The Comparative Programming Languages Course: A New Chain Of Development

Prof. W. Douglas Maurer
Department of Computer Science
The George Washington University
Washington, DC 20052
E-mail: maurer@seas.gwu.edu

Abstract

The programming language concepts which ought to be presented in the comparative programming languages course (either graduate or undergraduate) are all covered by choosing C++, Java, Perl, and Python as the languages to be compared. These include dynamic typing, object orientation, multiple inheritance, interpreters and compilers, keyword and default parameters, generics, operator overloading, complex numbers, universal hierarchies, exceptions, and garbage collection. We describe such a course, which we have given.

1 Introduction

Within a computer science curriculum, the comparative programming languages course typically appears as a senior or first-year graduate course. Most such courses remain profoundly influenced by the ACM Curriculum Recommendations (Volume I: Computing Curricula 1991) [8], which specifies that such a course should cover "the evolution of procedural languages (e.g., the Algol, PL/I, Pascal, Euclid, Modula-2, and Ada chains of development)."

In May through July, 2001, we presented a section of our first-year graduate comparative languages course, which we refer to as CS 210. In this section, we introduced a chain of procedural language development which is different from those listed above, namely C, C++, Java, Perl, and Python. All these languages share a considerable amount of syntax while, at the same time, represent a variety of programming language concepts, such as dynamic typing, object orientation, multiple inheritance, interpreters and compilers, keyword and default parameters, generics, operator overloading, complex numbers, universal hierarchies, and exceptions.

In August 2001, shortly after this course section ended, there was released a working draft [3] of the Steelman report of the Year 2001 Model Curriculum for Computing. Like its predecessor [8], it was produced by a joint task force from ACM and the IEEE Computer Society. As this is written, however, the extensive list of course descriptions which accompanies [3] (as Appendix B [4]) does not yet include a comparative languages course at all. Partly for this reason, we have undertaken here to describe our own experience and to advocate the inclusion, in the final version of [3], of a comparative languages course which explicitly mentions the chain of development starting with C.

The C chain (as we shall refer to it) divides naturally into two subchains, the "systems chain" (CPL, BCPL, B, and C) and the "general chain" (C++, Java, Perl, and Python). The first group is made up of systems programming languages, intended for use by expert systems programmers; they therefore do not include a number of safety features, such as subscript range checking. The second group is made up of general-purpose programming languages. All of these, except for C++, always check subscript ranges; and even in C++ one can have subscript ranges checked by using array classes, such as those of STL [7] or the College Board's AP Computer Science course (see, e. g., [1]). Similarly, the safeguards of object-oriented programming are avail-able in all the languages of this group (and are required in Java).

All four of the languages in the general chain are available on all major platforms. This may seem surprising for Perl and Python, both of which originated in the Unix world; however, both of them are now available for both Windows and the Macintosh, without the use of Unix on Macintosh or Windows machines. (For those who don't know Python, it is vaguely like Perl in the same sense that Java is vaguely like C++.)

SIGCSE'02, February 27- March 3, 2002, Covington, Kentucky, USA.

2 Advantages of Our Approach

It might seem unusual for a comparative languages course to concern itself with only a single chain of programming language development, rather than choosing languages from several chains. Our choice, however, has six major advantages:

(1) It includes almost every interesting programming paradigm:

(a) dynamic typing with no type declarations (Python) and with a very few type declarations (Perl), as well as static typing (C++, Java);

(b) required OO (Java) and optional OO (C++, Perl, Python);

(c) multiple inheritance (C++, Perl, Python);

(d) interpreters (Perl, Python, Java) and compilers (C++, Java);

(e) keyword parameters and default parameters (Python);

(f) generics (done with templates in C++);

(g) operator overloading (C++, Perl, Python);

(h) complex numbers built in (Python) or through operator overloading (C++);

(i) universal hierarchies (Java, Perl);

(j) languages with built-in exceptions (Java, Python) and with exceptions, but none built-in (C++, Perl); and

(k) storage reclamation with garbage collection (Java, Perl, Python) and without it (C++).

