Experience with an Analytic Approach to Teaching Programming Languages

Christopher T. Haynes
Computer Science Department
Indiana University
Bloomington, IN 47405
chaynes@indiana.edu

Abstract

Through the use of interpreters it is possible to teach programming languages in an analytic way without the mathematical overhead associated with other formal methods. This is a natural evolution of programming language pedagogy from present approaches that are largely descriptive. As a bonus, students receive training in the use of meta-linguistic abstraction in program design. An example of this approach is presented along with suggested variations and discussion of a number of practical considerations that may be crucial to success in the classroom.

Introduction

Early texts in the study of programming languages were descriptive of a variety of programming languages. By surveying a variety of language types they provided exposure to a number of language principles, but they were organized around languages, not principles.

The next generation of programming language texts were organized around underlying principles, but they were still descriptive. Principles were described informally (in natural language) and illustrated with related features in well known programming languages. This was an improvement, and almost all programming language texts intended for undergraduates are still of this kind. Sethi [11] is a modern example of this approach, while MacLennan [9] combines the language survey and principle-based approaches.

Programming languages generally have a large number of features that are for most purposes adequately treated by such descriptive techniques. Yet the most crucial features of programming languages, those that determine the kinds of program structuring techniques which they support, are often very subtle. Experience has shown that casual descriptions may fail to unambiguously convey the meaning of common features such as recursion and parameter passing. This has prompted the development in the last quarter-century of an extensive theory of programming languages, based on rigorous mathematical techniques such as denotational semantics and structural operational semantics.

Recently a variety of textbooks have appeared based on such analytic techniques, such as Gunter [4] and Winskel [15]. Unfortunately the mathematical sophistication required for the use of formal semantic techniques makes them unsuitable for most undergraduates. In addition, technology to simulate the behavior of programming languages based on such formal specifications is generally unavailable, and always complex. Thus highly tedious and error-prone formal computations are required to develop familiarity with the consequences of semantic definitions.

This paper considers an alternative analytic approach to teaching programming languages semantics that is based on interpreters. Semantic principles are studied by implementing them. As we shall see, this often allows programming language semantics to be expressed with close to the precision and concision of formal semantics, but without the mathematical overhead, and with the added advantages of a running implementation.

There appear to be several reasons why this approach is not widely used in teaching the principles of programming languages. Implementation of programming languages, or even semantically interesting components of traditional languages, is generally considered to be difficult. Also, implementations are generally of sufficient complexity that they fail to shed light on the semantics of the language being implemented: implementations are seldom compact or clear. Finally, even under the most favorable circumstances formal methods take more time to present than do informal treatments. Thus breadth must be sacrificed for depth, and the programming language landscape is broad indeed.

Let us address the latter issue first. We believe strongly that in the study of programming languages depth of understanding of the most basic and widely-used principles is far more important than a broad survey. The surface detail of programming languages is easily mastered through independent study if the student has been firmly grounded in funda-

[0] To appear in SIGCSE'98. Copyright © 1997 by Christopher Haynes

mentals, while the converse is not true. If fundamentals are not learned as part of a student's formal education, they often will never be learned. It is important, though, that study in depth be practical and not remain a theoretical exercise.

The remainder of this paper addresses the former difficulty: the need for clear and concise formal specification, without mathematical overhead. This is possible by the careful use of interpreters to implement mini-languages. This report is based on experience for more than fifteen years with this approach in the junior/senior level programming languages course required of all CS majors at Indiana University (Bloomington). Variations on this approach have been used in a number of courses elsewhere, from large research institutions (such as Iowa State, Northeastern, Rice, and the University of Illinois) to small liberal arts colleges (such as Oberlin and Indiana University at South Bend).

For reasons we shall discuss, the interpreter tradition is particularly strong in the histories of Lisp and Scheme, with interpreters playing a central role in the seminal works on both languages. [7, 12] At least two programming languages textbooks are based substantially on the use of interpreters [3, 8], and several introductory Scheme texts include simple interpreters [1, 2, 10].

