PL-Detective: A System for Teaching Programming Language Concepts

AMER DIWAN, WILLIAM M. WAITE, MICHELE H. JACKSON,
and JACOB DICKERSON
University of Colorado at Boulder

The educational literature recognizes that people go through a number of stages in their intellectual development. During the first stage, called *received knowledge* or *dualism*, people expect knowledge to be handed to them by authority figures (thus "received") and think in terms of black and white (thus "dualism"). Our experience indicates that many computer science students are at this first stage of learning. To help students move beyond this stage, we describe a system and strategy, the PL-Detective, to be used in a Concepts of Programming Languages course. Assignments using this system directly confront students with the notion that they can create knowledge via interactions with the PL-Detective and that discussion with students (rather than asking the instructor) is an effective way of learning how to reason. We present experimental results that show that the PL-Detective is effective in helping students move beyond the stage of received knowledge.

Categories and Subject Descriptors: D.3.m [**Programming Languages**]: Miscellaneous

General Terms: Human Factors, Languages

Additional Key Words and Phrases: Concepts of programming languages, educational tools, collaboration

1. INTRODUCTION

People go through a number of stages in their ability to learn [Perry 1970; Belenky et al. 1997]. The first stage, *received knowledge*, is when a subject expects an authority figure to provide the information that he or she needs to know. Subjects at this stage think in terms of black and white and thus this stage is also called *dualism* [Perry 1970]. In later stages, subjects increasingly realize that information can come from within themselves, i.e., they

Contact Author's addresses: A. Diwan, Computer Science Department, University of Colorado, Denver, Colorado. D. White, Department of Computer Science, University of Warwick, Coventry, CV4 7AL, UK; email: D.R.white@warwick.ac.uk; email: {Amer.Diwan,William.Waite,Michele.Jackson,Jacob.Dickerson}@Colorado.edu.

This work is supported by NSF grant CCR-0086255. Any opinions, findings and conclusions or recommendations expressed in this material are the authors' and do not necessarily reflect those of the sponsors.

Earlier versions of this work were presented at the 2004 and 2005 Conferences of the ACM SIG for Computer Science Education.

can *create* knowledge. Proceeding through these stages brings the awareness that there may be many possible answers, each with its own merits and weaknesses.

Our experience in the classroom indicates that many of our students are still at the stage of received knowledge: they prefer the instructor to lecture to them and expect exams and assignments to test only material that the instructor has covered in class. We want students to realize that they can create knowledge (rather than always counting on receiving it from an authority figure, such as an instructor). To this end we have designed, implemented, and evaluated a system, the PL-Detective,[1] with the goal of helping students to move beyond received knowledge. We identified two subgoals that would help us attain our main goal:

1. *Get students to collaborate with each other inside and outside the classroom*. When students collaborate with each other in solving complex tasks they realize that the instructor is not the sole source of information [Alavi 1994; Williams and Kessler 2000]. Collaboration is particularly important when a problem has a number of possible solutions, and none stands out clearly as the best. Finely balanced design decisions made by an individual are often based on unstated prejudice or very specific experiences. True collaboration can expose those prejudices and bring a variety of experiences to bear. Thus collaborative decisions are often better than those made by an individual and students learn the underlying material faster [Abercrombie 1960].

   Unfortunately, our findings from ethnographic observation and interviews with over 130 computer science and computer engineering students [Leonardi 2003; Waite et al. 2004] show that students actively develop strategies to avoid collaboration. We, the professors, need to design exercises that reward collaboration so that students have an incentive to collaborate.
2. *Get students to discover information by interacting with an experimental environment*. Experimental environments teach students to generate knowledge as a result of their interactions, instead of obtaining it from an authority figure. Such interactions are likely to be most effective and rewarding to students if the knowledge that they are trying to gain is complex and there are many possible strategies for gathering the information. A hardware simulator is an example of such an environment: students interact with the simulator by submitting traces and observing the steps that the simulator takes in order to simulate the traces. Many instructors have found such exercises to be effective pedagogical tools in Computer Organization courses.

We have developed a tool, the *PL-Detective*, for building assignments and demonstrations in the context of a course titled *Principles of Programming*

[1]Available with an open-source licence from `http://www.cs.colorado.edu/~pltools`.

Languages.[2] PL-Detective assignments reward true collaboration and provide an environment with which students interact in order to obtain information.

The PL-Detective is a flexible and extensible implementation of a language called MYSTERY. It supports a fixed syntax for MYSTERY but allows the semantics of the MYSTERY program to be varied. The current version of the PL-Detective exports ten interfaces, each of which controls the semantics for a single aspect of MYSTERY. We provide at least two implementations for each semantic interface. Every combination of implementations for the ten semantic interfaces defines a particular behavior for MYSTERY programs.

The PL-Detective supports two kinds of assignments, both of which encourage collaboration and require students to interact with the system to discover knowledge. The first kind, *language analysis*, is structured as a puzzle: Students attempt to discover the semantics of a particular MYSTERY implementation by running programs and observing the results. The second kind of assignment, *language design*, requires students to select implementations for the semantic interfaces such that a given MYSTERY program will produce specified results. For both kinds of assignments, the instructor limits the number of attempts allowed for each group of students to prevent an unprincipled "trial and error" approach.

We used social scientific techniques to evaluate the PL-Detective in two ways. First, we evaluated the effectiveness of the PL-Detective in meeting its first subgoal (i.e., whether it encourages students to collaborate) using in-class surveys and focus-group interviews (conducted by a member of the research team). Second, we evaluated the effectiveness of the PL-Detective in meeting its second subgoal (i.e., whether students acquire knowledge through interactions with the system) by collecting, coding, and analyzing data from eleven exercises using the PL-Detective over the course of 6 weeks. The study involved 29 groups of two or three students. Our results demonstrate that the PL-Detective is effective in achieving both subgoals.

This paper is the culmination of previous work in which we introduced the PL-Detective [Diwan et al. 2004] and provided initial reports of its effectiveness [Diwan et al. 1995]. Compared to the conference papers, the current paper contains more detailed and up-to-date description of all aspects (including evaluation) of the PL-Detective.

The rest of the paper is organized as follows. Section 2 describes the MYSTERY language. Section 3 outline the semantic interfaces supported by the PL-Detective; we have added three new interfaces since our conference papers. Section 4 describes exercises using the PL-Detective, adding more examples and descriptions compared to our conference papers. Section 5 describes in detail the implementation of the PL-Detective. Section 6 presents experimental results evaluating the PL-Detective with respect to the two subgoals laid out above. Section 7 discusses some of the lessons we have learned after using the

[2]Computer science and computer engineering majors take this course in their junior year after they have already taken a number of programming courses, including Data Structures and Computers as Components.

