Session 3520

Engineering Programming Language Concepts

Holly Patterson-McNeill, Carl Steidley
Texas A&M University-Corpus Christi

Abstract

The study of programming languages is beneficial to all levels of programmers. The first part of this paper reviews some of the reasons for studying programming languages. To isolate some of the issues of language design, definition, and implementation, mini-languages have been used in Programming Languages courses. Mini-languages are small and complete, yet restricted languages. They have a small syntax and simple semantics. Mini-languages and their compilers are being successfully used as a basis for a Programming Languages course.

1. Why Study Programming Languages?

The study of programming languages is beneficial to all levels of programmers whether they be computer science students, engineering students, or computer engineering students. The nature of the work done by graduating students requires that they be familiar with at least one programming language. Yet, this language will probably not be the one they actually use on the job. By studying programming language concepts, students can gain an increased capacity to express ideas, an improved background for choosing appropriate languages, an increased ability to learn new languages, a better understanding of the significance of implementation, and an increased ability to design new languages[6]. Programming languages are tools and a tool needs to be fully understood before it can be used properly[4].

Hand in hand with the linguistic theory[7] that one's language has a considerable effect on the way that one thinks is the influence of a programming language on the class of solutions we are likely to use. The language in which programs are developed places limits on the solutions we are likely to see and the kinds of control structures, data structures, and abstractions we can use. Thus, the forms of algorithms we construct may take different forms in different languages. By learning new language constructs, we can increase the range of our software development thought processes.

If one is given a choice of languages for a new project, it is natural to continue to use the language with which we are most familiar, even if it is poorly suited to the new project. If we were familiar with the capabilities of other languages, we would be in a better position to make more informed language choices.

Learning a new programming language can be lengthy and difficult. A thorough understanding of the fundamental concepts of programming languages can facilitate the understanding of the new language, allowing us to see how these concepts are incorporated into the design of the language being learned.

When learning the concepts of programming languages, it is necessary to look at the implementation issues that affect those concepts. An understanding of implementation tradeoffs can lead to an understanding of why languages are designed the way they are, which leads to the ability to use a language as it was designed to be used. Another benefit of understanding implementation issues is that it allows visualization of how a computer executes various language constructs. This in turn fosters an understanding of the relative efficiency of features chosen for a program.

Although having to design a new programming language may seem remote to most programmers, we often design the interface to programs. The interface is, in a sense, a kind of programming language. The criteria for judging that interface are similar to the criteria used to judge the design of a programming language. A critical examination of programming languages, therefore, will help in the design of such complex systems, and it will help users examine and evaluate such products. For those programmers who must design new languages, it is helpful to learn from the successes and failures of the designs of the past.

2. Mini-Languages as a Pedagogical Tool

Our ability to understand English does not give us the ability to understand French or German, although all three languages are based on similar principles because of their common Indo-European origin. English has a very tolerant grammar, which causes many English-speaking students to have problems with languages with more rigid grammars. Grammatical concepts that are only slightly used in English, such as the subjunctive tense, are important in other languages, such as French, and need to be understood before that language can be mastered[5]. Yet both languages allow the same basic ideas to be communicated.

The situation is very similar with programming languages; they differ widely in their external forms and range of facilities, but they are based on a relatively small group of basic concepts. The proliferation of programming languages has raised many issues of language design, definition, and implementation. In his classic paper, Ledgard[3] suggested the use of mini-languages to address these issues. The book by Marcotty and Ledgard[5] followed this paper, in which they use mini-languages to study language concepts in isolation and then seek the implementation of these concepts in real languages. Brusilovsky[1] and Krishnamurthi and Felleisen[2] describe the use of mini-languages to teach programming, problem solving and algorithmic thinking.

An immediate problem encountered in teaching a Programming Languages course is the complexity of most languages. An attempt to isolate important language features requires a good deal of study. Some Programming Languages courses try to study two or three complete languages--an imperative one, an object-oriented one and a functional one. The study of two or three complete languages can be overwhelming and confusing.

It is unfortunate that despite the large number of programming languages there are few accepted principles contributing to an existent theory of language design. MacLennan[4] argues for nineteen such principles, but admits that the different uses and users of programming languages, as well as computers, on which languages are implemented, require tradeoffs in those design principles. Mini-languages can be constructed to highlight these principles.

