CHAPTER 2

Understanding Procedures as Objects

Michael Eisenberg
Mitchel Resnick
Franklyn Turbak
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract

Programming languages that treat procedures as "object-like" entities (for example, allowing procedures to be passed as arguments to other procedures) offer major advantages in semantic power and syntactic elegance. In this paper, we examine how novice programmers appropriate the idea of procedures as objects. Based on a series of structured interviews with students in the introductory computer-science course at MIT, we develop a model of the students' ontology of procedures. We conclude that many students view procedures as inherently active entities, with few "object-like" properties. We speculate on the implications of these results for the design and teaching of languages that treat procedures as objects.

1. Introduction

Most programming languages (as well as the courses that teach them) encourage a sharp distinction between procedures and data. In this paradigm, procedures and data are typically viewed, respectively, as actions and objects: just as actions manipulate and transform objects in the physical world, procedures manipulate and transform data in computer programs.

This procedure-data distinction, however, is both artificial and limiting. In fact, procedures are *not* active entities at all: they are merely *descriptions* of processes. As such, procedures *are* data. Moreover, languages that treat procedures as data objects can realize tremendous benefits in semantic power and syntactic elegance.

Scheme, a dialect of Lisp, is a prime example of this approach. In Scheme, procedures are "first-class objects." That is, they have all of the same "object traits" as traditional data objects like numbers, lists, and vectors. For example,

Scheme procedures can be passed as arguments to other procedures, and they can be returned as the results of procedure calls. These first-class properties of procedures make possible simple and elegant implementations of object-oriented programming and delayed evaluation *(1)*, and continuation-passing models of programming *(2)*. Indeed, there is a growing recognition within the computer-science community of the importance of first-class procedures, and the concept is likely to influence the design of future programming languages *(3)*.

In this study, we interviewed a group of students learning Scheme in order to examine how novice programmers think about procedures. Most empirical studies of programmers have focused on syntactic issues and higher-level planning skills. Our aim here is somewhat different: we focus on programmers' semantic models, on their *ontology* of procedures. In particular: what characteristics do novice programmers attribute to procedures, and how does that ontology affect their ability to think of procedures as objects?

We believe that research in this area might provide a theoretical foundation for changes in language design and pedagogy. Only by more fully understanding "naive ontologies" of procedures can we hope to develop improved methods for helping students learn about the use of procedures as first-class objects.

Section 2 of the paper provides a brief background on first-class objects and Scheme procedures. Section 3 describes the methodology of the empirical study. Section 4 uses results from the study to develop a model of the subjects' ontology of procedures. The section also explores this ontology more deeply by focusing on several extended examples from the interviews. Section 5 suggests language-design and pedagogic changes that might help students to learn to think about procedures as objects. Section 6 suggests directions for future work.

2. Background

This section provides a brief discussion of first-class objects and Scheme procedures.[1]

2.1 First-Class Objects

In the semantics of programming languages, certain types of objects are classified as *first-class* [cf. Stoy *(6)*]. A first-class object has the following properties:

1. Variable names may be bound to it;
2. It may be passed as an argument to procedures;
3. It may be returned as the value of a procedure call;

[1] Readers interested in a more thorough treatment of these subjects should refer to *Structure and Interpretation of Computer Programs* by Abelson and Sussman with Sussman *(1)*. This text is used in the MIT introductory computer-science course that our subjects were taking. Eisenberg *(4)* provides a more elementary introduction to this material, while Rees and Clinger *(5)* gives a formal definition of the Scheme language.

4. It may be stored as an element of compound data structures (such as lists or arrays).

Numbers are first-class objects in virtually every programming language. For instance, the following line of Pascal code illustrates the first three properties of first-class objects as they apply to Pascal integers:

```
x := double(3)
```

Here, the `double` function (which we have presumably written earlier) takes an integer as an argument and returns an integer as its result; the value of the call to `double` is now associated with the variable name `x`. We could illustrate the fourth "first-class" property by assigning the result to an array location — e.g., `x[1]` instead of `x`.

2.2 Scheme Procedures

In Scheme, numbers and procedures are both first-class objects.[2] The `define` construct binds a variable name to an object, as in the following examples:[3]

```
(define a 1)
(define b (+ 2 3))
(define stuff (lambda (x) (/ x 3)))
```

In each of these examples, the Scheme interpreter *evaluates* (i.e., finds the object designated by) the rightmost subexpression and binds the specified name to the resulting object. In the first case, the Scheme interpreter evaluates `1` and binds the name `a` to the result (the number 1). In the second example, the interpreter evaluates the subexpression `(+ 2 3)` and binds the variable name `b` to the result (the number 5).

The third example uses the `lambda` construct, Scheme's method for creating a procedure object. In this case, the `lambda` expression creates a procedure that takes one argument and returns the value of that argument divided by 3. The interpreter first evaluates the `lambda` expression, then binds the name `stuff` to the resulting procedure object.

Scheme includes an alternative syntax for defining procedures. The `stuff` definition could also be written as:

```
(define (stuff x) (/ x 3))
```

The association of names with objects is shown in Figure 1. Note that both primitive and compound (i.e., user-defined) procedures are named in exactly the same way as other Scheme objects.