Program	→	*Block* | *Block* ;
DeclList	→	*Decl* | *Decl* ; *DeclList* | ϵ
Decl	→	VAR id : *Type* | TYPE id = *Type* | *ProcDecl*
ProcDecl	→	PROCEDURE id (*Formals*) : *Type* = *Block* | PROCEDURE id (*Formals*) = *Block*
Formals	→	*FormalList* | ϵ
FormalList	→	*Formal* | *FormalList* ; *Formal*
Formal	→	id : *Type*
Type	→	INTEGER | id | *SubrangeType* | *ArrayType* | *ProcType*
SubrangeType	→	[Number TO Number]
ArrayType	→	ARRAY *SubrangeType* OF *Type*
ProcType	→	PROCEDURE (*Formals*) : *Type* | PROCEDURE (*Formals*)
Block	→	*DeclList* BEGIN *StmtList* END
StmtList	→	*Stmt* | *Stmt* ; *StmtList* | ϵ
Stmt	→	*Assignment* | *Return* | *Block* | *Conditional* | *Iteration* | *Output* | *Expr*
Assignment	→	*Expr* := *Expr*
Return	→	RETURN *Expr*
Conditional	→	IF *Expr* THEN *StmtList* ELSE *StmtList* END
Iteration	→	WHILE *Expr* DO *StmtList* END
Output	→	PRINT *Expr*
Expr	→	*Operand* | *Expr Operator Operand*
Operand	→	Number | id | *Operand* [*Expr*] | *Operand* (*Actuals*) | (*Expr*)
Operator	→	+ | > | AND
Actuals	→	*ActualList* | ϵ
ActualList	→	*Expr* | *ActualList* , *Expr*

Fig. 1. Syntax of MYSTERY.

PL-Detective for three semesters. We hope that these lessons will be useful for other instructors using this tool. Section 8 presents related work. Section 9 concludes the paper.

2. MYSTERY LANGUAGE

The PL-Detective is a flexible and extensible implementation of the MYSTERY language. We based the design of MYSTERY on two principles:

- MYSTERY should exhibit many, if not most, of the *concepts* covered in traditional undergraduate courses covering the principles of programming language.
- MYSTERY should contain only the features *needed* to exhibit these concepts.

We used the textbook for our course [Sebesta 2003] (and, in particular, its excellent "design issues" lists) as a checklist for necessary concepts, and rigidly excluded features that required no additional concepts to explain them.

The MYSTERY syntax (Figure 1) is a subset of Modula-3 syntax [Nelson 1991] (which is based on Pascal [Jensen and Wirth 1991] syntax). We picked Modula-3 syntax instead of the more popular syntax from C [Kernighan and Ritchie 1988] because we find some aspects of Modula-3 syntax to be cleaner and easier to read. For example, the following declares a variable of procedure type with one

argument of type INTEGER and a return type of BOOLEAN in Modula-3 and C-family syntax:

```
VAR x: PROCEDURE (i: INTEGER): BOOLEAN; // Modula-3
boolean (*x)(int);                      // C
```

While the Modula-3 syntax is more verbose, we find it to be more readable and intuitive. Thus Modula-3 syntax will be easier for students to understand and master.[3] Note that just because we use Modula-3 syntax, it does not mean that our tool is suitable only for courses on Modula-3 and similar languages. Indeed, our own "Principles of Programming Languages" course focuses on many languages unrelated in syntax to Modula-3, such as Java, C++, and SML.

MYSTERY provides all of the standard structuring concepts for actions in imperative languages (sequencing, conditionals, iteration, and abstraction). Integers, subranges, arrays, and functions are all first-class values in MYSTERY (i.e., they can be assigned to variables and passed as parameters). To illustrate the concept of scoping, MYSTERY allows nesting of blocks and function definitions.

The + operator allows a program to create new integer values, while the > operator allows the program to create truth-values for use in conditions. MYSTERY does not have a first-class Boolean type since such a type does not introduce significant new semantic issues. We included the AND operator so that we could introduce the concept of short-circuit evaluation in MYSTERY. Other logical or arithmetic operations, such as OR or subtraction, do not introduce any new semantic issues and thus we omitted them.

Arrays are the only data-structuring mechanism provided by MYSTERY. MYSTERY omits records because many of the interesting semantic issues with records (e.g., distinction between name and structural type equivalence) can be explored with arrays. We omitted objects for the same reason. For example, arrays, subrange types, and first-class functions together can demonstrate several of the subtyping issues that arise in object-oriented languages. First-class functions are also invaluable for demonstrating some of the flexibility and power of functional programming languages.

3. SEMANTIC INTERFACES FOR MYSTERY

The PL-Detective exports ten semantic interfaces. Each semantic interface corresponds to one aspect of the semantics of MYSTERY. For example, the *type-assignability interface* determines the type assignability rules for MYSTERY: given an l-value, what are the legal types whose instances can be assigned to this value? To pick the semantics for the full MYSTERY one needs to pick an implementation for each semantic interface. The ten semantic interfaces are as follows; the numbers in parenthesis give the number of implementations that we already provide for the interface.

- **Scoping discipline**(2). Determines the binding of names. For example *static scoping* uses the static nesting of blocks to determine scoping, as provided by

[3]In our experience, the concept of first-class functions is foreign to most students in the class and thus, prior familiarity of the students was not an issue in this decision.

most current languages, such as C, C++, and Modula-3. Other possibilities include dynamic scoping, and variations of static and dynamic scoping.

- **Type of**(2). Determines the type of a constant or expression. For example, is the type of the constant "7" `INTEGER` or a subrange type `[7 TO 7]`?
- **Type equality**(4). Determines when two types are equal. For example, Modula-3 uses *structural equality*, which deems two types to be equal if they have the same structure. Other possibilities include name equality and combinations of the two (e.g., using structural equality for arrays and name equality for structs in C and C++).
- **Type assignability**(3). Determines when an assignment is legal with respect to types. For example, Java [Arnold et al. 2000] (for the most part) allows assignments when the type of the right-hand side can be widened to the type of the left-hand side. Other possibilities include requiring the types to be equal and allowing narrowing conversions.
- **Type passability**(3). Determines when an actual can be passed to a formal parameter. For example, in Modula-3 [Nelson 1991] the types of the actual and formal must be the same for `VAR` parameters but the actual's type can be (for the most part) any type that can be widened or narrowed to the formal's type for `VALUE` parameters.
- **Type operands**(3). Determines the types of arguments that one can pass to built-in operators such as "+" and ">." For example, one may allow only `INTEGER`-typed arguments to "+" or one may allow any any type that can be widened to an `INTEGER` (which includes subranges).
- **Type subscript**(2). Determines the types of expressions with which one can index into an array. An example is to require the expression to be of an `INTEGER` type. Another possibility is to allow any type as long as it can be widened to an `INTEGER` (which includes subranges).
- **Parameter passing mode**(4). Determines how to pass parameters. Possibilities are pass-by-value, pass-by-name, pass-by-reference, and pass-by-value-result.
- **Argument evaluation order**(2). Determines how and when to evaluate parameters to a call. For example, Java requires arguments to be evaluated from left to right. Another possibility is to use undefined evaluation order (e.g., C [Kernighan and Ritchie 1988] and C++ [Stroustrup 1991]).
- **Short circuit evaluation**(2): Determines whether or not the logical operators are short-circuited. Some languages use short circuit for some operators (e.g., "&&" in C) and do not use short circuit for other operators (e.g., "&" in C).