(2) It gives the students hands-on experience with the problem of comparing static and dynamic scoping. This is because Perl has a keyword (local) which sets up local variables with dynamic scoping, as well as another keyword (my) which sets up local variables with static scoping.

(3) It suggests a spirited discussion of the language complexity question. Should a good language be constructed by eliminating unnecessary features, so as to make the language easier to learn (like Java)? Or should its construction strive to give the user every conceivable feature; even if this means that there is usually more than one way to do something (as in Perl)?

(4) It suggests a spirited discussion of the go-to controversy by presenting two languages (C++ and Perl) having the go-to, and two languages (Java and Python) not having it. Paradoxically, Perl, although it has the go-to, also has the most extensive collection of substitutes for go-to of any major language. This allows students with annoying questions of the type "but how would you do *this* without a go-to?" to see how it can logically be done, and try it out in practice.

(5) It relates the course to new developments in data structures by introducing resizable arrays and resizable hash tables as built-in data types (in both Perl and Python). Recent work has shown the considerable advantages of such structures over non-resizable structures for many applications. In particular, resizing by doubling assures that amortized time and space requirements for N resizings are O(N) (as opposed to $O(N^2)$ when resizing is by a fixed amount, even one chosen by the programmer).

(6) The considerable syntactic similarity among these languages allows the students to write interesting programs in three different languages in one semester. In particular, each of our students wrote a Perl-subset interpreter written in Python; a Python-subset interpreter written in Java; and a Java-subset interpreter written in Perl. This was made considerably easier because of built-in hash tables in Perl and Python, and because the Perl approach to string pattern matching (of considerable use in writing an interpreter) has recently been adopted by both Python and Java.

Our use of C++, Java, Perl, and Python is not put forth as a "best possible" approach to the comparative languages course. It is rather in keeping with the general philosophy of the Steelman report, which allows a wide variety of implementation strategies for introductory computer science.

3 The Common Syntax Within the Chain

All four of the languages within our chain of development have the following syntax in common:

(1) Square brackets used for subscripts, and indices starting from zero.

(2) Variable names containing only upper and lower case letters, underscores, and digits; a digit cannot be first. (In Perl, there is an additional punctuation character at the beginning, denoting the type of the *result* — not necessarily the type of the variable. Thus @A denotes an array A, but $A[$J] denotes the scalar A[J] where J is a scalar.)

(3) Expressions involving the operators + - * / % == != < > <= >= << >> & | ^ ~ (and sometimes others).

(4) If-statements followed by a condition in parentheses, and then a compound statement (enclosed in curly brackets except in Python, where it consists of a colon followed by several statements indented one more level than the current statement).

(5) While-statements, which are syntactically like if-statements.

(6) Return-statements, to return the value of a function.

Despite the similarities among our four languages, however, each of the languages has uniquely interesting features.

4 Static and Dynamic Typing

The older term for languages with dynamic typing was "typeless" languages. By this was meant that there were no type *declarations*; you do not declare a variable as inte-ger, real, character, or the like. However, there are still types in

such a language; any variable can be of any type, and this type can change at run time (hence the name "dynamic typing"). The popularity of Perl (and, to a lesser extent, of Python) seems to be reviving the idea of dynamic typing and making it more important that students know about it.

Several characteristics are common to all dynamically typed languages. No common hardware supports dynamic typing directly (although this has been proposed from time to time; see, for example, [6]). Therefore dynamically typed languages cannot be compiled to object code. They are typically compiled into what is properly called an intermediate language, although this is most commonly known today as "compiling into bytecodes" (see, for example, [2], pp. 14-16; [5], p. 137; or [9], p. 466). There is then an interpreter for the bytecodes.

Although there are no ordinary type declarations, arrays, objects, and functions still have to be set up. Here the various dynamically typed languages have taken many different approaches. Early dynamically typed languages had *executable* declarations for arrays, each of which specified the array length. This is not necessary in either Perl or Python, in which arrays are resizable.

5 Presentation of OO in the Course

Today OO is so pervasive that even Perl and Python have it; and this may seem surprising to those who learned early versions of Perl or Python. Both of these got their start as languages for system administrators, whose programming tasks tend to be relatively simple and could not use OO until more recently. It is tempting to regard OO as being "tacked on" to both Perl and Python and thus of less importance than other forms of OO; however, C++ was similarly tacked on to C, while Java was designed directly from C++.