In the next section we give an example of such an interpreter. Next we suggest some of the wide variety of semantic variations that are possible using interpreters and related programs. Then we review practical considerations that may be critical in teaching with this approach. Lack of appreciation for such issues has, we believe, limited broader acceptance of this approach. Student responses are then reviewed and the value of such training in the larger context of program design is discussed. We conclude with remarks on our NSF sponsorship and an historical perspective.

An Example

In Figure 1 we present the heart of an interpreter for a mini-language that includes a number of the most essential semantic elements of popular languages. The host language in which this interpreter is expressed is Scheme, a modern dialect of Lisp. Though a number of other languages can, and have, been used to express interpreters for pedagogic purposes, the choice of Scheme has major advantages (as well as a few liabilities). We discuss these issues in detail later. It is not necessary to know Scheme or Lisp in order to gain an impression of this approach from the general description provided in this section.

The procedure `eval-exp` (evaluate expression) takes an expression, `exp`, and an environment, `env`. Expressions are represented as a form of structured data (Lisp s-expressions) for which Scheme provides extensive built-in support. Environments are represented via an abstract data type (ADT) whose straightforward definition (omitted here) includes the procedures `apply-env`, for environment lookup, and `extend-env`, which builds a new extended environment.

```
(define eval-exp
  (lambda (env exp)
    (if (variable? exp)
      (cell-ref (apply-env env exp))
      (record-case exp
        (quote (datum) datum)
        (if (test-exp then-exp else-exp)
          (if (true-value?
                (eval-exp (test-exp) env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
        (set! (var exp)
          (cell-set! (apply-env env var)
            (eval-exp exp env)))
        (proc (formals body)
          (make-closure formals body env))
        (else
          (let ((proc (eval-exp
                        (app->operator exp)
                        env))
                (args (eval-operands
                        (app->operands exp)
                        env)))
            (apply-proc proc args)))))))

(define apply-proc
  (lambda (proc args)
    (record-case proc
      (prim-proc (prim-op)
        (apply-prim-op prim-op args))
      (closure (formals body env)
        (eval-exp body
          (extend-env formals
            (map make-cell args)
            env)))
      (else
        (error "Invalid procedure:" proc)))))
```

Figure 1: A simple interpreter

Expression evaluation begins with a dispatch on the syntactic form of the expression. In this mini-language, an expression is either a variable reference, or a conditional, assignment, literal, or procedure expression introduced by an associated keyword (`if`, `set!`, `quote`, or `proc`, resp.), or a procedure call. The `record-case` form, an easy-to-define syntactic extension of Scheme, plays the part of a customary variant-record case dispatch, with binding of local variables to the contents of the selected record's fields.

For example, the if case has fields for the test, then, and else subexpressions. When this case is selected, the test expression is evaluated in the current environment (via a recursive call to `eval-exp`) to determine whether the then or else (consequent or alternative) expression is to be evaluated to obtain the value of the entire expression.

To support variable assignment in the mini-language, variables denote cells. A very simple cell ADT supports cell dereferencing and assignment (via the `cell-ref` and `cell-set!` procedures, resp.).

The record-case else clause handles procedure calls. First the operator and operands are evaluated in the current environment to obtain a procedure and its arguments. The procedure is then applied to the arguments. The procedure `apply-proc` dispatches on the type of the procedure, which may be either a primitive supplied in the initial environment (such as + or `*`) or a closure.

Closures result from evaluating a procedural (`proc`) expression. They are simply a record containing the formal parameters and body of the procedure and the environment in which the procedure was created. It is this environment that is extended when the closure is applied.

Variations

A variety of exercises are possible to broaden students' appreciation for the semantic richness of this simple program. For example, the representation of closures may be changed from a record to a host-language closure. This involves replacing the call to `make-closure` with an appropriate lambda expression and a corresponding change in `apply-proc`. These changes result in an interpreter that is a bit more concise and elegant, but initially harder for students to comprehend. This exercise does much to clarify the true meaning of closures (higher-order procedures).