Multiplying out the implementations for each interface, we see that the current snapshot of the PL-Detective supports 13824 different semantics for MYSTERY. Of course, not all combinations make sense (more on this in Section 4.2).

We chose the above semantic interfaces based on exercises and demonstrations we wanted to use in our classes. The original version of the PL-Detective [Diwan et al. 2004] had 7 semantic interfaces; now we have 10.

So far we have found it easy to add new interfaces and implementations as needed.

We have also found it easy to add new implementations of semantic interfaces. The files for semantic interface implementations are typically 20–30 lines of Java code (including imports, class headers, and other declarations). One notable exception is the class that implements static scoping, which is over 100 lines. Since this class needs to push and pop environments at the entry and exit of each block and procedure, it is not surprising that it is quite large.

4. EXERCISES USING THE PL-DETECTIVE

We now describe two kinds of exercises for which we have used the PL-Detective. Both kinds of exercises are designed to satisfy the two subgoals for the PL-Detective. Both kinds meet the first subgoal by being high on multiplicity. Prior work [Shaw and Blum 1965] suggests that greater collaboration leads to greater success in tasks with high solution multiplicity. Thus, we can expect these assignments to reward groups that collaborate. Both kinds meet the second subgoal by requiring students to interact with the PL-Detective in order to obtain information (although the kind of information that students get from the PL-Detective is different for the two kinds of exercises).

4.1 Using the PL-Detective for Language Analysis

From the perspective of students, the goal of an analysis assignment is to determine the semantics of a particular MYSTERY implementation by interrogating the PL-Detective. In order to interrogate, student groups submit programs in MYSTERY syntax using a web interface. PL-Detective responds with any output produced during compilation or execution of the submitted program. On receiving the output, a group may decide that it has figured out the semantics of MYSTERY or may decide to continue the interrogation by submitting another program. In the former case, the group produces a "language report" that defines the discovered semantics and carefully describes how the group came to their conclusion. That description includes the evidence gathered by the group, which takes the form of a sequence of programs that the group submitted, the output they received, and the conclusions they drew from each submission.

From the perspective of the instructor, the goal is to sharpen the students' understanding of a concept by having them reflect on the effects of different interpretations of that concept. To use the PL-Detective, the instructor first decides which semantic interfaces are relevant to the material that the assignment needs to cover. Since language features interact with each other, it may be that more than one interface is related to a given programming language concept. The instructor picks the implementations of all interfaces that are not relevant to the assignment in question and tells students what those implementations are. For the relevant interfaces, there are two possibilities: (i) the instructor can either choose their implementations (but not reveal the implementations to the students); or (ii) the instructor can use a randomization procedure to provide a potentially different configuration to each group. For example, in an assignment on parameter passing, one group may get pass-by-name, while another may get

```
PROCEDURE f(x: INTEGER) =
  BEGIN
    x := 20;
  END;
VAR p: INTEGER;
BEGIN
  p := 10;
  f(p);
  PRINT p;
END;
```

Fig. 2. Code to distinguish between parameter passing modes.

pass-by-copy-in-copy-out. Finally, the instructor needs to limit the number and kinds of programs that a group may use in its interrogation. The limit may be hard (e.g., the interrogation may not use more than five procedure calls total) or may be soft (any procedure calls beyond the fifth are charged a 5-point penalty). The instructor should base this limit on the number of attempts a student group would take to discover the semantics if the group undertook a careful and systematic exploration of the search space.

As a concrete example, consider an exercise designed to teach students about parameter passing modes. In this exercise, students probe the PL-Detective with programs designed to expose the difference between different parameter passing modes. Let's suppose a student group submits the program in Figure 2 to the PL-Detective. If this program produces the output "20," then students can immediately eliminate pass-by-value as a parameter-passing mechanism. It will take more probes to distinguish between the remaining parameter-passing mechanisms (e.g., pass-by-name, pass-by-copy-in-copy-out, pass-by-reference). If, on the other hand, this code produces the answer 10, the group can be quite confident that it uses parameter passing by value at least for integers.

When a student probes the system, the probe may or may not compile successfully. If it compiles successfully, it may or may not run successfully. In the case of an unsuccessful compile or run, it is important to provide output that is useful to the students. Since one error in a program can often cause other errors in the program and error recovery in a compiler is tricky and can sometimes yield unpredictable results, the PL-Detective halts on the first error. In other words, each program submission can yield at most one error message.

It is worth noting that error messages can contain as much useful information as program outputs. For example, consider an exercise where students need to discover whether MYSTERY uses static or dynamic scoping. If the program in Figure 3 produces an error ("x not found"), then it indicates static scoping. If the program prints 20, then it indicates dynamic scoping.

4.1.1 *Discussion.* The language analysis exercises attempt to support the goal of getting students to collaborate as follows. These assignments are complex and high on solution multiplicity [Shaw 1981] meaning that there is not "a single acceptable outcome that can be easily demonstrated to be correct." At each step of this assignment, students need to pick a strategic probe based on knowledge of all their previous probes and outcomes. Moreover, groups will

```
PROCEDURE f() =
VAR x: INTEGER;
BEGIN
  x := 20;
  g();
END;

PROCEDURE g() =
BEGIN PRINT x; END;

BEGIN
  f();
END;
```

Fig. 3. Distinguishing between static and dynamic scoping.

most likely pick very different sequences of probes, even if they arrive at the same conclusions. The nature of the task, therefore, resists segmentation, and the multiplicity rewards students who collaborate [Shaw and Blum 1965]. In addition, the number of attempts that students can use is limited; thus, students need to make each and every attempt count, which further rewards students who collaborate.

The language analysis exercises also directly support the goal of getting students to discover information by interacting with an experimental environment. Students probe it with "questions" (the programs) and the PL-Detective responds with "answers" (outputs and error messages) provided that the questions are good.

