>> (define square
          (make-procedure
               (x)
               (* x x)))

>>> (square 5)
25
```

And likewise, we could cube numbers:

```
>>> (define cube
          (make-procedure
               (x)
               (* (* x x) x)))

>>> (cube 2)
8
```

We could go on like this and write a procedure which raised numbers to the 4th power, and another for the 5th power, and so on. Or, we could save some time by writing a procedure which generated any power procedure that we wanted. So we would use it like this:

```
>>> (define cube (make-power-procedure 3))

>>> (cube 2)
8
```

Write make-power procedure.

```
(define make-power-procedure
  (make-procedure
    (exp)
    (make-procedure
      (x)
      (expt x exp))))
```

Give some examples of ways that you would use it. For example, how can I take the 10th power of the number 3 without ever defining a 10th power procedure into the table?

```
>>> ((make-power-procedure 10) 3)

>>> ((make-power-procedure 2) 4)
16

>>> (define power-proc (make-power-procedure))

>>> ((power-proc 2) 4)
16
```

```
>>> (define subtract-from-5
      (make-procedure (x) (- 5 x)))
```

What does this procedure do?

It decreases the number 5 by an input (x),

Give an example of how you would use it and what the result would be of your example.

```
>>> (subtract-from-5 10)
    -5
```

```
>>> (define apply-to-5
      (make-procedure
           (f)
           (f 5)))
```

What does this procedure do?

It takes a procedure (f) as an input and inputs 5 to that procedure.

Give some example uses and results.

```
>>> (apply-to-5 (subtract-from-5))
    0
```

```
>>> (apply-to-5 (make-power-of
      3))
    125
```

```
>>> (define make-doubler
      (make-procedure
           ()
           (make-procedure (x) (* 2 x))))
```

Then what would happen if we typed in the following expressions:

```
>>> ((make-doubler) 5)
    10
```

```
>>> (make-doubler)
    <PROCEDURE>
```

```
>>> make-doubler
    ?
```

subtracter

```
>>> (define create-subtracter
        (make-procedure
            (n)
            (make-procedure (x) (- x n))))
```

What does create subtracter do? What does it take as arguments and what does it return?

Create subtractor makes a procedure that outputs "subtract n from me". It takes the argument n. It returns a procedure

sorry →

What would be the result of the following expression:

```
>>> (create-subtracter 3)
```

<PROCEDURE>

What do the following expressions do?

```
>>> (define subtract-3 (create-subtracter 3))
```

Defines subtract-3 as a program that subtracts 3 from numbers. →

```
>>> (subtract-3 10)
```

Subtracts 3 from 10.
Outputs 7

What happens when we call apply-to-5 on create-subtracter?

```
>>> (apply-to-5 create-subtracter)
```

Creates a program that subtracts 5 from numbers.

Appendix C: Interview with Aaron

```
(define six-figure
      (make-procedure
            (shape size)
            (repeat 6
                  (shape size)
                  (rt 60))))
```

Interviewer: Ok, so here's a procedure I've defined called six-figure. Can you tell me what it does?
Respondent A: *Looks like it, uh, makes, um, these shapes six time. Sorta like my little ppm thing.*

I: Here in this procedure what are we passing in for arguments?
A: *Um, shape I think is gonna be... its gonna define the right turn, what direction to turn and stuff and size... wait... shape is gonna be the whole shape that you are going to repeat and size is going to be how much to spread it out.*

I: So what kind of an argument is size and what kind of an argument is shape in terms of type?
A: *Shape is going to be a procedure and size is going to be a number.*

I: If I am telling you that we are passing in to this procedure, as a piece of data - we're passing in another procedure. If you can put any sort of words to any sort of pictures that might get conjured up in your head when you think about the idea of passing a procedure as an input to another procedure. Do you get any sort of picture in your head for what it means to do that. You should just talk freely. This is free association.
A: *Uh, um. No really. Just sort of something inside of something.*

I: Something inside of something else?
A: *Like a box inside of a box.*

I: Does it seem like there is a difference between the two arguments in this procedure? Size and shape - are they different in nature? Are they the same? How are they different? Does it seem intuitively obvious in both cases that what we are doing is reasonable and logical? That's a lot of questions. Let's start with: does it seem the same to pass in a number as an argument to a procedure vs. to pass in a procedure as an argument to a procedure?
A: *Well just passing in a procedure, well, when you're passing in just like an argument its straightforward and passing in a procedure it just seems like you have to look through more stuff. Its like with the box inside of the box analogy the procedure would be a box that would contain more stuff and a number would just be something that can't go any smaller, that's already there.*

I: So in the case of the procedure its more complicated 'cause there's more stuff to unpack once you actually get into running the procedure. That seems like a reasonable analogy.

Do you have any kind of imagery around the idea of a procedure object? Like when you think of using or making procedure objects do you visualize anything in your head? Do you see them in any way? You talked about boxes: do you see them as boxes or are there other ways that you see them? Do you think you call that image up when you are programming at all?
A: *Well, I think of like what its gonna look like on the screen when its done but I don't think of analogies when I'm putting together the exact words.*

I: Do you see procedure objects as single units or are they things that have parts?
A: *Different parts - stuff inside of a container.*

I: I've been using two basic modes of description for all different kinds of objects. One is when I write on the board and we use the name/object association table to describe number objects and procedure objects and color objects. And the other thing is sometimes we use these other objects to talk about generating new procedure objects or passing things around, setting up a bunch of arguments to get ready to run a procedure. My question for you is do you think that either of these representations has been useful in helping you learn this stuff and if so, how?
A: *Seeing it up on the board is helpful. I mean, those are helpful its just, seeing it all out there is the most helpful for me. Its kinda like how I think is I visualize stuff.*

I: So, you think that the visualization of the table is a good thing?
A: *uh huh.*

I: And you think visualization of the physical objects - has that helped in any way? What different things - at what times might those conjure up a different image that helps you do something else? Are there different times where different imagery is useful?
A: *I think having it on the board was the most useful because... I think putting it inside of the little pouches and stuff its harder for me to see how it goes together and how it works.*

I: Sort of like "the big picture"?
A: *Yah.*

I: Its interesting though because you talked about things being containers and I'm wondering if that's at all because of the fact that we used these and they are inherently sort of containers for information. Do you think maybe this helped you come up with a more high level sort of image and then maybe when you need to get down into the nitty gritty of what's going on that [board] is better for conjuring up lower level programming kind of stuff?

A: *Could be. Even in BASIC I always thought of subroutines as blocks.*
I: Blocks, like physical blocks or blocks like blocks of code, blocks of text?
A: *Well, both -*

I: - So, right now, you're motioning with your hands. You're making a motion with your hands as if it was a physical, cubical box. So that's how you visualize subroutines in BASIC?
A: *Yah, well, I tell, like the big box to go get the little box.*

I: And you think you've always visualized that way - before we ever did any stuff here you always had this idea of BASIC subroutines being boxes? I'm noting again with the tape that he's motioning with his hands!

I: OK, when we first went over all of the background stuff about the table and such we talked about computer "space" - this is what's going on inside of the computer the computer world. So sort of in a very philosophical sense I want to ask you, do you think that the world that we talk about when we talk about the computer, is that a real world? We make some assumption that our world, the world we live in is real; is the computer world real?
A: *Its real its not alive. It doesn't really have space.*

I: It doesn't have space, so then its not a physical thing. Would it be more of an abstract thing?
A: *I don't know... not really.*

I: There are things that we encounter in our everyday world that are very concrete like the tape recorder or the telephone. Then there are abstract things like thoughts and ideas. So, what I'm trying to get you to do is classify the world of computation as being more along the concrete side or more along the abstract side.
A: *Uh, huh. I don't know its physically there - the electric stuff, but that's so easily gotten rid of. I don't know I guess maybe there's more concrete.*

