Examing the Difficulty with Thinking of Functions as Data Objects: Misconceptions of Higher Order Functions

Julie DiBiase

CU-CS-791-95

MISCONCEPTIONS OF HIGHER ORDER FUNCTION

Pedagogical accounts in mathematics and computer science education suggest that students experience unusual difficulty in learning the concept of higher order function. This work lends credence to such folklore through three investigatory protocols. First, traditional age students (college) are tested on their concept of function. Next, this document recounts interviews with experts in a variety of related disciplines. Lastly, young students (aged 10-14) are given exposure to higher order functions through a computer programming based curriculum; two case-study excerpts are included. From these investigations a systematic scheme of errors emerges. Results are collated to form high-level conceptual models of function. It is suggested that these results should be used to direct a revised pedagogy in the teaching of functions.

INTRODUCTION

```
>>> (define (double x)      >>> (define (apply-to-5 f)
      (* x 2))                        (f 5))
```

The above examples present two seemingly similar expressions in the Scheme programming language. 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 have difficulty with the `apply-to-5` function (DiBiase, 1995; Eisenberg, Resnick, & Turbak, 1987). 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 which has plagued students and educators of functional programming was the original motivation for these investigations.

Functional data objects provide a robust and elegant means for expressing ideas in many mathematics related disciplines. Unfortunately, educators in these areas report that students uniformly experience serious pedagogical problems with this material. The remainder of this work examines the exact nature of the misconceptions which motivate students' difficulties. To further frame the problem, the next section begins by defining the issue and providing some historical perspectives. Following that I will present the results of some more recent related research. The bulk of this work will report on a number of investigatory studies about students' concept of function; it will conclude with a discussion of the results and their implications for educators.

BACKGROUND

Definition of Higher-Order Function

Informally, the main idea examined by this work is that *functions*[1] *are manipulable as units of data.* This concept, central to the whole functional paradigm of computer programming, can be traced back far into the historical foundations of computer science; for example, it is

[1]Although strictly speaking the terms "function" and "procedure" have a subtle semantic distinction in functional programming, they are herein used interchangeably and indistinguishably. This representation (versus alternative represenations, e.g. graphs or sets) is most appropriate in the turtle-graphics environment used in this study.

at the heart of Turing's universal machine concept (Turing, 1937). As will be discussed, this research has indicated that students consistently encounter difficulty with the notion of functions as data objects (FD).

Under a more formal definition of FD (from (Stoy, 1977) on denotational semantics) functions have the following four properties, common to all "first-class" data objects:

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

Differentiation is an example from basic calculus which embodies properties 1-3 above. Consider the following example:

$$f(x) = 3x^2$$
$$f'(x) = 6x$$

In this example, there are 3 functions:

- `f` is the function which takes one argument, a number, `x`, and returns a number, the result of squaring `x` and multiplying it by `3`.
- `f'` is the function which takes one argument, a number, `x`, and returns a number, the result of multiplying `x` by `6`.
- `'` is the function which takes one argument, a function, `f`, and returns a function, the result of differentiation. In other words:

```
'(f(x)) = f'(x)
```

This example represents the essence of what it means for functions to have object, or "first-class" status, that is, the ability of a function (as in the derivative function) to act on another procedure (as in the function $3x^2$ above) as an argument and return a new function as its result.

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

[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.