4.2 Using the PL-Detective for Language Design Exercises

From the perspective of students, the goal of these assignments is to design a language that exhibits a particular behavior. An assignment specifies one or more programs and their desired outputs. Students must design a language (by picking appropriate implementations of the semantic interfaces) that yield the desired output. Students try out different designs using a web interface: They pick the implementations of semantic interfaces they want to use and the system runs the PL-Detective with their choices on the example program and produces the output. Given that many different semantics could produce the same answer, the group report needs to not just list which implementations they used but also state why. More specifically, the report needs to argue that their language design is a sound one and makes sense beyond the examples specified in the assignment.

From the perspective of the instructor, the goal is to get the students to exercise judgment in the design space defined by the semantic concepts. The instructor's responsibilities are similar to a language analysis assignment (Section 4.1). As before, the instructor needs to choose the implementation of the semantic interfaces that are not relevant to the assignment. Also, as before, the instructor needs to limit the number of attempts allowed per group.

As a concrete example, consider again an assignment designed to teach students about parameter-passing modes. The instructor provides the students with the program and desired output in Figure 4. The students need to pick

```
PROCEDURE f(x: INTEGER) =
  BEGIN
    x := 20;
  END;
VAR p: INTEGER;
BEGIN
  p := 10;
  f(p);
  PRINT p;
END;
```

Desired output: 20.

Fig. 4. Instructor-provided code and its desired output.

(i) which parameter-passing mode to use; and (ii) a type rule that determines when an actual is allowed to be passed to a formal. The instructor picks implementations for all the other semantic interfaces and reveals them to the students. For (i) students can pick from many alternatives, e.g., pass-by-result or pass-by-reference. However, the alternative they pick for (i) will determine what makes sense for (ii). If, for example, students pick pass-by-reference, then they will have to require that argument and result types must be identical (for part (ii); otherwise, they will end up with unsoundness in their language.

4.2.1 *Discussion.* The language design exercises support collaboration because there are many ways of approaching the problem. However, unlike the language-analysis assignments, there is multiplicity even in the final answer (the language design); many possible language designs may yield the same outcome for the examples specified in the assignment.

While the PL-Detective allows users to pick implementations of one semantic interface independently from implementations of other interfaces, many combinations do not make sense. For example, it is well known that allowing subtypes to be passed to a pass-by-reference parameter is unsound. Thus some answers may be preferable to others based on objective considerations (e.g., soundness) or subjective considerations (e.g., taste). We expect the very nature of these assignments to reduce dualistic thinking among students, which is one of the key aspects of moving away from received knowledge.

The language-analysis exercises also support the second goal of getting students to discover information by interacting with an experimental environment. Students interact with the PL-Detective by picking a semantics for MYSTERY; the PL-Detective responds with output and error messages for a given program when using the chosen semantics. Since the choices for one semantic interface often interact with choices for other interfaces, the task of picking a consistent semantics is complex. Thus, even though it is possible to do this exercise without interacting with the PL-Detective, it is much more manageable with the interactions.

5. IMPLEMENTATION

The PL-Detective consists of three components. The first component is a compiler that translates MYSTERY programs into Java. The second component is the

run-time system necessary for running the translated MYSTERY programs. The third is a user interface via which students can interact with the PL-Detective.

5.1 The PL-Detective Compiler

The PL-Detective compiler is a 4049-line[4] Java [Gosling et al. 2000] program that translates MYSTERY programs into Java programs. The semantic interface implementations contribute 973 to the line count of the compiler. During the translation, the PL-Detective makes all allocations, lookups, and control explicit.

By making all *allocations* explicit, we mean that all variables and values are allocated on the heap and deallocated by the Java garbage collector. Even integer constants are allocated on the heap; they are instances of the **`IntValue`** class (Section 5.2). In this way the PL-Detective has great flexibility in picking storage bindings for every variable and value. For example, if PL-Detective had mapped MYSTERY's local variables to Java local variables, we would have been stuck using stack dynamic binding for MYSTERY's local variables. Instead, by allocating MYSTERY's local variables on the heap, we can choose to emulate any storage binding we wish (e.g., static or stack dynamic).

PL-Detective also uses heap-allocated values to represent functions. More specifically, PL-Detective generates a closure for each MYSTERY function. The code component of each closure takes exactly one argument, which is a list of argument values. The environment component of each closure exports a method that maps names to values. Since all values and variables are allocated on the heap, these closures can support first-class functions, as found in functional languages. More specifically, one can easily write functional forms in MYSTERY (although we admit the syntax is not as compact or elegant as found in functional languages such as SML [Milner et al. 1990]!).

By making all *lookups* explicit, we mean that all variable accesses in MYSTERY programs are represented by environment lookups. The environments themselves, are of course, first-class values that are allocated in the heap. This gives us great flexibility in determining the scoping mechanism. For example, one can get static or dynamic scoping by merely changing the way in which environments are linked together (the linkage determines the order in which environments are searched when looking for a name).

By making all *control* explicit, we mean that we do not rely on any of Java's implicit control flow or evaluation order. For example, Java determines that arguments to functions are evaluated from left-to-right. By making this evaluation order explicit, PL-Detective can easily emulate Java's left-to-right evaluation or any other evaluation order that it wishes.

5.2 The PL-Detective Run-Time System

As mentioned in Section 5.1, the PL-Detective puts all MYSTERY values and variables on the heap. The PL-Detective run-time system defines the classes whose instances represent the data on the heap when a MYSTERY program runs.

[4]Excluding comments and blank lines.

The PL-Detective run-time system is 342 lines of Java code (excluding blank lines and comments).

Most classes in the PL-Detective run-time are subclasses of a `RTValue` class. Each subclass is used to represent one kind of value and contains methods relevant to the value. For example, the `Closure` subclass of `RTValue` has a method for the invoking the closure; the `ArrayValue` subclass exports a subscript method that returns the `RTValue` residing at a given index in the array value.

5.3 The PL-Detective User Interface

The user interface is a 108-line cgi-script that takes a MYSTERY program and semantic choices from a web form. It invokes the PL-Detective compiler to generate Java code, the Java compiler to generate bytecodes from the Java code, and the Java VM to run the Java bytecodes. We have two kinds of web interfaces that use this cgi-script. The first web interface provides menus that enable users to pick the implementations for some subset of the semantic interfaces. We use this interface for the language-design exercises. The second web interface has a hidden html-variable, "script," that names a file indicating which implementation to use for each semantic interface. The script file resides on the web server and is not accessible to users. We use this interface for the language-analysis exercises.

5.4 Summary of Implementation

The PL-Detective compiler and run-time system add up to about 4400 lines of code, which is quite small. Its structure is designed to be easily extensible with new semantic interfaces and implementations. This makes the PL-Detective easy to debug and extend. Since the first paper on the PL-Detective [Diwan et al. 2004], we have added three new semantic interfaces and a number of new semantic implementations. We estimate that all these changes combined took no more than a day of programming. The small size and clean design of the PL-Detective has also made it relatively bug free: so far approximately 300 students at the University of Colorado have used the PL-Detective for a total of approximately 20,000 programs and exposed only two bugs, both of which took minutes to fix.

