

PL-Detective: A System for Teaching Programming Language Concepts*

Amer Diwan
University of Colorado
Boulder, CO 80309
diwan@cs.colorado.edu

William Waite
University of Colorado
Boulder, CO 80309
waite@cs.colorado.edu

Michele H. Jackson
University of Colorado
Boulder, CO 80309
jackson@colorado.edu

ABSTRACT

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”). We believe that many computer science classes cater to 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 there are often multiple equally good answers and that discussion with students (rather than asking the instructor) is an effective way of learning how to reason.

Categories and Subject Descriptors

D.3.m [Programming Languages]: Miscellaneous

General Terms

Human Factors, Languages

Keywords

Concepts of programming languages, Education

1. INTRODUCTION

People go through a number of stages in their ability to learn [10, 3]. 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* [10]. In later stages subjects increasingly

*This work is supported by Any opinions, findings and conclusions or recommendations expressed in this material are the authors’ and do not necessarily reflect those of the sponsors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...\$5.00.

realize that information can come from within themselves; i.e., they can “create” knowledge. Proceeding through these stages brings the awareness that there may be many possible answers, each with its own merits and weaknesses.

Much of our computer science educational system caters to received knowledge: instructors spend most of the class period lecturing to students, interrupted only occasionally by questions. Assignments are often designed with particular answers in mind and thus there is little room for multiple “equally correct” answers. Students expect exams and assignments to test only material that the instructor has covered in class. A primary goal of our research is to discover and validate techniques that help students move beyond this stage, reducing their reliance on authority figures for knowledge and convincing them that most problems have a number of solutions.

One effective educational technique to reduce students’ reliance on authority figures is for them to learn from each other (i.e., “equals”) [2, 15]. This requires that they collaborate with one another, both inside and outside the classroom. Unfortunately, our findings from ethnographic observation and interviews with over 130 computer science and computer engineering students [8] show that they actively develop strategies to avoid collaboration. In order to get students to engage with one another collaboratively, professors must design assignments that clearly reward this type of interaction.

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 experience to bear. Thus collaborative decisions are often better than those made by an individual, and students learn the underlying material faster [1].

We have developed a tool, the *PL-Detective*, for building assignments and demonstrations in the context of a course titled *Principles of Programming Languages*. PL-Detective assignments reward true collaboration and directly expose students to problems with no clear “right” answer.

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 seven interfaces, each of which controls the semantics for a single aspect of MYSTERY. We provide at least two implementations for each semantic interface. Ev-

ery combination of implementations for the seven semantic interfaces defines a particular behavior for MYSTERY programs.

The PL-Detective supports two kinds of assignments, both of which encourage collaboration. The first, 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.

In this paper we summarize MYSTERY (Section 2), outline the semantic interfaces supported by the PL-Detective (Section 3), describe assignments using the PL-Detective (Sections 4 and 5) and outline the implementation of the PL-Detective (Section 6). Finally, we review prior work (7) and summarize the paper (8).

