Toward a Comprehensive Supplement for Language Courses

Krishnaprasad Thirunarayan, Stephen P. Carl
Dept. of Computer Science and Engineering
Wright State University, Dayton, OH-45435
Email: {tkprasad,scarl}@cs.wright.edu
URL: http://www.cs.wright.edu/~tkprasad
Phone: (937)-775-5109 Fax: (937)-775-5133

Keywords: Education, Innovative Teaching Strategies, Collaboration, Languages.

"The utility of a notion testifies not to its clarity, but rather to the philosophical importance of clarifying it." - Nelson Goodman (1955)

Abstract

Information Systems are coded in a variety of programming languages ranging from production object-oriented languages to scripting languages to declarative languages. It is crucial that the programmers have a good understanding of the fundamental concepts, notations and techniques that underlie these languages and issues related to evaluating and choosing from among these languages.

There exist many good textbooks and Web-based resources for the study of comparative programming languages. In spite of these resources, however, students often fail to grasp the fundamentals and their application to both modern language implementation and their own problem solving skills.

Modern languages are geared towards enhancing programmer productivity, sometimes at the expense of efficiency. For every language, paradigm, or style, there exists some application domain for which it is well-suited. We believe that a comprehensive set of programming exercises are needed that clearly bring out the role of each paradigm/language, especially the fact that they are *complementary* in nature rather than competitive.

1 INTRODUCTION

Information Systems encompass a wide variety of applications based on diverse techniques and tools such as Reasoning and Mining algorithms, Search and Collaboration tools, AI techniques, etc. The components that constitute information systems are implemented in a variety of languages based on different programming paradigms. Object-oriented style facilitates modular program organization and evolution using inheritance, polymorphism and dynamic binding; typed languages support reliable programming; meta-programming features, originally developed in the context of functional and logic languages, contribute to power and flexibility; and scripting languages promote rapid prototyping. In order to be able to understand, evaluate, and build applications, it is important to have a good grasp of the languages and the foundational infrastructure they provide. This paper addresses the problem of teaching such basics concretely and effectively, with the hope that analogous aspects will have broader appeal in teaching Information Systems too. We believe this paper will have direct influence over information systems in so far as interface design can be viewed as a toy language design activity, and indirect influence through such counterparts as the diversity of paradigms, lack of standard vocabulary and usages, difficulties in specification, lack of simple discriminating examples, etc. Information Systems educators can use the web for dissemination of courseware and collaborative web systems such as JWiki or Swiki web server to provide editable web pages.

1.1 Background

The study of programming languages is fundamental to Computer Science, and there are excellent undergraduate textbooks covering the subject, such as those by Sethi [9] and Friedman, et al [4]. There are also a number of resources on the Web, some of which support these textbooks and others which gather together materials from across the Internet, such as *The Teaching about Programming Languages Project* [6]. This site also includes a list of commonly used textbooks for both undergraduate and graduate courses, plus links to courses which use them.

In spite of these resources, many students in the comparative programming languages course fail to grasp the fundamental ideas underlying modern languages of various kinds (procedural, functional, logic, and so on) and have difficulty applying them in practice. One possible reason for this is the lack of a *consolidated* archive of concrete examples that clearly illustrate, compare, and contrast implementations of the fundamental concepts using modern languages. Abstract classroom discussions and existing materials do not seem adequate for many of today's undergraduates, who may have programmed in a language such as `C/C++/Java` for some time, but who seem to have less mathematical maturity.

In this paper, we outline an "example-rich" supplement that illustrates fundamental language concepts and provides a comparative study of modern languages. By making these and other examples conveniently accessible through a collaborative website [10], it will be possible for educators to add to or critique the material if they so desire. Links to exercises and examples from other public sources can be included which support the goal.

2 GOALS AND OBJECTIVES

The goal of our work is to enhance students' *grasp* and *appreciation* of the fundamental concepts, constructs, and paradigms that underlie modern programming languages. This will not only enable comparison and evaluation of various programming languages, but also help fundamentally in program development. This is especially important in the context of students who have narrow exposure to few programming languages, are ill-prepared in discrete mathematics and logic, and are misinformed about the relative merits and deficiencies of existing languages.