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>
```

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% answered incorrectly as to the outcome of the expression, responding that the expression was not fully specified:

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

The remainder of this report 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. The problem is first staged with some historical perspectives, followed by an initial analysis of students' misconceptions and educators perceptions thereof. The heart of this work lies in the final results section: a detailed case study analysis examining the genesis of younger students' concept of function.

History of the Problem

The problem in understanding functional data is as old as the history of computer science. In 1842, Ada Augusta, Countess of Lovelace, considered the nature of the theoretical machine then being proposed by Charles Babbage:

"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)

By clearly demarcating operations and data, Lovelace exhibited the same naive understanding about operations (functions) which we will see in present day students of function. Even Babbage himself admitted the importance of this issue while at the same time acknowledging that, for him, it presented an unconquerable challenge (Babbage, 1842).

A century after Babbage's initial musing about a self-modifying engine, Turing outlined the specifications for a machine which uses the specification of some other machine as data (Turing, 1937). Turing's ideas were in fact an elaboration of the same function-versus-object theme that is identifiable in Gödel's technique of representing proof sequences as numerals (Gödel, 1931). By 1945, Von Neumann envisioned the machine which would use calculation specifications (functions) as input. In Von Neumann's system -- the underlying model still used in computational designs today -- functions take on the same form as any other data object (Von-Neumann, 1945).

In present day computer science, functions and procedures provide the underlying organizational structure for all programming languages. 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)

Functions are the very core of computer science as well as mathematics(Thomas & Finney, 1993) (Cajori, 1928) and physics (Aleksandrov, Kolmogorov, & Lavrent'ev, 1963). This brief argument has hopefully confirmed that functional data is a is stimulating, perplexing and important concept to great as well as novice thinkers.

The Difficulty with Function

Although clearly an *important* concept, history informs us that functions were likewise perceived as notably *difficult*. This instinctual problem is perpetuated by current day computer 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 cannot be directly abstracted in the same way that numbers can indicating, in some sense, a different status (Winston & Horn, 1989).

A recent anthology published by the Mathematical Association of America dedicated itself entirely to examining problems with the concept of function (Harel & Dubinsky, 1992) including (Sierpinska, 1992) citing 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.

Insights into the concept of functional objects can be gained by probing mathematics' most common element: 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)

The challenges identified above relative to the genesis of the number system are parallel to the problems we currently see from students of function. Early civilizations 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, et al., 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. 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" (Menninger, 1969). 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, Hawks, & Nichols, 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. (Dubinsky & Harel, 1992)

Several studies have targeted computer programming models as aids in the development from action to process concept of function (Ayer, Davis, Dubinsky, & Lewin, 1993; Breidenbach, et al., 1992; Cuoco, 1993; Cuoco, 1995) (Cuoco, 1993; Cuoco, 1995). 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 subject participated in a 12 week study in which researchers attempted to use a computer environment (with which the student was already proficient) to examine the student's development from process to object concept of mathematical function (Kieran, Garaicon, Lee, & Boileau, 1993). The study reports that the subject did not acquire an object concept of function.

Based on Chi's theories of conceptual change (Chi, Slotta, & deLeeuw, 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. 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 embody more than one category and hence require conceptual alternation, e.g. matter and process (or, conceivably, 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 (Sfard, 1992). The following section describes three types of experiments which were performed to empirically verify the notorious difficulties that students experience with higher-order function.

RESULTS

Study 1: University Students of Functional Programming

Two distinct studies were performed using graduate and undergraduate students at the University of Colorado. Subjects ranged in experience from those who had recently been introduced to the Scheme programming language to those who had just a minimal knowledge of its syntax. This experiment was intended to study the problems and misconceptions (as alluded to by instructors) of students in a standard curriculum.

Study 1A

The first study was performed on a sample of eleven students from three different but comparable backgrounds. They were students who had 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 respectively relates to Properties 1-3 of first-class objects (cf. Stoy). Individual responses are paraphrased where interesting.

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.[4]

TABLE 1: 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

<u>Example 1:</u>

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
function	5
"error: need parenthesis"	1

Total wrong: 55% (6/11)

<u>Example 2:</u>

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

Asked: `>>> (apply-to-5 -1+)`

Answered:	
"4"	11

Total wrong: 0% (0/11)

<u>Example 3:</u>

Given:
```
>>> (define (create-subtracter n) (lambda (x) (- x n)))
```

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

Answered:	
function	4
"error: no argument"	2
"error: need 'x' to complete `lambda(x)`"	1
"error: no second function call for `lambda(x)`"	1
"error: `apply-to-5` not defined"	1
no answer	2

Total wrong: 64% (7/11)

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

In the second example of this study, all students answered correctly about the use of a functional argument to a function (Property 2).

The final example gave students the definition of a function which returned a function as its result. They were asked to predict the result of an expression that both takes a function as an argument and 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.

Study 1B

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 students attended two 1.25 hour introductory Scheme lectures which specifically emphasized the object nature of function and presented illustrative examples. Tables 2-4 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.

Example 2 of Table 2 exemplifies Stoy's property 2: functions can be arguments to other functions. Its asks that students write a function to "solve" two other functions.

TABLE 2: PROGRAMMING LANGUAGE 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 a procedure, `solutions`, which, given two functions, will return all solutions to those two functions (i.e. points of intersection on a graph) over a range of given values.

Answers:	
correct	12
no use of functional arguments	2
incorrect use of functional arguments	6
no attempt	8

Total wrong: 57% (16/28)

Example 3:

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	6