6. DOES THE PL-DETECTIVE MEET ITS DESIGN GOALS?

We now present experimental results that demonstrate that the PL-Detective is successful in meeting its design goals.

6.1 Subgoal 1: PL-Detective and Collaboration

Is the PL-Detective helpful in getting students to collaborate?

6.1.1 *Methodology.* To assess the effectiveness of the PL-Detective, we collected several forms of data. First, we invited students to work on an assignment in a specially equipped lab that would allow us to videotape their interactions. However, only two groups participated. Consequently, we used that data only to inform our analysis of other data. Second, we conducted focus-group interviews

Table I. Description of Assignments[a]

#	Description	Assign.	Limit
1	What is the storage binding of local variables.	3	3 prints
2	When are two types are equal	3	8 progs.
3	Do type declarations create a new type	3	2 progs.
4	Does MYSTERY use static or dynamic scoping	4	3 progs.
5	What are the semantics of array assignments	5	6 prints
6	What is the evaluation order in a procedure call	5	4 prints
7	Does MYSTERY use short-circuit evaluation	6	3 prints
8	When is one procedure type a subtype of another	6	6 progs.
9	What is the parameter passing mechanism	7	10 prints
10	What is the parameter passing mechanism	7	10 prints
11	Does MYSTERY use deep or shallow binding	8	4 prints

[a]Assignments 9 and 10 use different implementations for the parameter-passing mode semantic interface.

with students and administered a survey asking students to describe how they used the PL-Detective and to provide their general reactions regarding its usefulness. We administered these during lab sections. The instructor left the room, students completed the surveys, and then a member of the research team conducted the interview. The interviews were audio recorded. The students were informed that their responses were confidential, and that the instructor would see only summaries of the results. The surveys were filled out individually, rather than as a group. A total of 37 students (out of a total class enrollment of 90) completed surveys.

6.1.2 *Results.* The results of the survey strongly suggest that the PL-Detective is helpful in getting students to collaborate. Of the groups 60% indicated that "we do all the assignment together all the time." Another 30% indicated that they they collaborated for the earlier assignments. However, later in the semester, due to time pressures and individual members competency, they started splitting up the work. Only 10% of the groups indicated that they did not collaborate at all on the assignments.

This survey also indicated that 57% of the students found that the PL-Detective enhanced their learning experience. They felt that the tool allowed them "hands-on" application of concepts learned in class.

6.2 Subgoal 2: PL-Detective and Knowledge Discovery

We now discuss whether or not students were able to effectively use the PL-Detective as an information source.

6.2.1 *Methodology.* At the end of the Spring 2004 semester, we collected all submissions to the PL-Detective for the exercises described in Table I. Students worked on all assignments in groups of 2–3 members. There were 29 groups and group membership remained constant throughout the semester. All assignments in the course were submitted in groups and all members of a group received the same grade. The 11 PL-Detective exercises were spread out over 6 assignments, with one assignment per week. These exercises began

the third week into a 15-week semester.[5] Students had 1 week to complete each assignment. In addition, the exercises specified a limit of either the number of programs that students could submit or the number of PRINTs that they could execute (each PRINT outputs a single integer) without having their grade penalized. (Syntax errors did not count against the limit.) Such limits were intended to discourage a trial-and-error approach. If the input program to the PL-Detective fails (either at compile or run time) the PL-Detective reports only the first error message to the student. This fixes the amount of information students can derive from a single submission.

We derived the limits based on the nature of the exercise, the instructor's solution to the exercise, and the exercise's level of difficulty. Typically, we set the limit to be twice the number of attempts it took us to answer the exercise. For harder exercises we set higher limits, up to three times the number of attempts it took us to answer the exercise. In general, one can use the limit as an indication of the complexity of the exercise.

The PL-Detective records the sequence of submitted programs for each exercise and group. To determine whether PL-Detective was successful in meeting its second subgoal, we needed to know its characteristics as a source of information. Thus, we developed a coding scheme (available from the authors) that coded the explicit information given in the form of error statements and program output. We coded each submitted program based on whether or not error statements or program outputs were useful for answering the exercise. Our coding was conservative in that it considered information to be "useful" if the program outputs or errors provided useful information. With this coding it is possible that someone could extract some useful information from a "not useful" submission. For example, consider a submission that runs to completion, producing some output. If the output was not useful in answering the exercise, we labeled the submission as "not useful" although the fact that the program ran to completion may reveal some information relevant to the exercise at hand.

Two of the authors (the raters) trained in the coding scheme and independently coded the same randomly selected subset of the data (approximately 10% of all submissions). Interrater reliability was very high, with Cohen's Kappas ranging from 0.95–1.00. Given this high reliability, one rater then coded the remaining submissions. This methodology is commonly used in the social sciences [Folger and Poole 1981; Folger et al. 1984].

In addition to the above ratings we also used the feedback that the class TA gave to the students to determine whether a group was successful or unsuccessful in completing an assignment correctly.

6.2.2 *Results.* While working on the 11 PL-Detective exercises, groups submitted a total of 755 programs. Groups completed between 4 and 11 exercises ($mean = 10.3$) and submitted between 8 and 52 total programs ($mean = 26$). On average, students submitted 2.52 programs per exercise.

[5]A PL-Detective exercise appeared in Assignment 2 also, but we did not include it because it was designed just to get students familiar with the syntax of MYSTERY.

Table II. How Students Use the PL-Detective

	Had Output	Had Error
Useful information	301	81
No useful information	186	223

Table III. Getting Useful Information versus Getting Exercise Right

	Fully Correct	Not Fully Correct
Useful information	196	60
No useful information	8	32

Table II presents data on the 755 programs that students submitted to the PL-Detective. "Had output" means that a program produced some output. "Had error" means that a program did not compile or did not run to completion. Note that these two categories are not mutually exclusive: a program may produce output even if it does not run to completion. "Useful information" means that the program revealed information that was useful for doing the assignment. "No useful information" means that the program did not reveal information (according to our coding) that was useful for doing the assignment.

From this table we see that for programs that produced an output, 62% of them yielded useful information and for programs that produced an error only 27% yielded useful information. In other words, students were more effective at getting useful information from outputs than from errors. Overall, we found that 50% of the submissions produced useful information.[6]

Our past experience indicates that students often follow a trial-and-error approach to coding. One goal for the PL-Detective was to steer students away from such an approach toward a more principled experimental approach. Since such a large fraction of submissions yielded useful information, our results indicate that the PL-Detective accomplishes this design goal.