---

[2] Scheme supports other first-class objects, including lists, symbols, booleans, and strings. The examples in our study use only numbers and procedures.

[3] Most of the examples in this section were used in the student interviews. See the Appendix for a listing of all expressions used in the interviews.

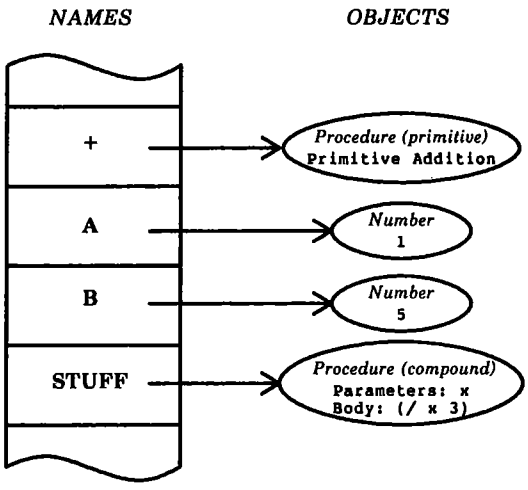

Figure 1: Bindings between names and objects in Scheme.

When the Scheme interpreter evaluates a name, it returns the object bound to that name. For example:[4]

```
a      ⟹   1
b      ⟹   5
stuff  ⟹   [COMPOUND-PROCEDURE STUFF]
```

Procedure calls in Scheme are illustrated by the following example:

```
(stuff b) ⟹ 1.66666
```

The Scheme interpreter first evaluates each of the subexpressions. `Stuff` evaluates to a procedure object – namely, the divide-by-3 procedure created earlier. The other subexpression, `b`, evaluates to the number 5. The interpreter then applies the procedure object to 5, and returns the result.

The above examples demonstrate the first property of first-class procedures – that variable names can be bound to procedures. The second property – that procedures may be passed as arguments to other procedures – is illustrated in the following example:

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

Evaluating this expression binds the name `apply-to-5` to a procedure that takes one argument. When the procedure named by `apply-to-5` is called with a procedure as an argument, the interpreter will apply the argument to the number 5 and return the result:

[4] We use the arrow character "⟹" to mean "evaluates to."

`(apply-to-5 stuff)` $\Longrightarrow$ *1.66666*

`(apply-to-5 (lambda (x) (* x x)))` $\Longrightarrow$ *25*

In the first expression, the argument to `apply-to-5` is the procedure bound to `stuff`. In the second expression, the argument to `apply-to-5` is the "squaring" procedure returned by evaluating the `lambda` expression.

The third property of first-class procedures – that they may be returned as the result of procedure calls – is illustrated by the following example:

```
(define (create-subtracter n)
  (lambda (x) (- x n)))
```

Evaluating this expression binds the name `create-subtracter` to a procedure which, when applied to an object, returns *another* procedure. This new procedure takes one argument and subtracts from it the value of n, where n refers to the original argument to `create-subtracter`. Thus, calling `create-subtracter` with an argument of 1 will return a "decrement" procedure – a procedure that takes one argument and subtracts 1 from it. Similarly, calling `create-subtracter` with an argument of 10 will return a "subtract-10" procedure. Here are some examples:

`(create-subtracter 1)` $\Longrightarrow$ *[COMPOUND-PROCEDURE 23407230]*

`((create-subtracter 1) 5)` $\Longrightarrow$ *4*

`(define increment (create-subtracter -1))` $\Longrightarrow$ *INCREMENT*

`(increment 7)` $\Longrightarrow$ *8*

`(apply-to-5 (create-subtracter 3))` $\Longrightarrow$ *2*

`((apply-to-5 create-subtracter) 3)` $\Longrightarrow$ *-2*

The final property of first-class procedures – namely, that they may be used as elements of compound data structures – was not within the scope of our investigation and will not be discussed further.

As a concluding example, we present a procedure more elaborate than any used in our interview; we include it here to provide a brief illustration of the power of the first-class procedure concept. Our example, `derivative`, takes one argument (a procedure), and returns as its result the procedure corresponding to the derivative of the argument:

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

Programmers with experience in other languages will recognize how difficult it would be to implement `derivative` without first-class procedures. Using the `derivative` procedure is straightforward:

```
(define double (derivative (lambda (x) (* x x)))) ⟹ DOUBLE

(double 5) ⟹ 10.

((derivative double) 5) ⟹ 2.
```

3. Methodology

3.1 Subjects

The subjects in the study were 16 MIT undergraduates enrolled in MIT's introductory computer-science course (6.001). A sign-up sheet was placed in the course laboratory, and students were selected randomly from among those who signed up. All the subjects had some programming experience prior to the course.[5] We conducted the interviews during the fifth and sixth weeks of the semester; first-class procedures had been introduced during the second week of the course. By the time of the interview, all students had completed a laboratory problem set that made heavy use of first-class procedures.

