RESULTS FROM USING AN ENVIRONMENT FOR INTERPRETER-BASED PROJECTS FOR THE PROGRAMMING LANGUAGES COURSES

Barbara M. Moskal\textsuperscript{1}, L. A. Smith King\textsuperscript{2}, and John Barr\textsuperscript{3}

Abstract—The Multiple Language Environment (MuLE) software tool is designed to support the teaching of programming languages using a combination of comparative and interpretive approaches. Accompanying the MuLE software tool is a user’s manual and a selection of pre-developed projects, scheme pre-labs, and testing programs. MuLE projects have been designed to vary in both size and complexity, allowing the instructor to select the appropriate level of classroom implementation. Formal assessment results indicate that students who used the MuLE software tool throughout a semester displayed a greater increase in content knowledge than did students who completed a traditional programming language course.

Index Terms—Assessment, Computer Science, Instructional Methods, Programming Languages

INTRODUCTION

Programming languages design and implementation is one of the seminal undergraduate courses in the computer science curriculum. A typical goal of this course is to introduce students to the process of writing code and thinking about the programming process. Throughout the programming languages course, students are encouraged to understand how the elements of programming languages can be used to create powerful abstractions and how these very same elements simultaneously constrain programming styles. After this course, students should no longer approach a problem as a program to be written, but rather as a solution to be built from multiple abstractions. In order to reach the goal of moving students from the process of writing code to reasoning like a computer scientist, a number of different instructional approaches have been used. These include the comparative approach, the interpreter-based approach and the hybrid-approach. A discussion of each approach follows.

Many programming languages courses convey the essential elements of languages by comparing and contrasting the semantic and syntactic features of a number of distinct programming languages. This comparative approach broadens the students' experience with the spectrum of language choices and reveals the common concepts that underlie most languages. This approach is built upon firm philosophical foundations, following Aristotle\textsuperscript{1} and Locke\textsuperscript{11} both of whom argued that specific experience is required in order to recognize general principles. Language features, however, cannot be explored in isolation. In particular, every construct provided in a language raises concerns with respect to how it will be implemented. This interdependency between theory and implementation is a characteristic of computer science as a discipline. Thus programming language design (theory) drives implementation, just as experience and computer architectures (empirical evidence) drive language design. The comparative approach exposes students to language diversity but when used alone the students may not recognize the relationship between implementation and language features.

An attractive alternative to the comparative approach has been the interpreter-based approach\textsuperscript{7, 10}. In this approach, programming language features are presented with the aid of an interpreter. An interpreter acts as a direct, complete and unambiguous explanation of the run-time semantics of the language. Building an interpreter gives the student direct, concrete experience in implementing language features. Examining interpreters for several languages assists the students in understanding common principles that are shared between languages. The interpreter-based approach exposes implementation concerns in a natural way. A difficulty with the interpreter-based approach is the danger of becoming bogged down in the minutiae of the implementation, and miss the larger issues. Furthermore, realistic interpreters are often too complicated. As a consequence, interpreter-based programming languages courses often implement syntactically and semantically simplified subsets of actual languages. Even this, however, can only be done if the instructor identifies a suitable language interpreter and attains access to the source code for modifications and extensions.

The MuLE software supports a hybrid of the comparative and interpreter-based approaches to teaching programming languages\textsuperscript{2, 4, 5}. MuLE consists of a launcher program, a utilities file, and a set of four interpreters for each of four major programming language paradigms: imperative, functional, object-oriented and declarative. MuLE permits students to easily experiment with a number of different language paradigms and to extend
or modify the interpreters for the languages to gain insight into the relationship between the implementation of languages and their use. The instructional support materials are designed to expose students to a variety of languages, language similarities and differences, and some parsing theory — all of which are consistent with the comparative approach. This material is supplemented by interpreter-based projects. Using MuLE to teach programming languages differs from a pure interpreter-based approach (e.g. [4]) in two important manners: 1) the interpreter projects are ancillary teaching tools, rather than the main focus of the course; and 2) since central language concepts are presented during instruction, the interpreter projects can be highly simplified while still being illustrative. The projects provide students with experience in relating language syntax and semantics with an implementation, but avoid the danger of bogging students down in an overly taxing interpreter implementation.