All OO begins by generalizing the concept of a record in Pascal or Ada, or a struct in C. The resulting generalization is called a class (C++, Java, and Python, following Smalltalk) or a package (Perl, following Ada). Its fields, or components, are called members (C++) or instance variables (Java) or attributes (Python) or blessed variables (Perl). This last phrase arises from the call to bless(p) within the package X, or, more generally, bless(p,X), which sets up what p points to as a blessed variable of X.

The generalization proceeds by associating certain functions with such a class or package called X. These are called methods (Java and Python, following Smalltalk) or member functions (C++); they have no special name in Perl. A constructor in C++, Java, or Python has a name which is the same as the name of its class. The object for which a particular function is invoked (called the *invocant* in Perl) is denoted by *self* (Python) or *this* (C++, Java), or (in Perl) either of the above but by common practice only.

There are then base classes and derived classes (C++) or superclasses and subclasses (Java, Perl, Python). These are subject to certain variations; thus, in Perl (only), variables are not inherited (although methods are); in Java (only), multiple inheritance is not allowed except in interfaces; while in C++ (only), polymorphism must be specified explicitly through a keyword (virtual).

6 Multiple Inheritance

The main disadvantage of multiple inheritance is inefficiency of variable inheritance. This is presumably why both Perl and Python have multiple inheritance, since neither of these languages is concerned with efficiency.

Java, which is concerned with efficiency, does not have multiple inheritance except for interfaces, which do not have variable inheritance. C++ is also concerned with efficiency, and Version 1 of C++ also had no multiple inheritance. Current versions of C++ have it, but at the re-quest of users and accepting the inefficiency involved.

Python very neatly solves the problem arising in C++ when, for example, class D is derived from both B and C, which are themselves derived from A; any member X of A is inherited twice by D unless A is declared as a virtual base class of B and of C. This problem does not arise in Python, where order matters, in multiple inheritance. Here, if D is defined by class D(B,C), then B comes before C, so that X is inherited by D through B and then not through C, because it was already inherited through B.

7 Interpreters and Compilers

The demise of the 8-bit microcomputers, with their strong dependence on Basic interpreters, decreased student interest in interpreters, in general, in many schools. Java reversed this to a certain extent, but just-in-time compilers for Java are now quite popular. Perl and Python, however, are also interpreted languages, and interpreters ought to be understood by students. They also ought to learn the terms "semi-interpreter" and "intermediate language"; regrettably, these terms are little used today, though the concepts they describe are very much used.

Among the advantages of an interpreted language (provided that you accept the inefficiency of interpre-tation) are the possibilities of expression evaluation and code execution functions, which are in both Perl and Python. An expression, or a piece of code, in Perl or Python is a string, and a function can evaluate that expression, or execute that code, if the language is in-terpreted. These functions are called eval and exec in Python, and they are called eval and system in Perl (there is also exec in Perl, but this does not return to the calling program).

8 Parameter Styles

The old Fortran idea that parameter passing modes ought to be fixed in a language, and unchangeable by the programmer, is now making a comeback. This does not mean that programmers are not concerned about what the modes are;

they need to know the difference between call by value (used normally in C++, Java, and Python) and call by reference (used with reference parameters in C++, or arrays and objects in Java, or one-element lists in Python). Perl is unique in having no built-in formal parameters at all; they appear in a list called @_ and have the names @_[0], @_[1]. @_[2], and so on, and they may be modified.

The one major language technique which is not found in any of our four languages is call by value and result, which is pervasive in Ada. Another parameter technique from Ada, however, is also found in Python, namely keyword parameters and default parameters. These are useful when a function has ten or more parameters, making it difficult to remember whether a certain parameter is the seventh or the eighth, for example.

9 Generic Programming

This is still another technique from Ada, allowing one to write a generic search program, for example, which searches for objects of any kind. In C++, this is done through templates. As in Ada, a generic function is not compiled unless it is used, and then the only instances of it which are compiled are the specific instances which are actually used.