```
(define (mng (lambda (n) (gne x n)))
```

Total wrong: 46% (13/28)

Although almost all students who tried the problem used functional arguments (this requirement was included in the problem specification), many showed an incorrect use of functions within the body of the procedure:

```
(if = f(n) g(n) ...
```

This type of shift in notation from scheme programming to standard mathematics was not observed in the same students' solutions to other problems in the homework.[5]

In Example 4 (Table 3), students were given an equation for approximating first derivatives; they were asked to write a derivative function in Scheme which, given some function, returned the approximation (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:

```
>>> (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 input and returning functions as results. The Scheme code for programming a derivative function is a direct translation of the approximation given in Example 4 of Table 3:

```
>>> (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 did not think they needed to use lambda (i.e. generate a new function); despite the ease of translation from the mathematical notation to Scheme, seven more students didn't even attempt the problem. In Example 5 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 6 is yet an order of magnitude more difficult than Example 4: 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."

[5]At first glance, it appears that the data from Tables 1 and 2 imply that property 1 is in fact more difficult than property 2; results presented in the next section will show that this is not a correct assessment. The oddity of the results so far stems from two sources: (1) Example 2 of Table 1 was a very simple demonstration of property 2 and most students had already been exposed to an analogous problem, and (2) Example 1 of Table 2 was judged harshly in that the 13 students who gave a correct answer that did emphasize the object nature of function were marked incorrect.

TABLE 3: PROGRAMMING LANGUAGE 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 4:

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 `(define (derivative fun x) ...)`	9
no attempt	7

Total wrong: 64% (16/28)

Example 5:

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 6:

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)

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 6 (Table 4: Examples 7-8). 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. This study hopes to shed light on why a problem like Example 8 scared 22 students away from even attempting it.

TABLE 4: PROGRAMMING LANGUAGE 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 7:

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 8:

Question: Use `function-applied-n-times` to redefine `derivative-maker`.

Answers:	
correct	6
no attempt	22

Total wrong: 79% (22/28)

Summary

- Students are very comfortable with the notion of naming functions.
- Students show some tendency to stray from an object concept of function when they begin passing functions as arguments.
- Students have notable trouble writing functions that create other functions. As users of these functions, students often mistake correct code for erroneous based on the idea that it is "missing" some critical piece of information.

Study 2: Expert Interviews with Instructors of Function

In an attempt to further verify the folklore surrounding functional data, experts from a variety of disciplines were interviewed about how they perceive and purvey the concept. All subjects were University of Colorado instructors in math, physics and computer science who had taught some form of the concept in a graduate or undergraduate classroom setting.

Interview 2A: Artificial Intelligence

Folklore in the functional programming community resoundingly agrees: students have a hard time with the idea that functions can be manipulated as data objects. 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 pedagogical miscommunication between students and instructors of functional objects seems to create frustration from both ends of the learning channel.

Interview 2B: Programming Languages

The instructor from the undergraduate programming language class studied in Experiment 1B was interviewed about the chronology of his personal concept of function and how he has used his own learning experiences to influence his communication with students:

> Coming from an electrical engineering background, I was reluctant to use functions in any ways in which I couldn't directly visualize the resulting assembly code. So how are things different now that I'm an "expert" on these concepts? Understanding the underlying details at an "instinctual" level, I can manipulate the abstractions with confidence, and avoid the need to explicitly "translate" the abstractions to lower level concepts.
>
> Certainly many students have problems when first presented with higher-order functions, just as they have trouble with other abstractions such as type parameterization. The students who are having more difficulty, have great trouble verbalizing their conceptualizations. To them higher-order functions and other abstractions seem to be very mysterious entities, and they attempt to deal with them by copying examples, much like applying formulae by rote. To try to assist students in understanding higher-order functions, I draw on my own experience that mental visualizations to explain the underlying mechanisms of the abstractions are helpful for achieving a comfortable working understanding of the abstractions.