MuLE also has a flexibility that makes it useful in the broader curriculum context. For example, projects have been developed for the following instructional settings: 1) team oriented projects that support programming-in-the-large (PIL) and 2) projects that support programming-in-the-Context (PIC) of an existing program type projects. Advocacy for teaching team-oriented, PIL and PIC software development skills has always been present in curriculum discussions, but the skills were often cloistered in a software engineering course or as a capstone project. Currently, there is a great deal of literary support for the idea that these skills should be developed throughout the computer science curriculum [8, 12]. The ACM computing curricula 2001 also includes an emphasis on team skills and PIL skills into their new curriculum [9]. Furthermore, it is recognized that “education should not be constrained to development” and that “students gain valuable insight in the process of reading, deciphering, and evaluating the design and implementation of an existing software system”[12]. In summary, MuLE can be used to support other goals throughout the computer science curriculum.

The purpose of this paper is to describe the MuLE software as a teaching tool for supporting projects which enhances the students’ grasp of fundamental programming language concepts and that deepens the students’ appreciation of the relationship between language features and their implementation. MuLE is a software environment that supports interpreter-based projects for multiple programming language paradigms (e.g. object oriented, imperative/procedural, functional, logic programming). MuLE is written in DrScheme (from Rice’s PLT software project distributed under the GNU Library General Public License) and runs under Windows 95/98/NT/2000, MacOS, and Unix/X. The development of the MuLE software tool has been partially funded by a National Science Foundation (CCLI-DUE 9952398) grant [2].

In addition to describing the MuLE environment and providing a brief introduction to several different types of possible student projects using MuLE, we will also report the results of our assessment efforts. As will be discussed, two pilot studies were completed. The purpose of these pilot studies was to improve the MuLE software, the materials that accompanied the MuLE software and the assessment instruments prior to implementing the full study. The evaluation of the full study focused primarily upon determining whether the students in the treatment group (those that used the MuLE software and instructional materials) displayed a greater increase in understanding from the beginning to end of the semester as measured by a content assessment than did the students in the control group (those that did not use the MuLE software and instructional materials).

**MuLE Software and Classroom Implementation**

The following sections describe the MuLE architecture and the accompanying instructional support materials. The support materials include a user's guide, an instructor's guide and a variety of student projects.

**MuLE Architecture**

As was discussed earlier, MuLE consists of a launcher program, a utilities file, and a set of four interpreters for each of four major programming language paradigms: imperative, functional, object-oriented and declarative. The built-in interpreters are SLam (Simple Lambda language, a functional language), SOOP (Simple Procedural language, an imperative language), SOOp (Simple Object Oriented Programming language), and SLIC (Simple LogIC language, a declarative language). The interpreters are designed to share as much code as possible to facilitate students moving from one interpreter to another for projects.

MuLE is started by running the launcher that provides an interface for starting each of the interpreters. Upon launch, an interpreter displays a double pane window in which the user can enter, edit, and evaluate programs. A user can start any or all of the interpreters and can even start multiple copies of a particular interpreter.

Each interpreter is a separate module which consists of a parse function (or possibly a set of parse functions) and an execute function. Projects can be based on writing a particular function, a complete component (like the parser), or on extending the features of a language. Specific details of MuLE and example projects for functional and procedural languages are given in [2, 3, 4, 5, 6]. Each of MuLE’s interpreters provides a context for introductory, counter-example, PIC or PIL projects.
Support Materials

Along with the MuLE software, a variety of support materials have been developed. The MuLE software comes with a seventy-page user's manual. Each language interpreter supported through MuLE is discussed in its own chapter. An Instructor's Guide has also been created and contains fully developed projects, solution hints and answers. Several of the student projects available in the Instructor's Guide are discussed in the section that follows.

Student Projects

The developed projects vary in difficulty, length and coupling and have been designed to serve a variety of purposes. For example, introductory projects have been created with the purpose of familiarizing the students with the MuLE architecture. These projects might involve routines that could be tested independently of the MuLE system or that offered a step-by-step explanation and process for modifying a piece of code.

Another category in which the MuLE system has a variety of activities is the “teaching by counter-example”. These assignments highlight a single feature without overwhelming the students with too much information concerning the differences between real languages that employ that particular feature. For example, one assignment asks students to consider the design choice between dynamic or static scoping strategies. The languages that are associated with dynamic scoping (Lisp, SNOBOL and APL) look very different from the languages traditionally associated with static scoping (Pascal, C, Ada). Students may confuse the feature of interest with other language differences. Within MuLE, static and dynamic scoping can be taught using a single language, SLam Simple LAMbda language, MuLE’s functional language). SLam normally uses dynamic scoping but can easily be reconfigured or reprogrammed to use static scoping. Assignments have been developed that illustrate to students the conceptual and implementation differences that exist between the two forms of scoping. Several other counter-example features have been built into the various languages of MuLE.