In another exercise, students may be asked to pre-process the expression being evaluated so that lambda expressions are annotated with a list of variables appearing free in their bodies. This allows an alternate "display" representation of closures in which only the cells associated with free variables are captured in the closure, rather than an entire environment. Display closures are close in their run-time representation to objects in languages such as C++, with free variables in the role of instance variables.

Another exercise is to change the environment ADT so that it supports assignment directly. This avoids the need for a separate cell ADT and may be somewhat more efficient, but prohibits the display representation of closures.

It is good practice to stage the development of interpreters. That is, start with a very elementary mini-language and gradually add features. In working up to our example we might have started with just variables, literals, and procedure calls. Adding conditional statements is an elementary step that builds student confidence. Next we add user-defined procedures (`lambda`), and finally assignment. In this way the semantic elements of each new feature are clearly evident. For example, this makes clear that environment bindings need not denote cells until assignment is added.

The interpreter presented here, or one similar to it, provides the basis for a great many further additions. Variations on parameter-passing mechanisms such as call-by-value, call-by-reference, call-by-name, and call-by-need are quite instructive and relatively straightforward. A variety of parameter-passing variations can be achieved simply by changing auxiliary function definitions if one starts with a sufficiently abstract interpreter. [3]

Modeling such object-oriented programming features as static and dynamic method binding, instance variables, inheritance, self reference, and even meta-level shifts are possible with somewhat (but not substantially) more complicated interpreters. [3] Other syntax-directed forms of program analysis, such as static type checking [6] and compilation [5], may be accomplished in a remarkably similar style.

It is also possible to learn much by varying the style of an interpreter while maintaining the same interpreted language. The continuation-passing style (CPS) transformation is of particular interest. It removes control ambiguities in the semantics of the interpreted language that were previously resolved by the host language, allows non-standard control operations such as non-local breaks to be easily added to the interpreted language, and highlights tail-calls. [3] Another variation models the store (assignable memory) directly as an object that is passed dynamically in the interpreter. If one combines these two transformations with lazy evaluation (either simulated in a non-lazy language such as Scheme, or directly using a lazy language such a Haskell), and obeys the principle of compositionality, one obtains interpreters that are in the form of traditional denotational semantics.

Practical Considerations

The language used to program an interpreter has a dramatic effect on the interpreter's form and its suitability for teaching purposes. Automatic storage management (as in Scheme, Lisp, ML, Haskell, or Java, but not C++, Ada, or Pascal) is almost essential unless the interpreter is low-level (e.g. stack-oriented). Higher-order procedures (as in Scheme, ML, Haskell, (with inconvenience) Common Lisp, or (using inner classes) Java 1.2, but not C++, Ada, or Pascal) allow more abstract interpreter formulations, as noted above, but are not required for more "data structure" oriented interpreters, such as our example. The built-in pattern matching provided by ML and Haskell is a substantial advantage, but a simple syntactic extension in Scheme or Lisp, such as our `record-case`, serves almost as well.

The syntax of the interpreted language is of little importance in learning programming language fundamentals (which are almost entirely semantic), but syntax is of potentially great importance in its effect on the interpreter's form. In a one semester course there are so many semantic issues of importance to cover that we spend almost no time on syntactic issues. This is possible by taking advantage of the built-in support for symbols and structured data associated with s-expressions in Scheme and other Lisp dialects. The need for a parser is entirely avoided by choosing syntax for the interpreted languages that is subsumed by the

s-expression syntax, as in our example. The built-in reader is the parser. A number of other built-in procedures operating on such structured data (such as `map` in our example) simplify interpreters written in Scheme and other Lisp dialects. Debugging is also facilitated, since there is a built-in printer for s-expressions. If it is desired to support more traditional syntax, a parser that produces syntax trees in the form of s-expressions may be used (as in [3]).