3.2 Task

In each interview, we provided the subject with a written sequence of Scheme expressions. We instructed the subject to evaluate each expression in order and to write his[6] response (as well as any scratch notes) below the expression. The subject was informed that some expressions might result in an error message. We encouraged the subject to talk about the rationale behind his answers. On occasion, we asked additional questions or requested that the subject elaborate on an answer. During the interview, the subject had the opportunity to review and rethink previous answers. We told the subject that answers would not be provided during the interview itself, but offered to provide an informal tutorial at the completion of the interview. The Appendix includes the complete sequence of Scheme expressions used in the interviews.

Two of the authors were present at each interview. Our data for each interview included: a tape recording of the interview, the subject's written responses, and our own written notes. (One tape recording was lost due to equipment failure.)

---

[5] All had programmed in BASIC, and several had programmed in Pascal (6), Fortran (5), C (5), and Logo (3). A few listed other programming languages as well.

[6] We use the masculine pronoun for both male and female subjects.

4. Analysis of Results

In this section, we first present a framework for understanding the subjects' responses. We then present a series of extended examples to support that framework. Finally, we discuss some of the issues raised by our results.

4.1 A Naive Ontology of Procedures

Although student responses varied over a wide range, most subjects seemed to share a common mental map of procedures. This mental map, while consistent among the subjects, is seriously at odds with the semantics of Scheme.

As described in Section 2, Scheme procedures are properly categorized as first-class objects. Of course, each type of object has its own special properties: number objects can be added together, procedure objects can be applied to arguments, and so on. But all first-class objects share the four "first-class properties." Figure 2 illustrates this "correct" ontology of Scheme objects.

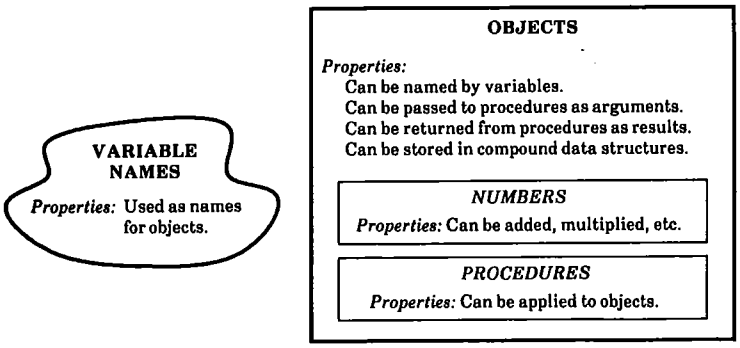

Figure 2: The "correct" ontology for Scheme objects.

Although a few of the subjects viewed procedures in this way, the majority seemed to categorize procedures very differently. Most important, students ascribed to procedures two essential traits:

Activity. The subjects tended to view procedures as the active manipulators of passive data (such as numbers). They associated activity with several aspects of a procedure call: finding and evaluating arguments, working to compute a result, and actually returning the result. Even when procedures were not in the operator position of a procedure call, subjects commonly viewed them not as static data structures, but as bundled up "computational energy" waiting to be unleashed.

Incompleteness. Subjects commonly saw procedures as incomplete entities that needed "additional parts" before they could be successfully used. The missing parts were usually the formal parameters of the procedure or other variables used within the body of the procedure. On occasion, subjects also regarded the parentheses that

specify a Scheme procedure call as parts required to "complete" a procedure. As the examples in Section 4.2 illustrate, subjects often had problems with procedures that they viewed as having too few or too many parts.

We distinguish between these two properties mainly for the sake of exposition. In practice these properties blur together. Indeed, the act of finding and using missing parts is a main component of a procedure's activity. Again and again, subjects referred to a procedure "needing," "wanting," "expecting," "demanding," and "requiring" its arguments. A classic example is Subject 14's statement that he thought of "`lambda` as like a hungry monster that wants food." There are clear traces here of both activity and incompleteness.

The focus on activity and incompleteness seemed to lead students to associate procedures more with the processes they describe rather than with the objects they are. Indeed, many students placed procedures in a category separate from other objects, viewing procedures as a different sort of thing – or, perhaps, not as a "thing" at all.

We can gain some insight into what the students thought procedures *are* by looking at what they said procedures are *not*: they are *not* objects, *not* values, *not* variables. Consider the following quotes, each made by a different subject:

> Junk is a bound variable, it's not a procedure. [Subject 3]
>
> B has a value rather than being a procedure. [Subject 9]
>
> A is not a procedure, it is just a thing. [Subject 10]
>
> What-not is really not a procedure – it's just a name, a variable name. [Subject 13]
>
> I often get tripped up whether these are procedures or variables [Subject 15]

Statements like these could be slips of the tongue, or simply loose use of terminology. But such comments were so common in the interviews that we believe they provide evidence of a faulty model of procedures, a model in which procedures and their names are fundamentally different from other objects and the variables that name them.[7]

Even subjects who had some sense of procedures as objects tended to view them as less "object-like" than numbers. One subject [Subject 5], for example, described an "abstraction" hierarchy among objects. In his view, numbers are the "most primitive objects." Accordingly, he described `sub-1` (bound to a procedure) as "much more of an abstraction" than `b` (which he saw as bound to the expression `(+ 2 3)`). In turn, `b` was more abstract than `a` (bound to `1`), since "you're setting [a] actually equal to a value."