```
(define color-square-maker
      (make-procedure
            (color)
            (make-procedure
                  (size)
                  (setmapc color)
                  (square size))))
```

I: OK, I've just written a new procedure on the board. I've defined a new procedure called color-square-maker. And, the idea of color-square-maker is that you say, well actually, I won't say any more. Take a second to look over this and tell me what it does and how you would use it.
A: *OK. Hmm. I want to look back at like ppm and compose but, OK. I guess you input color and it makes a square of that color, defined size. You would enter, wait color, and then it would make a procedure of the square of that size. So I think you would enter the command; you'd have to put the color.*

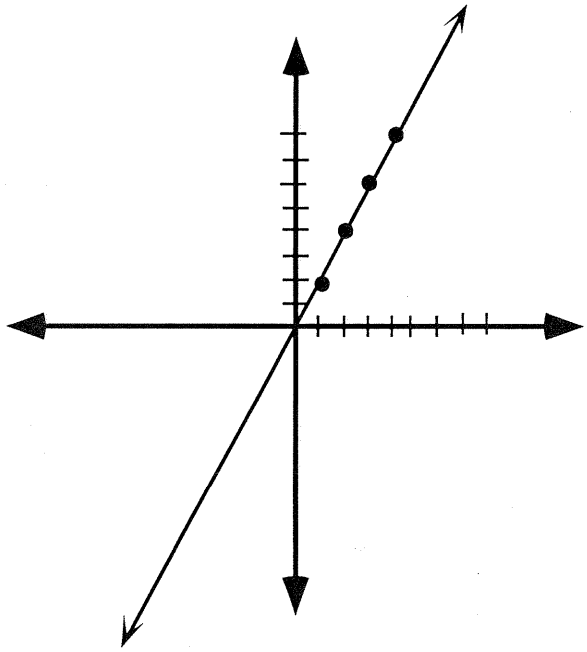
Appendix D : Slope Worksheet

Say we have the following functions which operate on numbers:

```
>>> (double 2)
4
>>> (triple 2)
6
```

1. Write `double` and `triple`.

Remember from math that we can represent functions as graphs. So, `double` could also be written y=2x or f(x) = 2x. We could graph it as follows:



Recall also that the slope of a line refers to how much the function changes value over a given interval. So, between x=0 and x=1, the function's slope is f(1) - f(0), or 2-0, or 2.

2. What is the slope between x=1 and x=2?
between x=2 and x=3?

Different functions may have different slopes.

3. What is the slope of the `double` function?
of the `triple` function?

We would like to write a procedure which lets us evaluate slope for different functions. It would work like this:

```
>>> (define double-slope-function (slope double))
>>> (double-slope-function 0)
2
>>> (double-slope-function 1)
2
```

The interval used to calculate the slope is defined by the input and the input+1. [so, `(double-slope-function 2)` calculates between 2 and 3]

4. Write `slope`.

5. Use slope on the `double` and `triple` functions. Predict what the result should be first.

6. Write a `quadruple` procedure. What should its slope be?

7. Write a `square` procedure which returns the 2nd power of numbers. [`(square 3) = 9`, for example]

8. Use your slope function to make a chart for `square`. Do you see any patterns.

x-interval	slope
0 1	
1 2	
2 3	
3 4	

9. Write a `cube` function and make a similar chart. Try to predict the results beforehand.