A secondary goal is to develop examples and exercises relevant to and coded in languages that embody modern computer science concepts and reflect current trends, without being bound by them.

When teaching a course over and over again, it is difficult to create new assignments and exam questions each time. We hope to help others develop ideas by sharing this work over the web and updating it with other's contributions.

3 MOTIVATION FOR SUPPLEMENTARY EXAMPLES

A good understanding of programming language concepts and paradigms enables Computer Science undergraduates to learn new languages more quickly and provides them a better grasp of extant languages. This is especially important given that graduates of our CS programs will likely use languages that are yet to be developed, as programmers today use Java and other web-based languages. A course in programming languages should prepare today's student to isolate analogies and differences between new languages and those which they are exposed to during coursework. Experience shows, however, that many students are still unable to assimilate the programming languages landscape. This is, in part, due to the fact that:

1. Textbooks lack concrete examples that *clearly* illustrate similarities, differences, and tradeoffs among related features, as discussed later in Section 4. Of course, there are exceptions in the treatment of topics such as parameter passing and runtime storage management.
2. Different languages embody the same concept using (sometimes wildly) divergent syntactical contructs, while similar-looking syntax may have subtly different semantics. As a result, students sometimes pass off as mere syntactic differences in programming languages certain significant conceptual differences.
3. Undergraduate students exhibit fundamental misconceptions about the following:
   - the differing roles of variables in imperative, functional, and logic languages.
   - the basic organizational differences between programs written in object-oriented versus procedural style.
   - iteration and recursion, especially the erroneous belief that iteration is "efficient and easy" while recursion is "inefficient and difficult", without any further qualification.

   These misconceptions are manifest in varying degree in the following ways:
   - Students often use technical jargon without exhibiting a clear understanding of the terms.
   - Some students can describe a concept in abstract terms, but cannot apply it to a concrete situation.
   - Some students seem to know the concept, but when requested to cite an illustrative example are unable to go beyond the "standard" hackneyed ones.

Textbooks such as [4, 5] address such problems by presenting a unified framework, the former by building interpreters in Scheme which add new language features, the latter by developing denotational semantics for expressing language features. Unfortunately, a typical CS program does not expose the student early enough or long enough–if at all–to Scheme or Discrete Math, so that these works may require considerable adaptation.

The concepts we are referring to are in fact well-known and the differences well-established. However, there does not appear to be a readily or widely available single resource that archives crystal clear examples. This is a deficiency that we would like to address.

4 SOME ILLUSTRATIVE EXAMPLES

In this section we briefly sketch out a small set of fundamental programming language concepts, examples, and sample problems to be used in supplementary materials. These examples are chosen to illustrate the concepts while providing a comparative study through concrete examples in modern languages. Code examples presented here reflect straight-forward implementation of an idea.

We first present the concepts which some students have difficulty with and give an illustrative example, followed in some cases by a set of relevant exercises.

4.1 Programming Styles

L-values vs. R-values

Students do not seem to understand the semantic difference between the *l-value* (location associated with the variable) and the *r-value* (the contents of the location) well enough to be able to apply it in practice.

Consider a C++ class definition which requires overloading the indexing operator `[]`. Because the indexing operator can appear on the left or right hand side of an assignment statement, it must return a reference to an array element (`int&`), rather than just the element itself (`int`):

```
class unconstrained_array {
private:
   int *p, lb, ub;
public:
   unconstrained_array(int l, int u) {
     lb = l;   ub = u;
     p = new int [ub - lb + 1]; }
   ~unconstrained_array() { delete [] p; }
   int first() {return lb;}
   int last() {return ub;}
   // Note: return type is "int&" and not "int"
   int& operator[](int i) { return( p[i] ); }
};
```