Mini-languages are small; they have a small syntax and simple semantics. They are complete, although restricted, languages in themselves that allow the languages to focus on a few concepts. A mini-language may be a subset of an existing language or a small language in its own right. Their value lies in their brevity of description and the isolation of important linguistic features.

Mini-languages allow the instructor to raise important issues in the area of formal definition of programming languages. The limited variety of syntactic and semantic content in such languages allow for a better focus on the acceptability of proposed strategies for dealing with the issues contained within the mini-language.

Mini-languages also can be used to emphasize some of the difficulties of language implementation. For example, a mini-language on generalized transfer of control poses the difficult issue of linking identifiers with proper values; a mini-language on type checking poses the problem of recognition of conditions that lead to program errors; a mini-language on string manipulation raises the question of efficiency in the algorithmic recognition of strings defined by a generative grammar.

None of the mini-languages in Marcotty and Ledgard[5] are exact subsets of existing programming languages although much of the notation and semantic material resembles portions of existing languages. They emphasize such features as the notions of assignment, transfer of control, functions, parameter passing, type checking, data structures, string manipulation, and input/output. Many important features of existing languages are omitted, including interrupts and events in real time, file and storage management, and simulation. Each of the languages are presented in the following format: a brief introduction to a topic in programming languages; an English description of the language covering the topic; several example programs in the mini-language; a discussion of the mini-language and its relation to issues in current programming languages.

There are several disadvantages to mini-languages. The foremost is that they are not "real" programming languages. They are not and will never be practical, accepted languages. Students also voice some frustration with not learning about current programming languages. Most of these mini-languages emphasize imperative programming concepts; object-oriented techniques are limited to one mini-language that illustrates separately compiled modules.

With the advantages in mind and despite the few disadvantages, the authors believe that mini-languages are a valid approach for the teaching of a Programming Languages course.

3. Mini-Languages as a Basis for a Programming Languages course

Rather than study two or three complete languages in one semester, we use mini-languages to illustrate language concepts. In addition to the mini-languages of Marcotty and Ledgard, we use a version of early pseudo-code[4], a primitive language, and DrScheme[2], a subset of Scheme and methodology developed at Rice University. These languages are simple to use. Because there are multiple languages, the instructor can pick and choose among them to emphasize the desired concepts. Such features may be assignment, control, functions, parameter passing, type checking, data structures, and input/output, issues of formal definition, difficulties of language implementation, and alternative programming paradigms.

The first mini-language studied is a pseudo-code. This pseudo-code is not the informal program design notation, but is a primitive, interpreted language. It is used to illustrate many of the steps and decisions in the design of a programming language. From a machine language to a symbolic (assembly) code, the students refine the language into something similar to what they have used in their Computer Organization and Assembly Language course. This step-by-step refinement illustrates the decisions made in language design and the pitfalls inherent in improvised and unpremeditated changes to a design.

The second mini-language studied is DrScheme. The DrScheme materials from Rice include a freeware version of the development environment, a manual, and programming exercises, all available on the Web. These exercises have been developed for use in a semester long introductory problem solving course, and are therefore appropriate for teaching a new programming paradigm. The use of DrScheme allows our students to study functional language concepts, an unfamiliar topic since our university uses either C or C++ for program development in other courses.

Next, the mini-languages from Marcotty and Ledgard are used. The programming exercises for these languages had been paper and pencil exercises. The students of our previous Programming Languages course have expressed frustrations in not being able to test their programs. The authors felt it would be better if these exercises were machine-implementable. Therefore several of the mini-languages now have interpreters developed by our students in our compiler construction project classes. Student satisfaction with the mini-languages has improved and the languages seem more viable and less theoretical.

The first Marcotty and Ledgard mini-language is Core. It is a simple imperative language that supports only integers in the range from 0 through 99999. It has a limited set of statements--assignment, single and double alternative decision (if-then and if-then-else), looping (while), and simple input/output. The students study informal descriptions of the syntax and semantics of Core and program samples while they become accustomed to learning a language from its context free syntax in Backus-Naur Form (BNF). This restricted language gives the students time to grapple with formal syntactic definitions--BNF, Cobol notation, and syntax charts--as well as formal semantic definitions--operational, denotational, and axiomatic semantics--without having to worry with too many language constructs.