Did the Students Effectively use the PL-Detective as an Information Source? Just because half the submissions to the PL-Detective yielded useful information does not mean that students were actually able to use the information to get to a correct answer for the exercise.

We looked at the extent to which receiving useful information was related to whether or not a group answered a exercise correctly. We present the results in Table III. We used the label "Fully correct" to indicate that a group's answer to an exercise was completely correct. "Not fully correct" means that either the answer was wrong or only partially correct. We use the label "Useful information" to indicate that at least one of the group's submissions for an exercise yielded useful information. "No useful information" means that none of the group's submissions for an exercise yielded useful information.

From this table, we see that 196 out of 256 times (77%) when a group came up with a program that would yield useful information for an exercise the group

[6]Recall that we are conservative with marking submissions as useful: it may be that in reality more than 50% of the submission are useful.

was able to answer the exercise correctly. In other words, students are often able to effectively use the information obtained through the PL-Detective. It is worth remembering that just because a group had a submission that yielded useful information, it does not mean that they had *sufficient* information to fully answer the exercise.

Yet 60 out of 256 times (23%) students had useful information but were not able to capitalize on it to get a correct answer. We had expected that as students use the PL-Detective for more assignments they would improve their ability to capitalize on the information received from the PL-Detective. However, we did not find any evidence of this: the exercise that accounts for 15 of these 60 cases occurred late in the semester (Exercise 8 in Assignment 6). Exercise 8 requires students to distinguish between three choices. One of the cases was subtle and students frequently came to the wrong conclusion about it (even though they had useful information about the other two cases).

To generalize the observations for Exercise 8 above, we found that when the exercise asked students to distinguish between just two possibilities (Exercises 3, 4, 5, 7, and 11 in Table I), students were likely to get the exercise right even if they had just one useful submission. When there were more possibilities, students generally needed more than one piece of useful information to get it right.

Surprisingly, we found that eight times a group had no useful information on an exercise but still came up with the right answer. We examined these cases and found the reasons to be one of the following: (i) Our coding was conservative (focusing only on outputs and errors) so some submissions were marked as "not useful" although other elements of the submissions could be useful (five times); (ii) students used the web link of one PL-Detective exercise to answer a different exercise, thus we did not code some submissions for an exercise as being part of that exercise (one time); (iii) students were able to get information from previous exercises (one time). For example, when doing Exercise 2 (Table I) groups could gather enough information to also answer Exercise 3;[7] (iv) grader error (1 time). Reason (i) often indicates particular cleverness and creativity on the part of students and thus it is exciting to see!

Finally there were 32 exercise-group pairs where groups failed to get any useful information from their PL-Detective submissions and also got the exercise wrong. Nearly one-half of these happened in the first assignment (Exercises 1, 2, and 3) but the others were spread out over later assignments. Thus, we did not discern any obvious pattern.

Did the Students Efficiently use the PL-Detective as an Information Source? We now consider the exercise of whether or not students were efficient at using the PL-Detective as an information source.

For exercises limited by number of programs, Figure 5 gives the fraction of allowed programs that each group used. A point (x, y) with label e on this graph says that group x used a fraction y of the allowed program submissions

[7]We believe that Exercises 2 and 3 really deserve to be treated as a single exercise but to simplify them we had split them up into two. Some groups actually recognized and took advantage of this!

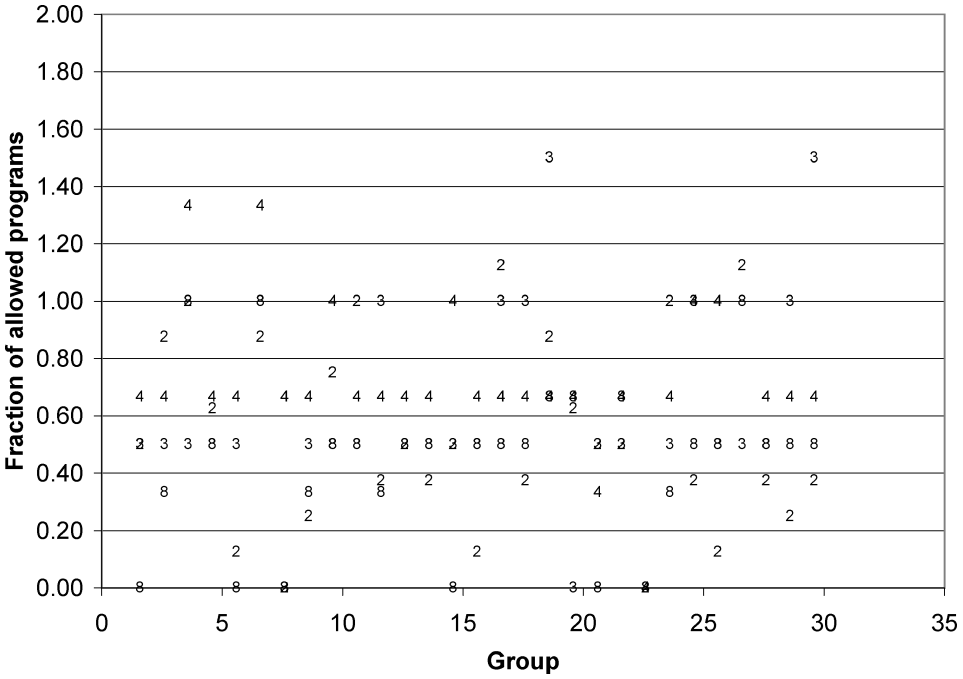

Fig. 5. Fraction of allowed programs used by groups.

for Exercise e. Points that are of the form $(x, 0)$ with label e indicate that group x did not submit any programs for exercise e. For exercises limited by number of prints, Figure 6 gives the fraction of prints that each group used. A point (x, y) with label e on this graph says that group x used a fraction y of the allowed prints for Exercise e. The points of the form $(x, 0)$ with label e mean one of two things: (i) group x did not submit any programs for Exercise e (6 instances); or (ii) group x submitted programs for Exercise e, but the programs did not produce any output (five instances). The second case includes one situation where the student group was able to get useful information without using any PRINTs. As an example of how the student did this, consider a situation where the value of a variable yields useful information and that the variable's value can be either 10 or 20. Rather than simply printing out the variable (and thus using up a PRINT attempt), one can use an IF statement to print only when the value was 10; in this way when the value was 20, they did not use any PRINT yet they got all the information that they needed! In other words, the fact that the program completed successfully without any output provided the student with useful information.

We clipped the y-axis of both Figures 5 and 6 at 2 to make them easier to read. Between the two graphs there were only four points above 2.