Another type of project has been created that focuses on “programming within the context of an existing program”. This PIC type project forces students to expand or alter code that has been written by someone else. To complete the project, students must understand the overall design of a particular interpreter in order to implement a specific component. The MuLE framework has also been used as part of a capstone experience which exposed students to “programming in the large”. PIL projects require students to design and implement 500-1,000 line modules that interface with several other components of the interpreter/environment.

Another important component of the MuLE support materials is that approximately fifty MuLE language programs have been created to support the testing of students' work. These programs are accompanied by an expected outcome document for each program with different results depending on which interpreter options are implemented (i.e. Different program outputs would occur if SLam's dynamic scoping was implemented/turned on instead of static scoping). The availability of these programs simplifies the demands placed on the instructor in the grading process.

PILOT STUDIES

In order to fine-tune the assessment instruments, the MuLE software, and the MuLE classroom projects, two pilot studies were completed. This section briefly describes the results of these pilot studies and how they were used to improve the larger study. In other words, this section describes the formative evaluation of the MuLE software.

Pilot Study 1

In the spring of 2001, the MuLE software was piloted at Ithaca College. Throughout the semester, the participating teacher used MuLE as an instructional tool in an upper level computer science course. The students in the participating classroom completed an early version of the pre and post content assessment. The questions that were included on the pre and post content assessment had been randomly selected from a pool of questions that had been developed through a collaborative effort of the project investigators and the evaluator. Originally, there was no overlap between the questions that appeared on the original pre and post content assessment. No significant changes were found between students' performance on the pre and post content assessment during the pilot study. Based on this observation, the researchers re-examined both the classroom implementation of MuLE and the questions that comprised the pre and post content assessment. The researchers determined that the questions on the posttest were harder than the questions on the pretest. In order to control for this in the future, it was decided that the same questions would be used for both the pre and post content assessment.

Another result of this pilot study was that the participating instructor provided a number of suggestions for improving the MuLE software and the implementation of the MuLE software in the classroom. Based on these suggestions, assignments were re-written to improve clarity and features were added to the tool to make it more user-friendly. Additionally, the concern was raised that the shift in difficulty in the progression of the assignments was too rapid, resulting in student frustration. This resulted in the development of additional activities as well as variations on activities so the instructor could more easily tailor the assignment to the level and sophistication of the respective students.
Pilot Study 2

During the spring of 2002, the MuLE software was implemented in a programming languages course at Lycoming College. The participating teacher administered an end of the semester survey to students and provided feedback on her experiences in using MuLE in the classroom. The pilot teacher suggested that students needed more practice with the variant-record constructs to successfully complete later MuLE labs. More specifically, she suggested that the ability to deeply nest blocks in some of MuLE languages was problematic for some students because C++ does not allow this sort of nesting. She also said the pre-hobbled MuLE student packages that she had received were "a God-send". (The pre-hobbled package included interpreter skeletons that were ready to distribute to students for a pre-determined set of assignments.) The idea of pre-packaging interpreter skeletons already matched with particular assignments proved essential for instructors wishing to minimize assignment development time. Additionally, the students' responses to the end of the semester survey suggested that their overall experience with MuLE had been positive.

FULL STUDY -- METHODS

The information that was gained through the pilot studies was used to revise and improve the MuLE software prior to full implementation. This section describes the full study which compared the content knowledge of students that used MuLE throughout a semester (treatment group) to those that participated in a traditional programming languages course. In other words, this section describes the summative evaluation of the MuLE project.

Treatment and Control Group

The treatment group was a class of eleven students from the College of the Holy Cross, a four year private liberal arts college. The control group was a class of nine students from Sacred Heart University, a four year private university. Both schools are private, mostly residential, catholic institutions with small class sizes. Instructional techniques at the institutions are similar in that computer science courses at both institutions provide labs and stress hands-on learning of concepts in addition to conceptual and theoretical material. Freshmen at Holy Cross, on average, have higher SAT scores, whereas those at Sacred Heart are likely to have had more computer science exposure. Holy Cross supports a minor in computer science while Sacred Heart offers a major in computer science with either a computer science or information technology concentration as well as a master's degree in information technology. A primary difference between the treatment and control group is that the treatment group used MuLE as part of instruction and the control group did not.