In this case, the professor describes how he has drawn upon the personal methods which he previously employed to unify his own conceptualization of higher-order function in order to assist his students in making a similar cognitive transition.

Interview 2C: Calculus

Among mathematicians, rhetoric about functional data was almost identical to that in computer science. One professor of calculus had the following to say about his personal and professional experience with functions as data objects:

> Interviewer: Did you or do you now have the notion that it's a piece of data?
>
> Mathematician1: *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.

Interview 2D: Group Theory

A professor of group theory (M2) describes personal struggles similar to his students':

> 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[6] was presented to M2, his behavior modeled that of the novice Scheme programmer:

> 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 notes that a variable has been used in the body of the procedure which has not been declared as an argument.

Interview 2E: Physics

In this interview, a physics graduate student and teaching assistant was questioned (P1). She was asked to reflect on the notion of a function as a piece of data from a physicists perspective:

> Well, you know, of course physicists apply functions to functions all the time. The method of substitution is one example of this, where a huge hairy block like *Cos(Sqrt(X-Y^2))* will just get called *Z* to ease calculation. *But* you know what is occurring to me about the way that physicists use functions as objects is that it isn't fully divorced from the understanding of a kind of hierarchy in which objects of functions are not equal in standing to functions. I think what keeps this intact

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

> as an understanding is that though you may toss around functions as objects, every function still has an object, whereas objects don't. I think that this may be an almost ineradicable bias in the mind of a typical physicist, and I might even go so far as to say that it is fundamental and justified... it is jarring and disturbing to see functions without their objects. It feels like an unfinished sentence.

P1 was then asked to describe her personal experiences learning about functions as objects:

> Well, I think that the first introduction I would have had to this idea would have been algebra, and methods like substitution: $X + Y = 12$ but $X = A + B$. That's when you start learning that sometimes there's more than meets the eye. That "objects" like X may have a more complex identity than just something numerical.
>
> In that sense, calculus is not the real beginning of these concepts, but it is certainly a place where that point gets driven home. But I maintain that there is a distinction in the mind, or maybe just my mind, between functional objects and just plain objects. In the first case, there's still something to "put in". I clearly remember having functions taught to me in this way: There is a little black box factory. Something enters the factory from the left and gets spit out on the right, transformed by a relation that was written underneath the factory on the blackboard. Techniques like taking the derivative are seen as making modifications to the factory.

Lastly, P1 comments on the general state of functional objects in physics:

> Let's go back to quantum: there are operators, they operate on functions, these functions may be functions of other functions, and on down the line, but *somewhere* down that line is a plain old object, *3 meters* or *.06 joules* or something like that. This goes back to what I was saying earlier, physicists are happy to treat functions like objects, to operate on functions to produce other functions, but this is not actually a radical equalization between functions and objects.

Summary

- 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.
- Experts describe a personal ontology of function which historically or even still currently supports a naive model of functional objects.
- Instructors report frustration with students' perceived inability to learn these concepts.

Study 3: Case Studies of 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 explore this question, this research studied the unfettered experiences of pre-high school students. 18 case studies were performed in order to judge younger students' impressions of and abilities with functional objects (for a complete account, see (DiBiase, 1995)). Subjects worked one-on-one with an instructor while learning SchemePaint (Eisenberg, 1991), a graphics-enhanced version of Scheme (Eisenberg, Clinger, Harheimer, & Abelson, 1990). Data was collected from the students in two ways:

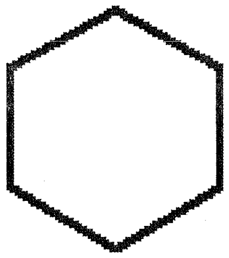
1. Written questionnaires. Students were periodically required to complete questionnaires that tested their acquisition of certain concepts.
2. Session transcripts. SchemePaint allows one to save a computerized transcript of a user's interaction with the interpreter; after each session, researcher observations were integrated with session transcripts.

The following sections present excerpts from two of the 18 case studies: they were selected because they are representative of the consistent errors observed among the majority of students.

Case Study 3A: Brooke

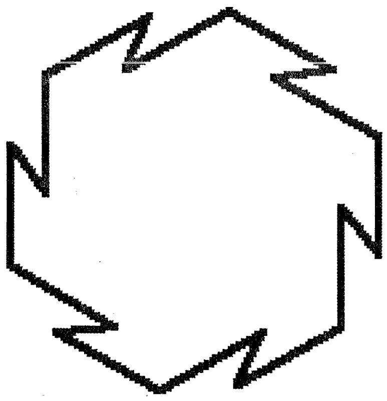
Brooke was a 10 year old female in 5th grade. She had no prior programming experience. One of her first tasks was to write a standard turtle-graphics-like geometric procedure which could make a hexagon:

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