Here is an example of client code which uses this operator as both l-value and r-value:

```
int sum_of_sq(unconstrained_array a) {
   int s = 0;
   unconstrained_array b (a.first(), a.last());
   for (int i = a.first(); i<=a.last(); ++i)
      s +=  (b[i] = a[i] * a[i]);
   return (s);
}
```

Iteration vs. Recursion

The typical student is surprisingly misinformed about the relationship between iteration and recursion. Relative to iteration, they regard recursion as difficult and inefficient. One can illustrate that *repetition* is captured in procedural style using iteration and in functional style using recursion. *Tail-recursive* definitions can be translated into equivalent iteration for space-efficiency. To emphasize the fact that languages and styles are independent, the solution can be coded in C and Scheme. Further, Scheme can be cited as a language that mandates that every implementation perform tail-recursive optimization, thereby making looping-constructs dispensable. The convenience and expressiveness of recursion can be illustrated using operations over recursive data structures that would otherwise require an explicit stack.

Sample Exercises

- Write a program in procedural language to read and evaluate a sequence of infix arithmetic expressions containing integers and binary operators.
- Implement a family of evaluators (as in [4]):
  1. A calculator for constant arithmetic expressions.
  2. A calculator with memory for expressions.
  3. A programmable calculator to accomodate user-defined macros and functions.

  This can be further generalized to work on polynomials instead of numbers:
  1. Specify *ADT Polynomial* using algebraic specification techniques.
  2. Implement a polynominal calculator with memory.
  3. Implement a programmable polynominal calculator.

OOP vs. Procedural

The fundamental difference between object-oriented style and the procedural style can be understood in the level of *encapsulation* and *abstraction* they support. They differ in data and code organization in the context of integrating multiple implementations of the same behavior [1, 2, 7]. Furthermore, addition of a new procedure is *incremental* in procedural style (but not in object-oriented style), while an addition of a new implementation is *incremental* in object-oriented style (but not in procedural style).

The `size()` function in Scheme illustrates procedural style:

```
(define (size C)
   (cond ((vector? C)  (vector-length C))
      ((pair? C)    (length C))
      ((string? C)  (string-length C))
      ( else        (vector-length C)) ))
(size '(one ``two'' 3))
```

The same style in the `C` language would use *union types*, while in Pascal it would use *variant records*. The run-time type tests in Scheme are replaced with checks for appropriate tag field values. The following recasting of the `size()`-example in Java illustrates object-oriented style:

```
interface iCollects { int size(); }
class cVector extends Vector implements iCollects
{}
class cString extends String implements iCollects {
   public int size() { return length(); }
}
class cArray implements iCollects {
   int[] array;
   public int size() { return array.length; }
}
iCollects c = new cVector(); c.size();
```

Another significant point about the object-oriented style is the role of interfaces [3]. For example, in client-server programming, the procedural style encourages servers to *export* their services under some name while requiring clients to supply the glue code to deal with name mismatches. In object-oriented style, the *public interfaces* specify a protocol that a client can depend on and a server must respect. These ideas underlie *interchangeability* and *substituitivity* of software components on a larger scale.

Sample Exercises

- Write evaluators to compute the type and the value of an expression in message passing style and in procedural style in Scheme. Repeat using object-oriented style and procedural style in C++ or Java.
- Given n clients and m servers, quantify the superiority of object-oriented over procedural decomposition considering potential changes.

Logic vs. Functional/Imperative

To distinguish imperative languages from logic and functional languages, it is important to illustrate what *declarative specification* means. The benefits of the logic programming paradigm can be brought out by illustrating the role of pattern matching in making programs concise, and illustrating how an interpreter supplies the necessary control details in solving a query.

For example, consider a definition of the operation `append` in Prolog, followed by examples of a verification task `append([a],[1,2],[a,1,2])`, a constraint solving task `append([a],X,[a,1,2])`, or an enumeration task `append([a,1,2],X,Y)`) based on the definition.