The salient features of the students' naive ontology are captured in Figure 3. The figure depicts procedures as active, incomplete entities that comprise a class distinct from other Scheme objects.

The naive ontology in Figure 3 reflects the subjects' reliance on functional, as opposed to structural, models *(7)*. That is, the subjects were much more concerned

[7] As mentioned before, procedures *are* different than other objects in some ways. For example, procedures, unlike numbers, can be applied to arguments. But most subjects see the differences as more fundamental.

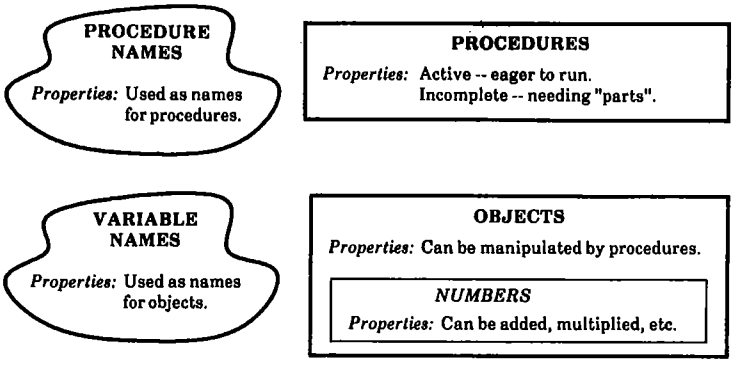

Figure 3: A "naive" ontology for Scheme objects.

with what procedures *do* or how they are *used*, rather than with what they are or the mechanism by which they work. While functional models can be a powerful approach for reasoning about programs, the examples below show how they can lead to serious misconceptions in the absence of a robust structural understanding.

4.2 Extended Examples

This section probes the naive ontology more deeply by focusing on several specific expressions from the interviews.

4.2.1 Decrement

Early in the interviews, we defined `b` and `decrement` as follows:

```
(define b (+ 2 3))
(define (decrement x) (- x 1))
```

Then we asked subjects to evaluate three expressions:

```
(decrement b)
b
decrement
```

All 16 students gave the correct response (namely, 4) for `(decrement b)`, and 15 students gave the correct response (5) for `b`.[8] But subjects had more trouble evaluating `decrement` alone. Four subjects – at least as their first response[9] – said that `decrement` would return an error (when, in fact, it returns a procedure object, with printed representation `[COMPOUND-PROCEDURE DECREMENT]`).

[8] One student fell into the trap of thinking that evaluation of `(decrement b)` would change the value of `b`.

[9] One subject later changed his answer.

The responses of these four students exhibit many aspects of the naive ontology. All four attributed the error to a lack of arguments for `decrement`; they interpreted `decrement` as the name of a procedure that must be applied, rather than the name of an object that can be returned. The notions of "activity" and "incompleteness" were evident in their reasoning. Subject 13, for example, explained: "[`Decrement`] has no arguments to evaluate, nothing to evaluate." Subject 1 used similar reasoning: "You're just calling `decrement` and you're not giving the arguments with it . . . because `decrement` is a procedure which requires an argument."

Significantly, several students who gave the *correct* answer for the evaluation of `decrement` revealed in their explanations that they, too, viewed the evaluation of `decrement` as fundamentally different from the evaluation of `b`. In fact, many of them did not see the question as a matter of evaluating `decrement` at all.

As Subject 15 explained: "You haven't asked it to evaluate [`decrement`]. You've just asked it about `decrement`, you haven't asked it to evaluate it or anything." Similarly, Subject 5 explained: "It doesn't return a value, it just returns that it is a procedure. So this doesn't evaluate an expression. It more just confirms the fact that it is a procedure."

In their laboratory experience with the Scheme interpreter, these students had no doubt seen that evaluating a "naked" procedure name like `decrement` gives a printed result of the form `[COMPOUND-PROCEDURE` *name*`]`. But since they did not view procedures as objects, they did not see this phenomenon as the evaluation of a name. Rather than modifying their mental map of procedures to explain this behavior, they simply inferred that the interpreter had a special-case rule for handling naked procedure names — something like, "Inform the user that this is the name of a procedure."

Subjects gave similar responses to the three other examples of naked procedures in the interview (see `sub-1`, `thing`, and `stuff` in the Appendix). In each case, between 3 and 6 students expected the naked procedure to return an error, explaining that procedures need arguments and/or parentheses to be meaningful.

4.2.2 `Apply-to-5` and `Create-Subtracter`

The `decrement` example highlighted students' difficulties with the first (i.e., naming) property of first-class procedures. `Apply-to-5` and `create-subtracter` probed students' understanding of other first-class properties. These procedures were discussed in Section 2 of this paper; their definitions are repeated below.