```
   (hexagon 20)
```

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. In this polygon, the standard `hexagon` from above has been permuted through the addition of a functional argument which specifies the movement of the turtle as it draws the sides of the shape (as opposed to the original numerical argument which specified the size of a side). I showed Brooke an example similar to the following figure which shows a six-sided shape with "zigs" for sides:



`(hexagon zig)`

I asked her to write a procedure that could achieve this or any number of other similar effects, keeping the side length constant. Brooke worked out the problem by hand on a dry-erase board. She spent about five minutes deliberating and modifying seeking very little instructor assistance along the way. Below is a transcript of her modification process, with student-teacher interaction included:

STEP 1:

```
line 1 (define (hexagon 20)
line 2     (repeat 6
line 3         (fd 20)
line 4         (rt 60)))
```

[Brooke: "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)))
```

Brooke's initial errors are representative of typical errors of novice programmers dealing with functional arguments. Students have a hard time divorcing the idea of a variable function from the specific instantiation of a function in an example. Above, Brooke was aware that she needed to add a variable (in this case, `pro`) which represented the abstraction of the functional argument to move the turtle along the side of the six-shape. However, she was reluctant to let this abstraction stand alone, in the absence of some instance of function (in this case, `fd`). Instead of a half-way function like Brooke's, many students exhibit even less resolvable behavior by omitting the pro variable entirely re-writing a function to work just for the example argument (e.g. `zig`). Still others define a pro argument to the function, which still using some actual value (e.g. `zig 20`) in line 3 above.

Summary

• Initial observations indicate that students of functional arguments to function have difficulty transitioning from concrete to symbolic representations of function.

Experiment 3B: Hector

The subject of this case study, Hector, was a 7th grade male with some limited programming experience in the C language. He learned basic Scheme in a fairly accelerated fashion. The following description is from the latter part of his work after he has already mastered properties 1 and 2 through repeated exposure to them.

After about ten hours of work with SchemePaint, Hector was informed that his task was to build a "library" of SchemePaint functions which performed manipulations on color-objects. 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. 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). For example:

```
>>> (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 was 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 forgot 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 realized what he did wrong, but it is interesting to note his initial instincts nevertheless.

His next task required a conception of property 3: this was the first time Hector was required to generate such ideas. He was asked to write a procedure called `compose` which, given two transforms (`t1, t2`), creates a new procedure which, when applied to a color-object, returns the result of applying `t1` and `t2` to a given object. The correct solution is:

```
>>> (define (compose t1 t2)
      (lambda (color) (t2 (t1 color))))
```

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."

Summary

• Students have trouble with the notion that a functions as anonymous objects.

• Students have trouble with functions which do not bind all their variables at runtime.

DISCUSSION

Historically and currently, then, 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 (see Example 1: Table 1).

ii. *<u>Property 2 is more difficult.</u>* 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. Early in their computer programming experience, students are able to assign symbolic variable names to numbers; the same is not true of functions (see Experiment 3A: Brooke). As described by P1, functions are not units in and of themselves without arguments. Recall that programming language students reverted to a more mathematical notation that includes the argument inserted in (as opposed to just juxtaposed with) the function (see Example 2: Table 2).

iii. *<u>Property 3 is most difficult.</u>* Lambda expressions, functions which return other functions as result objects, create many problems for students. In addition to the two specific errors relating to missing variables and anonymous objects (Experiment 3B: Hector), lambda expressions also prove difficult because statement semantics are obscure and suggest few fruitful analogies (of what significance is the word *`lambda`* to a student unfamiliar with the calculus of same name?).

The above outlined details of misconception represent pieces of a larger cognitive puzzle. These pieces can be fit together to produce various models of conception (similar in vein to (Chi, et al., 1994)). Many other projects have undertaken the task of specifying mental models, particularly in relation to concepts in the physical sciences (Gentner & Stevens, 1983). This work takes a similar approach to "mental modeling". Figure 1 exemplifies two ontological pictures, borrowed from (Eisenberg, 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 summarizes 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 case, procedures are assigned some special status such that they are not candidates to be data objects.

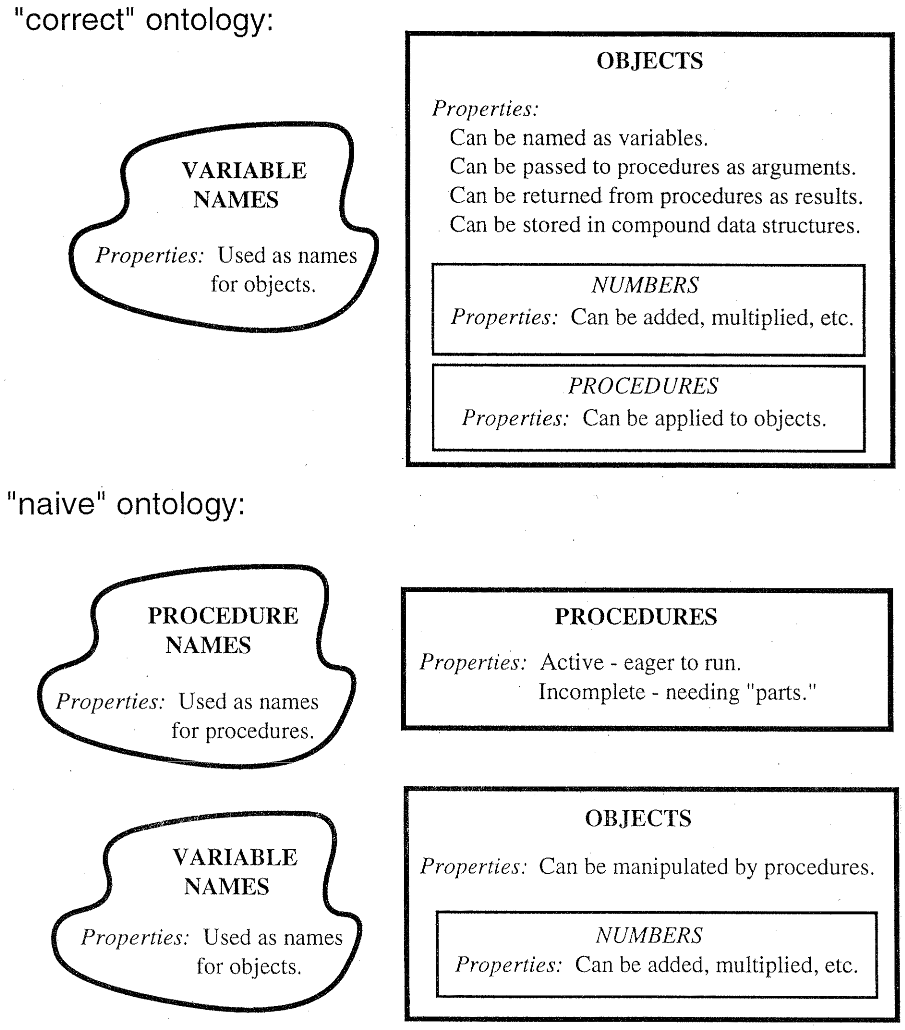

"correct" ontology:
VARIABLE NAMES
Properties: Used as names for objects.
OBJECTS
Properties:
Can be named as variables.
Can be passed to procedures as arguments.
Can be returned from procedures as results.
Can be stored in compound data structures.
NUMBERS
Properties: Can be added, multiplied, etc.
PROCEDURES
Properties: Can be applied to objects.
"naive" ontology:
PROCEDURE NAMES
Properties: Used as names for procedures.
PROCEDURES
Properties: Active - eager to run.
Incomplete - needing "parts."
VARIABLE NAMES
Properties: Used as names for objects.
OBJECTS
Properties: Can be manipulated by procedures.