In the design of programming languages it is now customary to define a core syntax, which cannot reasonably be translated into a smaller syntax, and derived syntax, which may be transformed into core syntax. For example, the mini-language of our example has literals (`quote` expressions), but no constants such as numbers that are not quoted. This is justified because constants are derived syntax that is translated into an equivalent literal (`3` becomes `(quote 3)`). A more interesting derived form is non-recursive local binding (`let`), which may be transformed into a procedure call. It is a valuable exercise to implement such transformations with a syntax-expansion procedure that is called prior to invoking the interpreter.

Another syntactic transformation, mentioned earlier, annotateded lambda expressions with a list of their free variables. A related exercise replaces each variable reference with its lexical address: the position of its referent in the environment (computed statically). Of course the interpreter is then modified to use this information. Though the system still employs interpretation, it then has a run-time architecture that is closer to that of a typical compiled implementation. (Compilers routinely calculate lexical addresses.) Such basic run-time issues are sometimes discussed in traditional programming languages courses, but unfortunately direct experience with them is typically limited to compiler courses.

Student Response

Overall student evaluations for this course are typical of courses in our major, with a few exceptionally enthusiastic students every semester. Yet the difficulties students face in this course and the post-graduation benefits are not typical.

Recursion is fundamental to the structure of interpreters. If students do not come to the course with a strong background in recursion (and frequently even when it seems that they do), much of the difficulty experienced with this approach may be traced to weakness in understanding of this most fundamental principle. Using operational models and pattern-matching skills, students are often able to solve traditional exercises in recursive programming without gaining a deep conceptual grasp of recursion. Such crutches invariably collapse under the complexity of an interpreter's run-time behavior, forcing students to develop a working grasp of recursion's power. A number of students have commented that the most valuable lesson they learned from this course was a true understanding of recursion.

A great benefit of this approach is that the material comes alive through hands-on assignments. Though the amount of code required in the exercises assigned in our course is less than that of other programming-oriented courses our majors take, students invariably report that the ratio of thinking per line of code is much higher in this course.

Students frequently rate this course as the most challenging and theoretical in their undergraduate education (in spite of the absence of formal mathematics). But it is worth it. In formal and informal reviews of student performance after graduation, this course is often listed by students and their employers as one of the most valuable in their education.

Meta-Linguistic Abstraction

There is a substantial bonus to the extensive use of interpreters. It trains students in the use of *meta-linguistic abstraction:* the solution of problems via design of specialized programming languages. Though this might seem a far-flung notion, it is very practical.

Most large programs have significant components that are driven by tables or other forms of structured data. In a sense the rules for structuring the data constitute a language that the program interprets. Frequently these languages involve elements such as recursion, symbol-value binding, name-spaces, scope, keywords, type-correctness, and control structures: many of the same principles underlying traditional programming languages. By studying such basic principles, our students are better prepared to design and implement programs that employ these techniques, though design of a "programming language" may not enter their mind.

Looking at this another way, the suitable use of abstractions (such as procedures, objects, and types) is central to programming. In virtually all large programming projects, some elements of the problem at hand cannot be captured optimally given the abstraction mechanisms supported by any given language, or even any collection of existing languages. There may then be tremendous leverage in the programmer's asking: What would be the *best* abstraction to solve this problem? What would be the *ideal* language in which to specify a solution to this problem? (Usually this is a language that is tailored specifically to the details of the problem at hand.) This is what it means to use meta-linguistic abstraction: in a sense the ultimate form of abstraction.

As noted above, programmers routinely use very elementary forms of meta-linguistic abstraction in the design of tables and other data structures. But such use is critically constrained without training in the range of semantic possibilities in programming language design and in appropriate implementation techniques.

When programming language semantics is taught only in the context of complex general-purpose languages, and language implementation is associated with optimizing compilers, it is no wonder that programmers seldom think of adopting language design concepts to the solution of programming problems. By contrast, students are empowered

to make powerful use of meta-linguistic abstraction when they are taught language principles via implementation of a number of mini-languages using small interpreters.