We see that, on average, students used a little over 50% of the limited attempts. Given that the limits were generally 2–3 times the attempts it took the instructor to do the assignment, students were only a little less efficient than the instructor. We also see that groups rarely exceeded the limit for an assignment (i.e., there are few points above the horizontal 1 line).

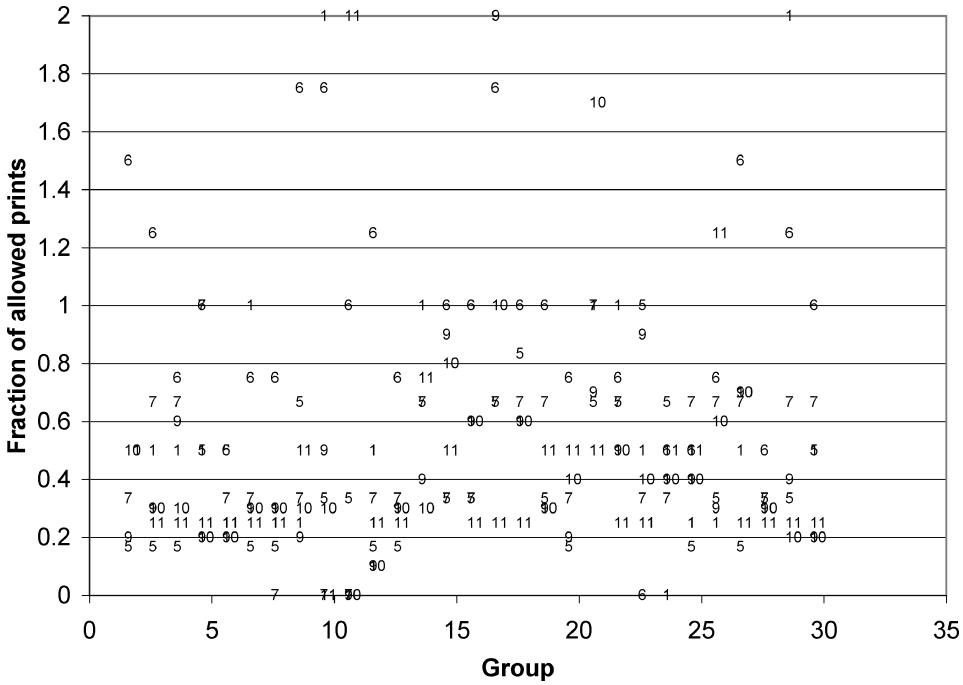

Fig. 6. Fraction of allowed PRINTs used by groups.

We found small but statistically significant correlations between the number of prints or submissions a group used on an assignment and whether the group completed the assignment correctly. First, there is a negative correlation between the percentage of allowed programs that a group used and whether the group answered the exercise correctly ($p = 0.05, r = -.218$); i.e., groups submitting fewer programs were somewhat more likely to get a correct answer. Second, there is a positive correlation between the percentage of allowed prints that a group used and whether the group got the exercise right ($p = 0.05, r = .155$); i.e., groups submitting more prints were somewhat more likely to get a correct answer.

To summarize, we found that the students were very efficient at using the information gained through the PL-Detective: on average they used about one-half as many attempts as they were allowed. In fact, we found cases where the student groups were extremely clever about gaining information without using up attempts.

6.3 Summary of Results

Our results indicate that the PL-Detective is effective in meeting both its subgoals and thus is likely to help students move beyond received knowledge.

7. LESSONS LEARNED

We now draw upon our experience using the PL-Detective and the data in Section 6 to highlight the strengths and weaknesses of the PL-Detective.

7.1 We Can Ask Harder Questions If Students Can Effectively and Efficiently Interact with an Environment to Get the Information They Need

PL-Detective questions are challenging for students because they are high in multiplicity. For each question there are numerous ways of approaching the problem and, at least for the language design exercises, there are many equally good answers.

For example, rather than asking "what will this program print with static and dynamic scoping" (which has only one correct answer), we were able to ask "write a program that distinguishes between static and dynamic scoping" (which has infinitely many equally good answers). However, in order for students to be successful in answering these harder exercises, the PL-Detective has to be a good source of information: students should be able to extract the information they need effectively and efficiently. Our results fortunately indicate that this was indeed the case. Moreover, our results indicate that there was no learning curve: in other words, students did not get any more effective or efficient over time. Thus, we hypothesize that the PL-Detective appeals to student intuition: they were able to use the PL-Detective well from the start. In future work we plan to collect data to test our hypothesis.

7.2 Allowing Options that Make No Sense Is an Effective Pedagogical Tool

Especially when using the PL-Detective for language design exercises and for exploration, the PL-Detective presents a very rich environment: the current snapshot of the PL-Detective supports 13,824 different semantics, some of which make sense and others that do not. We have found that allowing all combinations (rather than only the sensible combinations) provides for a great learning experience. Students who are using the PL-Detective for exploration often find that they need to experiment with picking the semantics before they encounter one that gives them the behavior that they expect. Since the involved issues can be quite subtle (often even we do not get it right first time!) being able to explore them experimentally is useful. This exploration often leads to a greater understanding and appreciation of the issues involved. We can talk all we want in class about these interactions but until students run into them on their own they often do not appreciate them.

7.3 Students Use Resources Efficiently

From Section 6.2 we know that students, on average, are only slightly worse than the instructor in terms of efficiently using the PL-Detective. In other words, students rarely use significantly more attempts than the instructor while doing PL-Detective assignments. We spent significant thought and effort into setting the limits. This is reflected in the fact that for some exercises we limited the PRINTs while for others we limited the submissions. However, it did not matter how the limit was set: students really tried to get the answer right in the fewest attempts possible. Thus, it seems that the very presence of the limit encouraged students to engage in a well-thought out process rather than using unprincipled trial-and-error approach to solving the exercises.

7.4 PL-Detective User Interface Still Needs Work

The current user interface to the PL-Detective, while it has been improved significantly since the first version, is still clunky. Students enter their programs in a html form's text area, which is not as rich an editing environment as students are used to. For example, students cannot use tabbing to indent their code and there are no facilities for automatically formatting the code. While one could imagine increasingly sophisticated web-based interfaces, we do not think that is the way to go. A better approach in our opinion is to write plugin for the Eclipse IDE. Students would enter their code in Eclipse, which provides one of the richest programming environments available today. The plugin would effectively wrap the student program and semantics selections in an html PUT message and communicate with the web-server running PL-Detective. In other words, students would not need to be aware of the web-based submission and its limitations. Tony Sloan at Macquarie University in Australia is already working on such an interface for the PL-Detective an expects to release it shortly.