Their next language is called Control. Again the students study the language from its BNF and program samples. The mini-language Control builds on Core to implement exiting from a loop (exit), unconditional transfer (goto), multiple alternative decisions (if-then-elseif-else and case), and counting loops (for). The language for Procedures gives the students practice with passing parameters using the familiar modes of pass-by-value, -result, -value-result, and -reference, as well as with passing parameters by an unfamiliar mode called pass-by-name. Subsequent languages provide additional data types, formatted input and output, structures, recursion, exception handling, and parallel processing.

The well-defined structures and small instruction sets of these mini-languages make them easy for the students to understand. They provide concrete examples of unusual (from the student perspective) language design solutions and useful comparisons with the languages the students already know.

Some of the programs and homework problems that students must complete include implementation of floating point arithmetic using only integer arithmetic, translating spaghetti code into structured code, and using guard statements. Students get to practice with language constructs not available or familiar to them in the languages they had learned in their previous courses. Their repertoire of problem solving techniques is expanded and their facility with new languages is improved. Because the students have to write simple programs using the mini-languages, they learn that even these small languages are useful for more than just teaching.

4. Conclusions

There are many benefits of studying programming language concepts. Among these are an increased capacity to express ideas and implement solutions, an improved background for choosing appropriate languages, an increased ability to learn new languages, a better understanding of the significance of implementation, and an increased ability to design new languages. An immediate problem encountered in teaching a Programming Languages course is the complexity of most languages. The use of mini-languages with their restricted syntax and semantics aids in this instruction. This approach has been successfully used in the Programming Languages course at our university.

The current list of mini-languages is incomplete for a thorough study of programming language concepts. The extensions needed include a mini-object language and a real-time language. A more complete set of interpreters/compilers is needed. These compilers will be generated in the Compiler Design course that is the follow-up to our Programming Languages course. These languages provide the basis for projects in other courses, including human-computer interaction, technical writing, and software engineering. The compilers will need to be supplemented with user-interfaces that will facilitate teaching, debugging, and programming. A full set of user documentation especially for error assistance will need to be generated.

Bibliography

1. Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., and Miller, P. Mini-languages: A Way to Learn Programming Principles, *Education and Information Technologies 2* (March 1997), 65-83.
2. Krishnamurthi, S. and Felleisen, M. Tutorial: Innovations in Introductory Computer Science Curricula, Presented at the Consortium for Computing in Small Colleges (CCSC): 10th Annual South Central Conference, St. Edward's University, Austin, TX, (April 16-17, 1999).

3. Ledgard, H. Ten Mini-Languages: A Study of Topical Issues in Programming Languages, *Computing Surveys* 3 (September 1971), 115-146.
4. MacLennan, B. J. *Principles of Programming Languages, 3rd Edition*, Oxford University Press, Inc., New York, 1999.
5. Marcotty, M. and Ledgard, H., *Programming Language Landscape, 2nd Edition*, Macmillan Publishing Company, New York, 1986.
6. Sebesta, R. W., *Concepts of Programming Languages, 2nd Edition*, The Benjamin Cummings Publishing, Inc., 1993.
7. Whorf, B., *Language Thought and Reality*, MIT Press, Cambridge, MA, 1956.

HOLLY PATTERSON-MCNEILL
Holly Patterson is an Assistant Professor of Computer Science at Texas A&M University-Corpus Christi. Dr. Patterson-McNeill received a BA in Mathematics from the University of Texas at Austin in 1970, an MS in Computer Science from the University of Texas-San Antonio in 1977 and a Ph.D. from the Department of Computer Science at Texas A&M University in 1996.

CARL STEIDLEY
Carl Steidley is a Professor of Computer Science and Department Chair for Computing and Mathematical Sciences at Texas A&M University-Corpus Christi. Dr. Steidley received BA in Mathematics from California State University, Northridge in 1971, an MS in Mathematics from California State University, Northridge, and a Ph.D. in Computer in Education from the University of Oregon in 1983.