2. 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 [11] (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 [9] (which is based on Pascal [6] syntax). We picked Modula-3 syntax instead of the more popular syntax from the C family [7] 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 family
```

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.¹

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).

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 they do not introduce significant new semantic issues in MYSTERY. We included the

¹In 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.

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 (and more) can be explored with arrays (e.g., distinction between name and structural type equivalence). We omitted objects because many of the semantic issues with objects can be demonstrated by features already in MYSTERY. 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.

Finally, to illustrate the concept of scoping, MYSTERY allows nesting of blocks and function definitions.

3. SEMANTIC INTERFACES FOR MYSTERY

The PL-Detective exports seven interfaces, called 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 the l-value? To pick the semantics for the full MYSTERY one needs to pick an implementation for each semantic interface. The seven semantic interfaces are as follows; the numbers in parenthesis give the number of implementations that we already provide for the interface.

- Order of evaluation of parameters, which determines how and when to evaluate parameters to a call (2).
- Evaluation of logical expressions, which determines whether or not the evaluation is short-circuited (2).
- Parameter passing modes, which determines how to pass parameters (4).
- Scoping disciplines, which determines the binding of names (2).
- Type assignability, which determines the legality of assignments and parameter passing (3).
- Type equality, which determines when two types are equal (3).
- Type of, which determines the type of a constant or expression (2).

Multiplying out the implementations for each interface, we see that the current snapshot of the PL-Detective supports 576 different semantics for MYSTERY. Of course, not all combinations make sense (more on this in Section 5).

We have found it easy to add new implementations of semantic interfaces. The files for semantic interface implementations are typically 20-30 lines of 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 program, block, and procedure, it is not surprising that it is quite large.

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

Figure 1: Syntax of Mystery

4. 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. To interrogate the PL-Detective, student groups submit programs in MYSTERY syntax using a web interface (e.g., [4]). 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 reflecting 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 fixes the implementation of all interfaces that are not relevant to the assignment in question and tells students how they are fixed. For the relevant interfaces, there are two possibilities: (i) the instructor can either fix 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 copy-in-copy-out. Finally, the instructor needs to limit the number and kinds of programs that a group may use in its interro-

gation. The limit may be hard (e.g., the interrogation may not use more than 5 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 assignment designed to teach students about parameter passing modes. In this assignment, 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. More work remains to be done before the group is able to fully define the parameter passing semantics. For example, how does the language pass arrays? Does it pass just the base of the array as pass-by-value (ala C) or does it actually copy the entire array (ala Modula-3)?

When a student probes the system, the probe may or may not compile successfully (e.g., syntax error). If it compiles successfully, it may or may not run successfully (e.g., type-mismatch error). In the case of an unsuccessful compile or run, it is important to provide output that is useful to the students but is not so detailed that it solves the mystery. For example, imagine an assignment where students must discover whether MYSTERY uses static or dynamic scoping. Giving an error at compile time that a variable is undefined or out of scope would give too much information to the student about the semantics of MYSTERY. To address this situation, the web-based user interface for the PL-Detective delivers all error messages (except for syntax errors) at run