NUMBERS
Properties: Can be added, multiplied, etc.

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

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, et al., 1994).

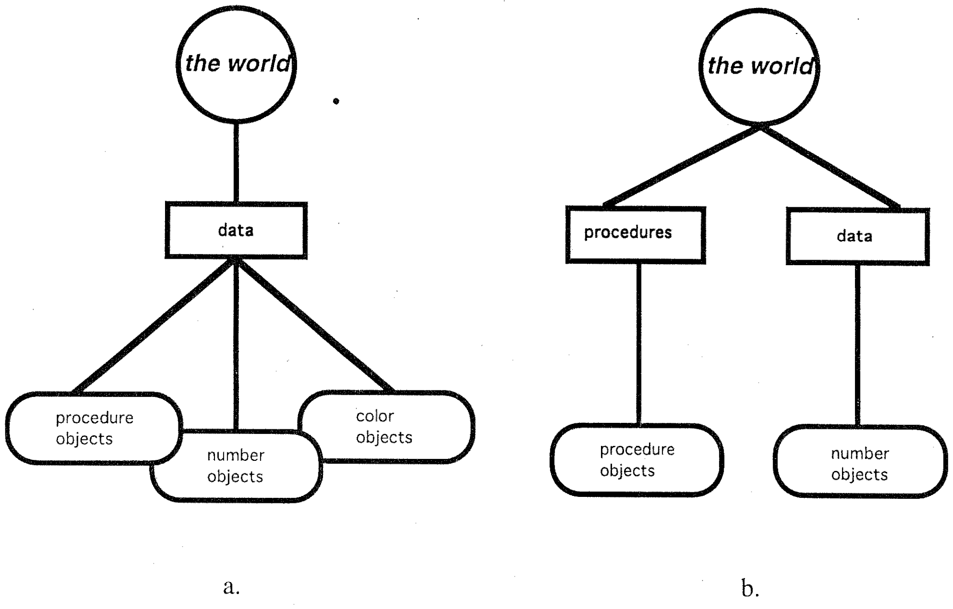


Figure 2: High Level Model of Functional Data
This figure describes the difference in status for functions under a correct (a.), and a naive (b.) cognitive model.

CONCLUDING REMARKS

Minimally, the results of this study should attract the attention of the functional programming and mathematics education communities. For years, proponents of functional programming have seemed frustrated and confused about lack of acceptance of functional programming within the larger computer science community. In a Turing award lecture almost two decades ago, John Backus, an avid fan of functional programming, called 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. Research has indicated that tools which enhance mental imagery in concept formation may assist students of function (DiBiase, 1995) in their transition from process to object (cf. Dubinsky); very recently, there has been some attention to visual language tools which support functional data objects. However, this paper does not advocate any single pedagogical solution. Rather, it is intended as a call to attention for both the functional programming and mathematical communities. The ideas presented in this work, while illuminating a real pedagogical problem, also may open the door for a wider audience to take interest in both disciplines.

As demonstrated, different forms of the functional data problem underlie many important concepts in math, computer science, and physics. It is the hope of this work that the taxonomy of errors which has emerged will cause mathematics and computer science educators to more formally address the inadequate pedagogy around higher-order functions and seek practical solutions.

REFERENCES

Aleksandrov, Aleksandr, Kolmogorov, Andrei, & Lavrent'ev, Mikhail (Ed.). (1963). Mathematics: Its Content, Methods, and Meaning. Cambridge: MIT Press.

Allen, James (1987). Natural Language Understanding. Menlo Park: Benjamin-Cummings.

Ayer, Thomas, Davis, George, Dubinsky, Ed, & Lewin, Philip (1993). Computer Experiences in Learning Composition of Functions No. Clarkson University.

Babbage, Charles (1842). Notes on the Analytical Engine. In P. M. a. E. Morrison (Eds.), Charles Babbage and His Calculating Engines: Selected Writings by Charles Babbage and Others (pp. 225-295). New York: Dover Publications, Inc.

Backus, John (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, Daniel, Dubinsky, Ed, Hawks, Julie, & Nichols, Devilyna (1992). Development of the Process Conception of Function. Educational Studies in Mathematics, 23, 247-285.

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