7.5 Some of the PL-Detective Assignments Are Too Small and Should Be Combined

One of the main pedagogical goals of the PL-Detective was to involve students in experimentation: students submit programs and, based on their output, submit more programs. We observed this behavior for the exercises where students had to distinguish between 3+ possibilities (e.g., Exercise 9, Table I). But there were several exercises (e.g., Exercise 3 in Table I) where students had to distinguish between only two possibilities and thus did not engage in experimentation. It may be worth either extending the PL-Detective so that it gives more possibilities in the cases where there are only two or combining two exercises with two possibilities each into a larger exercises with more possibilities.

8. RELATED WORK

We are not aware of a system that has similar goals to the PL-Detective. However, while designing the PL-Detective and its associated exercises, we have been greatly influenced by the vast literature on learning and collaboration in the communication field. Two pieces of research that have been particularly influential in our work are Belenky et al. [1997] and Perry [1970].

The notion that student collaboration can improve the learning experience for students is well known. Alavi [1994] conducted an experiment where one-half of her class used GDSS (group decision-support system) and one-half did not. She found that the two halves performed similarly up to the finals, but the half that used GDSS performed better in the finals. Thus, she found a benefit to using collaboration in the classroom. Moreover, it took some time before students started exhibiting a benefit from the approach. Williams and Kessler [2000] found that using pair programming in class not only enhanced student enjoyment of the course, but also enabled them to perform better. Williams and Kessler also reported that it took some time before the pairs started working together effectively. While this prior work effectively tried to take the tools used in industry and apply them to the classroom, our approach has been to design

tools and strategies specifically for the classroom. We hope that our more direct approach will yield even greater benefits than observed in prior work.

Prior work on the Conversational Classroom [Waite et al. 2003] also aims to move focus away from received knowledge by involving students in collaborative learning. PL-Detective has some of the same goals, most importantly getting students to collaborate and thus learn from each other. However, while the conversational classroom focuses on collaboration in the classroom, PL-Detective focuses on collaboration outside the classroom.

The structure of the PL-Detective is not in itself novel. It is simply a compiler for a small language that has been carefully modularized along dimensions that we considered important for our class. The novelty of the PL-Detective comes from its focus on getting students to work together and providing an environment with which students can interact in order to discover knowledge. The bottom-line goal is to get students to move beyond the phase of received knowledge; i.e., to realize that there can be more than one right answer and that they can *create* knowledge.

9. SUMMARY AND CONCLUSIONS

We have described a system, the PL-Detective and a strategy to help students progress beyond the stage of received knowledge. The assignments supported by the PL-Detective have four key aspects. First, they are designed to reward group work and thus encourage students in a group to actually collaborate rather than segment the task into pieces that can be done independently. Second, they require students to interact with the PL-Detective in order to discover knowledge, thus proving to the students that they can create knowledge. Third, they are designed to directly expose students to the notion of multiplicity; that there is no one right answer and that their carefully reasoned argument is more important than the final answer. Fourth, since these assignments are structured as games, we hope that they will make a relatively theoretical class (concepts of programming languages) more fun for the students. Our results strongly suggest that the PL-Detective is effective in meeting its design goals: getting students to collaborate and involving students in interactions where they discover knowledge.

REFERENCES

ABERCROMBIE, M. L. J. 1960. *The Anatomy of Judgment; An Investigation into the Processes of Perception and Reasoning*. Hutchinson, London.

ALAVI, M. 1994. Computer-mediated collaborative learning: An empirical evaluation. *MIS Quarterly 18*, 2, 159–174.

ARNOLD, K., GOSLING, J., AND HOLMES, D. 2000. *The Java Programming Language*, 3rd ed. Addison Wesley, Reading, MA.

BELENKY, M. F., CLINCHY, B. M., GOLDBERGER, N. R., AND TARULE, J. M. 1997. *Women's Way of Knowing*. Basic Books.

DIWAN, A., JACKSON, M. H., WAITE, W. W., AND DICKERSON, J. 1995. Pl-detective: Experience and results. In *36th ACM Technical Symposium on Computer Science Education (SIGCSE)*. St. Louis, Missouri.

DIWAN, A., WAITE, W. M., AND JACKSON, M. H. 2004. PL-detective: A system for teaching programming language concepts. In *35th ACM Technical Symposium on Computer Science Education (SIGCSE)*. Norfolk, VA.

FOLGER, J. P., HEWES, D. E., AND POOLE, M. S. 1984. Coding social interaction. *Progress in communication sciences 4*, 115–161.

FOLGER, J. P. AND POOLE, M. S. 1981. Relational coding schemes: The question of validity. *Communication Yearbook 5*, 235–247.

GOSLING, J., JOY, B., STEELE, G., AND BRACHA, G. 2000. *The Java Language Specification*, 2nd ed. Addison Wesley, Reading, MA.

JENSEN, K. AND WIRTH, N. 1991. *PASCAL—User Manual and Report*. Springer Verlag.

KERNIGHAN, B. W. AND RITCHIE, D. M. 1988. *The C Programming Language*, Second edn. Software Series. Prentice-Hall, Englewood Cliffs, NJ.

LEONARDI, P. M. 2003. The mythos of engineering culture: A study of communicative performances and interaction. M.S. thesis, University of Colorado, Boulder.

MILNER, R., TOFTE, M., AND HARPER, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts.

NELSON, G., ED. 1991. *Systems Programming with Modula-3*. Prentice Hall, Englewood Cliffs, New Jersey.

PERRY, W. G. 1970. *Forms of Intellectual and Ethical Development in the College Years*. Holt, Rinehart and Winston, New York.

SEBESTA, R. W. 2003. *Concepts of Programming Languages*, 6th edn. Addison Wesley, Reading, MA.

SHAW, M. E. 1981. *Group Dynamics: The Psycholoogy of Small Group Behavior*, 3rd edn. McGraw Hill, New York.

SHAW, M. E. AND BLUM, J. M. 1965. Group performance as a function of task difficulty and the group's awareness of member satisfaction. *Journal of Applied Psychology 49*, 151–154.

STROUSTRUP, B. 1991. *The C++ Programming Language*, 2nd edn. Addison-Wesley, Reading, MA.

WAITE, W., JACKSON, M., AND DIWAN, A. 2003. The conversational classroom. In *34rd ACM Technical Symposium on Computer Science Education (SIGCSE)*.

WAITE, W. M., JACKSON, M. H., DIWAN, A., AND LEONARDI, P. 2004. Student culture vs group work in computer science. In *35rd ACM Technical Symposium on Computer Science Education (SIGCSE)*. Norfolk, VA.

WILLIAMS, L. AND KESSLER, R. 2000. Experimenting with industry's pair programming model in the computer science classroom. *Journal of Computer Science Education 10*, 4 (Dec.).

Received February 12, 2004; revised April 20, 2005; accepted April 25, 2005.