```

PROCEDURE f(x: INTEGER): INTEGER =
  BEGIN
    x := 20;
    RETURN 0;
  END;
PROCEDURE main(): INTEGER =
  VAR p: INTEGER;
  VAR q: INTEGER;
  BEGIN
    p := 10;
    q := f(p);
    Print(p);
  END;

```

Figure 2: Code to distinguish between parameter passing modes

time even for errors detected at compile time. We have also written the error messages so that they indicate the error (e.g., “Line 11: type mismatch”) without saying exactly what error was detected. There is a tradeoff involved here: if we put too much information into the error messages, we may reveal too much of the solution; if we reveal too little, we risk frustrating the students unnecessarily. We expect that we will revise the form and timing of error messages once we have some experience with the system.

The language analysis assignments have several properties relevant to our goals of helping students move beyond received knowledge. First, these assignments are complex and high on solution multiplicity [12] 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, different groups will most likely pick a completely different sequence of probes even if they arrive at the same conclusions. The nature of the task, therefore, resists segmentation, and the multiplicity rewards students that collaborate [13].

5. 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 note which 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 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). As before, the instructor needs to fix the implementation of

the semantic interfaces that are irrelevant to the assignment. Also as before, the instructor needs to limit the number of attempts allowed per group.

As with the language analysis assignments, the language design assignments also serve our goal of helping to move students beyond the stage of received knowledge. These assignments are similar to language analysis assignments in that 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 a 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 of the answers may be preferable to others based on objective considerations (e.g., soundness) or subjective considerations (e.g., taste). Thus, we expect the very nature of these assignments to reduce dualistic thinking amongst students, which is one of the key aspects of moving away from received knowledge. Prior work [13] suggests that greater collaboration leads to greater success in tasks with high solution multiplicity. Thus, we can expect these assignments to also reward groups that collaborate.

6. IMPLEMENTATION

The PL-Detective consists of two components. The first component is a compiler that processes MYSTERY programs. It is written in Java [5], and translates MYSTERY to Java.

The second component of the PL-Detective is the run-time system. It provides classes for representing data during program execution (e.g., the Closure class for representing procedure values or IntValue for representing integer values). The compiler and run-time system are 3200 and 300 non-blank, non-comment lines of code respectively. The semantic interface implementations contribute 515 to the line count of the compiler.

PL-Detective generates a closure for each MYSTERY function. The code component of each closure takes exactly one argument, which is a list of values. The environment component of each closure exports a method that maps names to values. PL-Detective represents values using dynamically allocated instances of the RTValue class and its subclasses. The above mentioned closures are also of a class (Closure) that is a subtype of RTValue. The different subclasses of RTValue export methods specific to the kind of value. For example, ArrayValue exports a subscript method that returns the RTValue residing at a given index in the array value. As another example, the Closure class exports a method that invokes the code portion of the closure. The run-time system defines RTValue, environments, closures, and their subclasses.

It is worth emphasizing that the PL-Detective places all values and environments on the garbage-collected Java heap. This organization gives us significant flexibility in implementing interesting semantics for MYSTERY. For example, since closures and environments reside in Java’s garbage collected heap, it is easy for us to support first class functions. If instead, we had used stack space for storing values of variables we would have had to incorporate complex mechanisms for copying and restoring the stack to support escaping func-

tions. Similarly, we can easily emulate different scoping mechanisms by changing the linking of environments.

7. 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 collaboration in the communication field. Two pieces of research that have been particularly influential in our work are Belenky *et al.* [3] and Perry [10].

The notion that student collaboration can improve the learning experience for students is well known. Alavi [2] conducted an experiment where half of her class used GDSS (group decision support system) and 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 [15] 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. We will be evaluating PL-Detective in a classroom setting starting Fall 2003.

Prior work on the Conversational Classroom [14] also aims to involve students in collaborative learning. PL-Detective work 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 to learn from each other. 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 by thinking through the issues and discussing them with their peers.

8. 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 three 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 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. Third, 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. We are currently using the system in class and hope to report on our results in the near future.

9. REFERENCES

- [1] M. L. J. Abercrombie. *The anatomy of judgment; an investigation into the processes of perception and reasoning*. Hutchinson, London, 1960.
- [2] Maryam Alavi. Computer-mediated collaborative learning: An empirical evaluation. *MIS Quarterly*, 18(2):159–174, 1994.
- [3] Mary Field Belenky, Blythe McVicker Clinchy, Nancy Rule Goldberger, and Jill Mattuck Tarule. *Women’s way of knowing*. Basic Books, 1997.
- [4] Amer Diwan, William Waite, and Michele Jackson. An infrastructure for teaching skills for group decision making and problem solving in programming projects. In *33rd ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2002.
- [5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java language specification*. Addison Wesley, second edition, 2000.
- [6] Kathleen Jensen and Niklaus Wirth. *PASCAL - User Manual and Report*. Springer Verlag, 1991. ISBN 0-387-97649-3.
- [7] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1988.
- [8] Paul M. Leonardi. The mythos of engineering culture: A study of communicative performances and interaction. Master’s thesis, University of Colorado, Boulder, 2003.
- [9] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice Hall, New Jersey, 1991.
- [10] W. G. Perry. *Forms of intellectual and ethical development in the college years*. Holt, Rinehart and Winston, 1970.
- [11] Robert W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, 6th edition, 2003.
- [12] M. E. Shaw. *Group dynamics: The psychology of small group behavior*. McGraw Hill, 3 edition, 1981.
- [13] M. E. Shaw and J. M. Blum. Group performance as a function of task difficulty and the group’s awareness of member satisfaction. *Journal of Applied Psychology*, 49:151–154, 1965.
- [14] William Waite, Michele Jackson, and Amer Diwan. The conversational classroom. In *34rd ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2003.
- [15] Laurie Williams and Robert Kessler. Experimenting with industry’s pair programming model in the computer science classroom. *Journal of Computer Science Education*, 10(4), December 2000.