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

(define (create-subtracter n)
  (lambda (x) (- x n)))
```

`Apply-to-5` takes a procedure as an argument; `create-subtracter` returns a procedure as a result.

The following expression, using both of these procedures, is particularly useful for illuminating students' models of procedures:

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

The evaluation process for this expression may be summarized as follows: the `apply-to-5` procedure takes as its argument a procedure, and applies that procedure to the number 5. In this instance, the argument to `apply-to-5` is the `create-subtracter` procedure; thus, the `create-subtracter` procedure is applied to 5, and the result is a "subtract-5" procedure — i.e., a procedure which, when applied to some argument, will subtract 5 from that argument.

Although `(apply-to-5 create-subtracter)` is a perfectly valid Scheme expression, nine of the sixteen students stated – at least as their first response[10] – that the expression was in error; one remained unsure of the result. Here is a sampling of quotes from the interviews:

> I don't think this'll work... Because **create-subtracter** when you use, you have to have two arguments, I mean you have to have an argument to **create-subtracter** which is n... and then you have to apply it to some... I mean, you have to give it an argument for x... So I mean **create-subtracter** will only have one parameter in it, it needs another one. [Subject 1]
>
> ... with **create-subtracter**, you haven't given that any arguments. [Subject 2]
>
> **Create-subtracter** doesn't have an argument which it needs, cause when you defined it you define **create-subtracter** of n, so you don't have an argument. [Subject 4]
>
> I can't picture where the argument x comes from... it has the expression x minus 5, but it doesn't have any value for x, so it can't evaluate the expression minus x 5. [Subject 9]
>
> Probably an unbound variable... it's either x or n. [Subject 10]
>
> Possibly error... since n doesn't have a value. [Subject 12]
>
> I'm confused as to where both variables – the x or the n – are coming from... [Subject 13]
>
> ... I think that this would... give a wrong number of arguments error. Because **create-subtracter** needs one argument and isn't given one argument. [Subject 15]

These comments reveal a remarkable consistency. Eight of the ten students who had difficulty with the expression tended to focus on the "missing" argument values for either n or x. For some students (Subjects 2, 4, 12, and 15), the problem was that `create-subtracter` had not explicitly been given an argument. By fixating on the absence of an argument for `create-subtracter`, these students were unable to consider passing that procedure as an argument to `apply-to-5`. For others (Subjects 1 and 9) the problem was that, after `create-subtracter` was applied to 5, there was no value to be bound to x; these students were uncomfortable with the idea of a procedure being returned as the result of a procedure call. Still other students (Subjects 10 and 13) mentioned both "missing" variables in their discussion.

In all these cases, subjects focussed on finding the arguments for a procedure, regardless of whether the procedure is actually being invoked. This illustrates yet again that they saw procedures as active (always eager to run) and incomplete (requiring arguments in order to run).

[10] Two subjects later changed their answers.

Other examples involving `apply-to-5` and `create-subtracter` provide further evidence that subjects ascribed these traits to procedures. For example, consider the following expression:

```
(apply-to-5 (create-subtracter 3))
```

Since both `create-subtracter` and `apply-to-5` are actually being invoked and since all "parts" needed by both procedures are available, this expression was much less problematic for students than the previous one. In fact, 14 of the 16 subjects responded correctly that this expression would evaluate to 2.

A more complicated example is the following:

```
((apply-to-5 create-subtracter) 3)
```

Note how this contains `(apply-to-5 create-subtracter)` as a subexpression. As we have seen, this subexpression evaluates to a "subtract-5" procedure; the application of this procedure to 3 gives the number -2 as the result.

As with `(apply-to-5 (create-subtracter 3))`, the expression `((apply-to-5 create-subtracter) 3)` supplies all "parts" needed to complete both procedures. Even though subjects had more difficulty with this expression, 11 of the 16 converged upon the correct answer. Interestingly, 4 of these 11 subjects had previously concluded that `(apply-to-5 create-subtracter)` was an error.

The fact that `((apply-to-5 create-subtracter) 3)` had all its parts available presumably simplified reasoning about the expression. For a number of the subjects, a crucial step was noticing that both a 3 and a 5 were available to be used as arguments. Subject 13, who never converged on the correct answer, provided a particularly interesting example of reasoning by parts. His comments clearly show that he was keying in on the 3 and the 5 as the relevant parts to match to the arguments n and x, but he wasn't sure whether the expression resulted in 2 or -2. It appears that his structural models were not well-developed enough to say which number corresponded to which variable.

4.2.3 Junk

The junk example proved especially troublesome for the subjects. Only four of the sixteen subjects were correct in their initial answers; two more eventually arrived at the correct answer. The example consists of two expressions:

```
(define junk ((lambda (y) (- 3 y)) 5))
```

```
(junk b)
```

where b has previously been bound to 5.[11]

[11] The use of 5 in the expression for defining junk was unfortunate because b is also bound to 5. During the interviews, we asked most subjects to consider scenarios where either the 5 or the value of b had been changed. This allowed us to distinguish between the two values in analyzing their protocols.

The first expression applies a "subtract-from-3" procedure to 5 and gives the name junk to the resulting number (i.e. -2). Thus, the first expression is equivalent to (define junk -2). The second expression tries to apply the object named by junk to the object named by b, and is thus equivalent to the expression (-2 5). Since -2 is a number, not a procedure, this attempted application results in an "application of a non-procedural object" error. This is a type error; the value of the first subexpression of a procedure call must be of type procedure.

Most subjects expressed confusion about the definition of junk. Indeed, the definition does not match the common patterns of definition generally encountered in the course. Nine students explicitly stated that junk was a procedure or function, and three more, while making no similar claim, clearly used junk as a procedure. Most probably, the presence of an explicit lambda led students to think that junk was a procedure.

A major source of students' confusion with (junk b) was the mismatch between the number of parts needed and the number of parts supplied. The procedure specified by the lambda takes only one argument, but there are two numbers available – the 5 in the definition of junk and the one specified by b. This confusion is clear in a number of the protocols:

> I'm trying to figure out what this 5 does. [Subject 1]
>
> I can't remember what the term 5 is supposed to do. [Subject 2]
>
> Even if you put b in for y, this [the procedure specified by the lambda] is going to give you a number and just this other number [5] is standing right there, doing nothing. If junk does put this 5 in the junk definition, then you have this extraneous argument [b] here . . . [Subject 6]
>
> And if you say the junk of 10 [if the value of b were 10], then my first thought would be to subtract 10 from 3, but then I don't really know what you do with the 5, but you can't really get rid of the 5, because it's there . . . [Subject 13]
>
> I really don't know what the 5 is doing out there. [Subject 12]
>
> I don't know what the 5 is for, exactly. [Subject 15]

As is evident from the above quotes, the "extra" 5 in the definition of junk was disconcerting for many students. The subjects displayed a great amount of inventiveness in circumventing the "mismatch in number of parts" impasse. Four students viewed junk as a procedure whose argument had been already "supplied", but which had not been properly activated. For them, junk could be activated by a pair of parentheses (Scheme's method of procedure invocation). Thus, these students claimed that (junk) evaluated to -2, and (junk b) gave an error (usually a "wrong number of arguments" error). For example, Subject 5 went so far as to claim (correctly) that the definition was equivalent to (define junk -2), but reasoned later that junk "is a procedure of no arguments. There's a lambda in there, yes, but it's already being evaluated. You're already assigning it 5. It doesn't take any other arguments."

In another set of interpretations, eight subjects treated junk — at least at one point — as a procedure of one argument. In these interpretations, calling junk on b was an appropriate use of the procedure. These subjects handled the problem of "too many parts" in a variety of ways. One heuristic was to ignore the "extra" information. Thus, two subjects chose to disregard the 5 in the definition of junk,

and treated `junk` as simply the procedure specified by the `lambda` expression. Another treated `5` as the argument to the procedure specified by the `lambda` expression regardless of what the value of `b` was.

Four subjects did not ignore the "extra" information, and found a way to use both the value of `b` and the `5` in the definition of `junk`. One of these four saw the `lambda` expression and the `5` in the `junk` definition as the two elements of a sequential expression. The `lambda` expression was applied to the value of `b`, but the result of this application was ignored, and the `5` (the second element of the sequence) was always returned.[12] The remaining three subjects also applied the procedure specified by the `lambda` expression to the value of `b`, but further tried to apply the returned object (presumably a number) to the `5`. Although this approach ultimately leads to the right kind of error (the object being applied to `5` is a number, not a procedure), it is for altogether the wrong reasons.

4.3 Discussion

The analysis of the above examples admittedly ignores some subtleties in the subjects' reasoning. Many subjects displayed uncertainty about their answers. They often altered answers to particular expressions, and their reasoning about procedures commonly evolved during the course of their interviews.

Nevertheless, when the students used incorrect reasoning, there was a marked consistency in the nature — and even wording — of their explanations. The commonality is particularly striking considering that these explanations were *wrong*. The students were not mimicking the reasoning presented by an instructor or textbook; they were responding to the expression with a rationale of their own creation. We can only conclude that the difficulty in seeing a procedure as an object is both widespread and (at least to some degree) consistent in its symptoms.

This difficulty is indicative of the students' reliance on functional, rather than structural, reasoning in evaluating Scheme expressions. In the absence of a firm grasp of Scheme's objects and interpretation rules, subjects appealed to numerous special-case evaluation strategies. The `decrement` example — in which students expected the interpreter to treat "naked" procedure names using some informative *ad hoc* rule — is a clear case of this behavior. (This phenomenon is not unlike what Pea *(8)* refers to as the "conversational metaphor" in novices' interactions with interpreted languages.)

Finally, it should be noted that the central concerns of many other programming studies, though important, are not at issue in these examples. The students were *not* required to assimilate or "chunk" large sections of code; no expression was longer than two lines, and students never needed to look back at more than two definitions to evaluate an expression. Thus, the sample expressions placed no burden on the students' memorization skills, nor did they require recourse to higher-order notions of goals or plans [cf. *(9)*, *(10)*, *(11)*]. Moreover, none of the procedures had misleading names; many of the names (such as `apply-to-5` and

[12] The subjects were apparently making use of the fact that the value of a sequence of expressions in Scheme is the value of the last expression in the sequence.

`create-subtracter`) are, in fact, suggestive of the procedures' purposes. Thus, no expression violated any implicit rules of programming etiquette [cf. *(12)*, *(13)*].

Rather, our questions were designed to focus on students' ontologies of procedures. For the novice, a firm grasp of the nature of the objects in a programming language is arguably a prerequisite to mastering higher-level programming techniques. Exploring how students view objects can thus provide an important window into how they learn and understand programming.