Chi, Michelene, Slotta, J. D., & deLeeuw, N. (1994). From Things to Processes: A Theory of Conceptual Change for Learning Science Concepts. Learning and Instruction, 4, 27-43.

Cuoco, Al (1993). Constructing Functions from Algebra Word Problems No. Education Development Center.

Cuoco, Al (1995). Computational Media to Support the Learning and Use of Functions. In A. diSessa, C. Hoyles, & R. Noss (Eds.), Computer and Exploratory Learning Berlin: Springer.

DiBiase, Julie (1995) Building Curricula to Shape Cognitive Models: A Case Study of Functions as Data Objects. Ph.D. thesis, University of Colorado, Boulder.

Dubinsky, Ed, & Harel, Guershon (1992). The Nature of the Process Concept of Function. In G. Harel & E. Dubinsky (Eds.), The Concept of Function Mathmatical Association of America.

Eisenberg, Michael (1991). Programmable Applications: Interpreter Meets Interface. SIGCHI Bulletin, 27(2), 68-83.

Eisenberg, Michael, Clinger, Willam, Harheimer, Anne, & Abelson, Harold (1990). Programming in MacScheme. San Francisco, CA: The Scientific Press.

Eisenberg, Michael, Resnick, Mitchel, & Turbak, Franklyn (1987). Understanding Procedures as Objects. In G. M. Olson, S. Sheppard, & E. Soloway (Eds.), Empirical Studies of Programmers: Second Workshop (pp. 14-32). Norwood, NJ: Ablex Publishing Corp.

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

Gentner, Dedre, & Stevens, Al (1983). Mental Models. Hillsdale, NJ: Lawrence Erlbaum Associates.

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

Harel, Guershon, & Dubinsky, Ed (Ed.). (1992). The Concept of Function: Aspects of Epistemology and Pedagogy.

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

Kieran, Carolyn, Garaicon, Maurice, Lee, Lesley, & Boileau, Andre (1993). Technology in the Learning of Functions: Process to Object? In Psychology of Mathematics Education, 15 . Asilomar Conference Center: Monterey.

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

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

Sfard, Anna (1992). The Case of Function. In G. H. a. E. Dubinsky (Eds.), The Concept of Function Mathmatical Association of America.

Sierpinska, Anna (1992). On Understanding the Notion of Function. In G. H. a. E. Dubinsky (Eds.), The Concept of Function Mathmatical Association of America.

Stoy, Joseph S. (1977). Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory. Cambridge: MIT Press.

Thomas, George B., & Finney, Ross C. (1993). Calculus and Analytic Geometry (8th ed.). Reading: Addison Wesley.

Turing, Alan (1937). On Computable Numbers, with an Application to the Entscheidungsproblem. In Proceedings of the London Mathematical Society, 42 (pp. 433-460).

Von-Neumann, Jon (Ed.). (1945). First Draft of a Report on the EDVAC. New York: Springer-Verlag.

Winston, Patrick Henry, & Horn, Berthold Klaus Paul (1989). LISP (3rd ed.). New York: Addison-Wesley.