Sample Exercises

- Using Prolog, formalize information about human relationships in order to answer queries about a highly tangled family where the husband's daughter by a former marriage marries the father of the (current) wife [11].
- What does the following Prolog program compute?

```
suspect([],[[]]).
suspect([H|T],S) :- suspect(T,P),
                    accomplice(H,P,R),
                    append(P,R,S).
accomplice(_,[],[]).
accomplice(H,[X|Y],[[H|X]|R])
                  :- accomplice(H,Y,R).
```

  Identify the guilty and informally justify your claim.

Paradigm Comparisons

The role of various languages and specification techniques can be nicely brought out through the following set of exercises:

1. Design an attribute grammar to capture static and dynamic semantics of a sequence of statements.
2. Turn it into an executable specification by rewriting it in Prolog.
3. Construct a table-driven, parser-based program in a procedural language for efficiency.

4. Write a program to: (1) append two lists and (2) transpose a matrix, in a functional, logic, object-oriented and procedural language.

4.2 Programming Language Design

Here is an incomplete list of programming language design issues and ideas that deserve emphasis.

Portability

In order to grasp the benefits of portability and show that it often is lacking in various languages, we cite simple examples that output different results when run on different platforms. For instance, the following (deceptively simple) C language example behaves differently when compiled and run under UNIX on SUN3, SPARC-20, MIPS, and ALPHA machines, and in each case generates a different output than what students expected:

```
#include <stdio.h>
main() {
 int i = 5;
 printf("\t i = %d, i/++i = %d, i = %d\n\n",
            i,      i/++i,      i);
}
/* Compilers: cc, gcc
      SUN3 :          i = 6,  i/++i  = 1, i = 5
    SPARC20:          i = 6,  i/++i  = 1, i = 6
ALPHA/MIPS :          i = 5,  i/++i  = 1, i = 6
INTUITION:          i = 5,  i/++i  = 0, i = 6 */
```

The point is not what the correct answer is, but that the implementations do not all agree on the interpretation. Language standardization efforts address such issues; when the above example was run on different platforms whose compilers conform to the recent ISO C++ standard, each produced the same answer. By contrast, the designers of ML and Java have tried to achieve portability (platform independence and architectural neutrality) through strict language definition.

Associativity of Operators

Mathematically speaking, integer addition, string concatenation, and real multiplication, etc. are all associative operations. The Java infix operator `+` denotes both concatenation and addition, and is *not* associative due to coercion rules (e.g., `(5 + 3) + "abc" =/= 5 + (3 + "abc")`)). Similarly, `*` is not associative due to underflow/overflow issues. The Java Language Specification declares these operators *left associative* to disambiguate infix expressions that are not fully parenthesized.

Scope vs Lifetime

Students with experience in C/C++ often confuse *scope* and *lifetime*. This is partly because the scope of a variable is the same piece of code which has control when the variable has been allocated on the runtime stack. Their independent nature can be illustrated using Scheme closures or Java inner classes.

Typing

The relationship between static typing, dynamic typing, typing in object-oriented programming languages, and so on can be clarified using examples from various languages. Typing in OOPLs presents an interesting tradeoff between the efficiency associated with statically-typed languages and the flexibility associated with dynamically-typed languages, without sacrificing reliability.

Variable types in languages such as C++ and Java have a superficial resemblance, but differ significantly in the details. Primitive types in both languages behave similarly, but composite types do not. A reference to a Java object corresponds (roughly) to a variable holding a *pointer* to a C++ object, but the Java syntax resembles that of the C++ structure variable rather than a pointer. Java has no counterpart to C++ structures or Eiffel *expanded types* [7].

Ada *generics*, C++ *templates*, and ML *functors* provide a rich source of examples of abstraction and its benefits, such as how various semantic constraints are expressed, minimizing code duplication and promoting reuse.