5. Pedagogical and Language Design Issues

One of the important benefits of studying the learnability of programming languages is that such investigation can lead to improvements in the teaching and design of programming languages. Our results suggest some possible avenues of exploration in these areas.

At the very least, teachers of Scheme (and presumably other languages with first-class procedures) should be aware of the special difficulties inherent in the first-class procedure concept. For example, upon introducing the concept of a procedure as argument, teachers might lay particular emphasis on the fact that the argument value, although a procedure, is not "looking for" any arguments. In addition, we noticed that the interview format (a sequence of expressions covering a broad range of uses of procedures) forced students to reconsider their models in the interest of consistency. This format could be profitably used in instruction.

In the interviews, we observed that students' textual and graphical representations of procedures were flawed — or non-existent. This is hardly surprising given that the computer's printed representation is the rather abstruse `[COMPOUND-PROCEDURE` *name*`]`, and the MIT course does not provide an explicit representation for procedure objects until later in the course. It is difficult to view something as an object if you don't have a way to envision it. Thus, one possible teaching strategy would be to provide, early in the course, an abstract representation for procedures that stresses their "object-like" character; this representation could then be made more concrete and realistic over time.

The procedure-representation problem could also be addressed through changes in language design (or perhaps more accurately, language interface design). MIT Scheme's printed representation is unhelpful for novices striving to understand what a procedure might be. It would be worthwhile to explore some alternative printed representations of procedures. For example, a procedure might be displayed as a picture of a script or recipe; these are images which we ourselves use in explaining the nature of procedures [cf. *(14)*, *(15)*].

The existence of two syntaxes for defining procedures is another potential source of confusion, inasmuch as it perpetuates the mistaken distinction between procedures and other objects. Although removing the procedure-specific form from the language might be too radical a change, programming courses could stress the more general form, at least in the beginning of the course.

Further, we believe that `define` expressions could return more informative

results.[13] In the MIT version of Scheme, `define` expressions return the name being bound. It would be more useful if Scheme printed some representation of both the name and the object to which it is bound. Such a representation would underscore that names are bound to procedures in the same way that they are bound to all other first-class objects.

6. Future Research

In conclusion, we suggest a number of directions for future research:

Generality of the results. We believe that the results of this study are meaningful for any language with first-class procedures, but this issue should be resolved empirically. In other words, how "Scheme-specific" are our results? Do students' difficulties with first-class procedures transcend the particular language in which they are working?

Procedures in other domains. Do mathematics students dealing with concepts such as "groups of operators" or "function spaces" experience the same problems as students of Scheme? Is there any transfer between domains – that is, after students have mastered the concept of first-class procedures in Scheme, can they transfer the concept to work in mathematics?

Origins. What are the origins of the naive ontology of procedures? Is the ontology related to linguistic structures (e.g. noun/verb distinctions) in English? Is it related to experience with other programming languages?

Finer-grained models. How do students view the first position of a procedure-call expression? In what ways do students see "naked" procedure names as different from names bound to other objects? How do variations in Scheme syntax affect students' ontological models?

Other first-class objects. What difficulties do students have in understanding languages in which other objects, such as environments and continuations, are first-class?

Acknowledgments

We would like to thank Andy diSessa, Hal Abelson, Gerry Sussman, and Roy Pea for providing suggestions and advice during the course of the study. We'd also like to thank Louis Braida and Rod Brooks for helping us to obtain subjects from their course.

The sole criterion used in listing the authors' names was alphabetical order.

[13] Since defining is done for effect rather than value, its return value is arbitrary in terms of program semantics.

References

1. Abelson, Harold and Sussman, Gerald Jay with Sussman, Julie. *Structure and Interpretation of Computer Programs.* MIT Press, 1985.

2. Steele, Guy Lewis, Jr. and Sussman, Gerald Jay. *Lambda: the Ultimate Imperative.* AI Memo 353, MIT AI Lab (1976).

3. Atkinson, Malcolm P. and Morrison, Ronald. *Procedures as Persistent Data Objects.* ACM Transactions on Programming Languages and Systems, 7:4, October 1985.

4. Eisenberg, Michael. *Programming in Scheme: an Introduction.* Scientific Press, 1988 (in preparation).

5. Rees, Jonathan and Clinger, William (eds.). *Revised^3 Report on the Algorithmic Language Scheme.* ACM Sigplan Notices, 21:12, December 1986.

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

7. diSessa, Andrea. *Models of Computation.* In *User Centered System Design,* Norman, Donald and Draper, Stephen, eds. Lawrence Erlbaum, 1986.

8. Pea, Roy. *Language-Independent Conceptual "Bugs" in Novice Programming.* J. Educational Computing Research, 2:1 1986.

9. Anderson, John R. and Jeffries, Robin. *Novice LISP Errors: Undetected Losses of Information from Working Memory.* Human-Computer Interaction, 1:2 (1985) 107-131.

10. Rist, Robert S. *Plans in Programming: Definition, Demonstration and Development.* In *Empirical Studies of Programmers,* Soloway, E. and Iyengar, S. (eds.). Ablex, 1986.

11. Spohrer, James C.; Soloway, Elliot; and Pope, Edgar. *A Goal/Plan Analysis of Buggy Pascal Programs.* Human-Computer Interaction, 1:2 (1985) 163-207.

12. Soloway, Elliot; Ehrlich, Kate; and Black, John. *Beyond Numbers: Don't Ask "How Many"... Ask Why.* In *Human Factors in Computing Systems,* CHI'83 Conference Proceedings, 1983.

13. Soloway, Elliot and Ehrlich, Kate. *Empirical Studies of Programming Knowledge.* IEEE Transactions on Software Engineering, Sept. 1984.

14. Turbak, Franklyn. *Grasp: a Visible and Manipulable Model for Procedural Programs.* Master's Thesis, MIT 1986.

15. Eisenberg, Michael. *Bochser: an Integrated Scheme Programming System.* MIT Laboratory for Computer Science Technical Report 349, October 1985.

Appendix

This appendix presents the expressions used in the interviews. The expressions are to be treated as if they were evaluated one after the other in a single session with the Scheme interpreter. Subjects were asked to describe the value returned by each expression; they were informed that some expressions might lead to errors. Whenever a subject said that an expression would lead to an error, we asked him to describe the type of error; in certain cases, we also asked the subject to generate a similar expression that would not give an error.

The results of evaluating the expressions are also provided below (these, of course, were not presented to subjects).