NSF Sponsorship

This work was supported in part by NSF grant CDA-9312614, *Tools and Techniques for Use of the Scheme Programming Language in Undergraduate Education*, with co-principle investigators Daniel P. Friedman, R. Kent Dybvig, George Springer, and LauraLee Sadler. Perhaps a quarter of the effort in this large grant was devoted to the programming language course pedagogy outlined here, with other major effort devoted to programming environment development and CS1 and compiler course pedagogy. The grant was for three years beginning in 1993, but work has continued to date with university matching funds.

Dissemination has employed traditional means such as publication of books, presentations at major conferences with published proceedings (domestic and foreign) and less formal workshops in the US and Mexico. In addition, a web cite has been constructed containing a variety of material prepared by this project, as well as related work elsewhere. [13] To provide further support for Scheme users, a small supplemental grant was obtained to fund a research assistant for a year to construct a web-based repository of Scheme-related material from all available sources. [14]

Finally, we recognized the value of direct contact with prospective teachers for effective pedagogy dissemination, particularly when a paradigm shift and a variety of new skills are involved. Thus we offered a half-day workshop at a major conference and a series of extended workshops, including two devoted to the programming languages course: a two-week workshop in the summer of 1995 and a one-week workshop in the summer of 1996.

Conclusion

As a scientific discipline matures, formal techniques are first introduced to the research community, then become standard fixtures of graduate education, and finally are incorporated into undergraduate instruction. Applying analytic techniques in general, and interpreters in particular, to the study of programming languages began as a research technique and has been widely used in graduate education. We have argued that if suitably approached it is also valuable in undergraduate education.

Acknowledgments

Professor Daniel P. Friedman provided the original inspiration for this approach at Indiana University and has continued to be a driving force behind its development, including its use in graduate studies and programming language research. Other principle contributors include professors Mitchell Wand and William Clinger (now both at Northeastern). Throughout the years a large number of creative and industrious teaching assistants have also made substantial contributions as they assisted us with these courses. Finally, as always when pedagogy is breaking new ground, the feedback of countless students has been invaluable, as has their patience with constant curricular experimentation.

References

[1] ABELSON, H., AND JAY SUSSMAN WITH JULIE SUSSMAN, G. *Structure and Interpretation of Computer Programs*, second ed. MIT Press, 1996.

[2] FRIEDMAN, D. P., AND FELLEISEN, M. *The Little Schemer*, fourth ed. MIT Press, 1996.

[3] FRIEDMAN, D. P., WAND, M., AND HAYNES, C. T. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 1991.

[4] GUNTER, C. A. *Semantics of Programming Languages*. MIT Press, 1992.

[5] HAYNES, C. T. Compiling: A high-level introduction using scheme. In *Proceedings of the Twenty-Eighth SIGCSE Technical Symposium on Computer Science Education* (1997), pp. 253–257.

[6] HAYNES, C. T. Type checking and inference. Technical Report 491, Indiana University, Bloomington, Indiana, 1997.

[7] JOHN MCCARTHY, *et al.*. *LISP 1.5 Programmer's Manual*. MIT Press, 1962.

[8] KAMIN, S. N. *Programming Languages: An Imterpreter-based Approach*. aw, 1990.

[9] MACLENNAN, B. J. *Principles of Programming Languages: Design, Evolution, and Implementation*, second ed. Holt-Rinehart & Winston, 1997.

[10] MANIS, V. S., AND LITTLE, J. J. *The Schematics of Computation*. Prentice-Hall, 1995.

[11] SETHI, R. *Programming Languages: Concepts and Constructs*, second ed. Addison-Wesley, 1996.

[12] SUSSMAN, G. J., AND STEELE JR., G. L. Scheme: an interpreter for extended lambda calculus. Tech. Rep. AI Memo No. 349, MIT Artificial Intelligence Laboratory, 1975.

[13] www.cs.indiana.edu/scheme-repository.

[14] www.cs.indiana.edu/eip.

[15] WINSKEL, G. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.