4.3 Object-Oriented Programming

Analogies and diverse examples can help students internalize the meaning and significance of the confusing lot of OOP concepts: objects, classes, encapsulation, inheritance, composition, delegation, polymorphism, to name a few. For example, in tracing the following Java program one needs a good grasp of static overload resolution, coercion, and dynamic dispatch.

```
class P {
   public void f(P p)
   { System.out.println("f(P) in P. "); }
}
class C extends P {
   public void f(P p)
   { System.out.println("f(P) in C. "); }
   public void f(C cp)
   { System.out.println("f(C) in C. "); }
}
class DynamicBinding {
```

```
    public static void main(String[] args) {
        P px  = new P();   C cx  = new C();
        P py  = cx;
        px.f(px);           //f(P) in P.
        px.f(cx);           //f(P) in P.
        py.f(px);           //f(P) in C.
        py.f(cx);           //f(P) in C.
        cx.f(px);           //f(P) in C.
        cx.f(cx);           //f(C) in C.
    }
}
```

Sample Exercises

- Given the Java class hierarchy above and a sequence of variable declarations such as `P a;  C b;`, specify the type correctness of the assignments `b = a;` and `a = (A) b;`, considering casts and coercions.
- Explain the significance of OOP in the context of Java Collections Framework.

5 CONCLUSIONS

We have presented a set of concepts and exercises developed to provide concrete examples of certain programming language concepts and illustrate problems which different paradigms are well-suited to. This is motivated in part by the lack of a comprehensive set of examples among available resources, and in part by student problems, issues, and misconceptions discovered while teaching PL classes at both the undergraduate and graduate levels. The student surveys and evaluations in the Language courses, including unsolicited feedback on this prerequisite material from graduate students taking advanced courses, seem to indicate that the students benefited immensely from concise and clear examples we presented. We intend to provide more on-line examples in a collaborative setting featuring quick access to specific concepts. We hope this will provide others with relevant material, as well as help students with self-study [8, 6]. We also believe that many of the issues addressed here have analogies in the realm of Information Systems, and our approach can be beneficial. Information Systems educators can use the web for dissemination of courseware and collaborative web systems such as JWiki or Swiki web server to provide editable web pages.

References

[1] H. Abelson, G. J. Sussman, and J. Sussman: *Structure and Interpretation of Computer Programs.* 2/e, MIT Press, 1996.

[2] T. Budd: *An Introduction to Object-Oriented Programming.* 2/e, Addison-Wesley, 1997.

[3] B. Cox and A. Novabilsky: *Object-oriented Programming, An Evolutionary Approach.* Addison-Wesley, 2/e, 1986.

[4] D. P. Friedman, M. Wand, and C. Haynes: *Essentials of Programming Languages.*, MIT Press/McGraw-Hill, 1992.

[5] M. J. C. Gordon: *The Denotatioal Description of Programming Languages.* Springer-Verlag, 1979.

[6] G. T. Leavens: *The Teaching about Programming Languages Project,* `http://www.cs.iastate.edu/~leavens/teaching-prog`

[7] B. Meyer: *Object-Oriented Software Construction.* 2/e, Prentice-Hall, 1997.

[8] `http://www.cs.wright.edu/~tkprasad`.

[9] R. Sethi: *Programming Languages: Concepts and Constructs.* Addison-Wesley, 2/e, 1996.

[10] Swiki/CoWeb introduction at `http://coweb.cc.gatech.edu/csl/9`.

[11] N. Wirth: *Algorithms + Data Structures = Programs.* Prentice-Hall, 1976.

Brief Biographical Sketches:

Krishnaprasad Thirunarayan received his Ph.D. in Computer Science from the State University of New York at Stony Brook, M.E. in Computer Science from the Indian Institute of Science, Bangalore, and B.Tech. in Electrical Engineering from the Indian Institute of Technology, Madras. He is currently an Associate Professor in the Department of Computer Science and Engineering at the Wright State University, Dayton, Ohio. His research interests are in Knowledge Representation and Reasoning, and Programming Languages:Design, Specification, and Implementation.

Stephen P. Carl is currently an instructor and a doctoral student in the Department of Computer Science and Engineering at Wright State University, Dayton, Ohio. He has an M.A. in Computer Science from the University of Texas at Austin, and a B.S.E.E in Electrical Engineering from Rice University, Houston. His research interests are in Simulation and Implementation of Languages and Environments for Internet Applications.