```
(define a 1) ⟹ A

a ⟹ 1

(define b (+ 2 3)) ⟹ B

b ⟹ 5

(- b 1) ⟹ 4

(define c b) ⟹ C

c ⟹ 5

(define d (- b 1)) ⟹ D

d ⟹ 4

(b + 3) ⟹ Error! Object being applied is not a procedure: 5

(define (square x)
  (* x x))          ⟹ SQUARE

(square b) ⟹ 25

b ⟹ 5

(define (mul-by-self) square) ⟹ MUL-BY-SELF

(mul-by-self 4) ⟹ Error! Wrong number of arguments: 1

(define (decrement x)
  (- x 1))          ⟹ DECREMENT

(decrement b) ⟹ 4

b ⟹ 5

decrement ⟹ [COMPOUND-PROCEDURE DECREMENT]
```

```
(define sub-1 decrement) ⟹ SUB-1

(sub-1 b) ⟹ 4

sub-1 ⟹ [COMPOUND-PROCEDURE DECREMENT]

(define (what-not x) 1) ⟹ WHAT-NOT

(what-not) ⟹ Error! Wrong number of arguments: 0

(define (thing) (+ 4 b)) ⟹ THING

thing ⟹ [COMPOUND-PROCEDURE THING]

(thing) ⟹ 9

(define stuff (lambda (x) (/ x 3))) ⟹ STUFF

stuff ⟹ [COMPOUND-PROCEDURE STUFF]

(stuff b) ⟹ 1.66666

(define junk ((lambda (y) (- 3 y)) 5)) ⟹ JUNK

(junk b) ⟹ Error! Object being applied is not a procedure: -2.

(define something (lambda () (- 9 b))) ⟹ SOMETHING

(something 6) ⟹ Error! Wrong number of arguments: 1

(define (apply-to-5 f)
  (f 5))                ⟹ APPLY-TO-5

; Give an example of how you would use APPLY-TO-5

(define (create-subtracter n)
  (lambda (x) (- x n)))        ⟹ CREATE-SUBTRACTER

; Give an example of how you would use CREATE-SUBTRACTER

(apply-to-5 create-subtracter) ⟹ [COMPOUND-PROCEDURE 12430420]

(create-subtracter apply-to-5) ⟹ [COMPOUND-PROCEDURE 12437452]

(apply-to-5 (create-subtracter 3)) ⟹ 2

((apply-to-5 create-subtracter) 3) ⟹ -2

(((apply-to-5 create-subtracter) 3) 9) ⟹ Error! Object being applied
is not a procedure: -2
```

EMPIRICAL STUDIES OF PROGRAMMERS: SECOND WORKSHOP

edited by

GARY M. OLSON

University of Michigan

SYLVIA SHEPPARD

Computer Technology Associates, Inc.

ELLIOT SOLOWAY

Yale University

ABLEX PUBLISHING CORPORATION
NORWOOD, NEW JERSEY

Printed in the United States of America

Library of Congress Cataloging-in-Publication Data

Empirical studies of programmers : second workshop / edited by Gary M. Olson, Sylvia Sheppard, and Elliot Soloway.
p. cm. — (Human/computer interaction)
Papers presented at the Second Workshop on Empirical Studies of Programmers held in Washington, D.C. on Dec. 7-8, 1987.
Bibliography: p.
ISBN 0-89391-461-4. ISBN 0-89391-462-2 (pbk.)
1. Computer programmers—Congresses. I. Olson, Gary M. II. Sheppard, Sylvia. III. Soloway, Elliot. IV. Workshop on Empirical Studies of Programmers (2nd : 1987 : Washington, D.C.) V. Series: Human/computer interaction (Norwood, N.J.)
HD8039.D37E46 1987 87-25942
331.7'610051—dc19 CIP

Ablex Publishing Corporation
355 Chestnut Street
Norwood, New Jersey 07648