Data Collection

Programming languages courses usually fall into the category of courses offered once a year or once every other year at smaller institutions or institutions with smaller programs. The time periods for data collection at each participating institution was dictated by when the given institution was offering the programming languages course. Data collection for the control group took place in the fall of 2001 and data collection for the treatment group took place in the spring of 2002. In the treatment classrooms, the MuLE software and instructional materials were used throughout the semester.

Content Assessment

A pre and post multiple-choice content assessment was completed in both the treatment and the control classrooms. This instrument was designed to assess students' knowledge in the following content areas: General Language Paradigm (3 questions), Grammars (3 questions), Interpreters (3 questions), Expression Evaluation (4 questions), Static versus Dynamic Scoping (2 questions), Parameter Passing Methods and Implementation (2 questions) and the Scheme Language (1 question). These content areas were selected because they were common to the goals and objectives of both the treatment and control classrooms. The specific questions that comprised this instrument can be found at http://cs.holycross.edu/~mule/Questions.html.

FULL STUDY -- CONTENT ASSESSMENT RESULTS

Table I contains the average proportion of questions the students answered correctly within each area as well as across all questions. As this table suggests, the students in the treatment group displayed a greater increase in knowledge as measured by the pre and posttest than did the students in the control group.

The above observation of the increased performance in the control classroom as compared to the treatment classroom was also tested for statistical significance. Due to the small number of students in each classroom, the decision was made to use an unpaired, one-tailed, t-test to compare the average difference in students' overall pretest and posttest scores between the two groups. Probability plots were used to verify that the normality assumption was met. The alternative hypothesis in this analysis was that the average difference would be greater in the treatment group than it was in the control group. A significant difference was found ($t = -2.28$, $p = .02$) at the $\alpha=.05$ level. The students in the treatment group displayed a greater increase in their average test performance than did the control group.
The control group. The reader is reminded that the content knowledge as measured by the content assessment than did displayed a significantly greater increase in content knowledge in the seven areas of interest. In fact, the students that used the MuLE software throughout a semester had a positive impact on students’ content knowledge in the seven areas of interest. Implementation of the MuLE software in the classroom, supports the assertion that using the MuLE software

| TABLE I |
|-----------------|-----------------|
| RESULTS OF CONTENT ASSESSMENT |  |
| | Control n=9 | Treatment n=11 |
| | Pre | Post | Pre | Post |
| General Language Paradigm | .41 | .59 | .45 | .60 |
| Grammar | .33 | .56 | .39 | .94 |
| Interpreter | .15 | .33 | .42 | .55 |
| Expression | .83 | .89 | .91 | .98 |
| Evaluation | .39 | .33 | .23 | .77 |
| Static vs. Dynamic Parameter Passing | .33 | .39 | .65 | 1.00 |
| Scheme | .11 | .33 | .09 | .73 |
| Total | .42 | .54 | .52 | .80 |

SUMMARY

Programming languages courses should expose students to a diversity of languages, explore programming language similarities and differences, and discuss language implementation issues. In order to accomplish these goals, a combination of comparative and interpreter-based approaches to instruction is needed.

MuLE provides a software environment that is a hybrid of the comparative and interpreter-based approaches — making it a natural choice for teaching programming languages. As was discussed, MuLE has a complete user’s guide and instructor’s manual. A variety of student projects and programs for testing students’ solutions have also been developed. The programming language topics covered through the support materials include: interpreters, grammars, memory binding, expression evaluation, language paradigms, static versus dynamic scoping, parameter passing mechanisms, class implementation, unification with respect to logic languages and the scheme programming language. Students explore these topics through the completion of projects. These projects are designed to utilize the MuLE interpreters, which are available in each of the four major paradigms.

The MuLE software and support materials have been carefully developed with continuous attention to both formative and summative assessment activities. The formative assessment included two pilot studies and the information that was acquired through these studies was used to improve both the MuLE software and the support materials. The results of the summative evaluation, which was completed in conjunction with the formal implementation of the MuLE software in the classroom, supports the assertion that using the MuLE software throughout a semester had a positive impact on students' content knowledge in the seven areas of interest. In fact, the students that used the MuLE software throughout a semester displayed a significantly greater increase in content knowledge as measured by the content assessment than did the control group. The reader is reminded that the content assessment was developed with the purpose of measuring important concepts that were common to both courses. We believe that these results support the assertion that the MuLE software can and has been used as an effective tool for supporting student learning in programming languages courses. Our future efforts will be devoted to exploring the possibilities of developing MuLE projects in a broader array of content areas.

REFERENCES