10 Operator Overloading

Our languages also have various approaches to overloaded operators. In Java, there are none; in C++ and Python, they have special names (operator+ and the like in C++; __add__ and the like in Python); while in Perl they are functions with any name, and there is an overloaded operator list which associates such functions with their corresponding operators.

11 Complex Numbers

Over the years, programming language designers have, several times, misunderstood the absolute necessity for complex numbers on the part of scientists and engineers. Fortran has them, but Pascal and C do not. This misunderstanding is still taking place today, specifically in the case of Java. As a language for the Internet, Java is extremely popular; but there have been those who hoped that Java would evolve into something more, namely a replacement for C++ in all its aspects. This is clearly not happening, and the lack of complex numbers is, in our opinion, one of the reasons why not.

The designers of C++ insured that complex numbers would be available through overloaded operators, which are not in Java. One might argue that this is the real reason why Java has no complex numbers; however, Python has complex numbers built in, and a future version of Java could do the same.

12 Universal Hierarchies

Part of the essence of OO in the minds of some people has always been the universal hierarchy of subclasses. Java, in particular, places great value on its universal hierarchy, with Object at the top, and certain abstract methods of Object (toString, clone, equals) which are supposed to be overridden in any newly defined Java class. Similarly, all exceptions are descendants of a general exception class; all components in the abstract windowing toolkit are descendants of a general component class, and so on.

In view of this, it may be surprising to some students how controversial universal hierarchies are. Java and Perl have them, while C++ and Python do not. The universal hierarchy in Perl, in fact, is primitive, although there is a Perl class called UNIVERSAL, which corresponds to Object in Java. More importantly, C++ is still evolving, but not in the direction of a universal hierarchy, although C++ namespaces (which correspond roughly to Java packages) have recently been introduced.

Is there an argument against a universal hierarchy on theoretical grounds? A universal hierarchy forces the user to think at least a little about the entire programming field; but this might itself be perceived as a disadvantage. For example, one might wish to design a language which is accessible to many programmers, each of whom is concerned with one particular application area and has no need to know (and no time to learn) any others.

13 Exceptions

All of our four languages have exceptions, but there are important differences among them. The most idiosyncratic is Perl, which implements exceptions as an extension of the eval function discussed in section 8 above. An evaluation function normally acts on a string, but eval in Perl can also act on an unquoted block of code. This is like try in C++, Java, or Python; that is, it has no purpose other than to direct the system to manage exceptions.

Perl also has a function called die, whose argument is a fatal error message, as a string. This function, when used inside of an eval block, serves the same purpose as throw in C++ and Java, or raise in Python; that is, it triggers an exception.

Both Java and Python use finally as a keyword, to introduce a block which is always done after a try block, whether or not an exception was triggered.

14 Garbage Collection

Of our four languages, only C++ is without a built-in garbage collector. The main disadvantage of garbage collection, of course, is inefficiency, with which Perl and Python are not concerned anyway. Java, however, tries to be efficient, especially with its just-in-time compilers, and yet it uses garbage collection. This fact triggers a discussion of the unusual degree of danger attendant on user dealloca-

tion, particularly when many kinds of deallocation are being done simultaneously.

15 The Class Projects

In our view, every computer science student, at some point, ought to write an interpreter, or at least the first few pages of one; and the comparative languages course is a good place to require this. Such interpreters, if they are being written today, ought to use pattern matching to implement the interpretation, and symbol tables which are hash tables. Built-in hash tables are in both Python (where they are called dictionaries) and Perl; and the pattern matching functions of Perl have recently been copied into both Java and Python.

A word about unfortunate terminological choices is in order here. Perl uses a pattern specification protocol which it refers to as "regular expressions." These, however, are not the regular expressions familiar from automata theory, and included in knowledge unit PL7 of the programming languages course described in [8]. Compounding the confusion, the new pattern matching operations of Java and Python both use this same protocol. We believe that students must learn this as part of computer science education, just as they learn that a heap may be either dynamic storage or an array representation of a complete tree.

We present our students with a general lecture on how interpreters work, together with tie-ins to hash tables and pattern matching. We give another complete lecture on regular expressions in the Perl sense. We then announce the three class projects, written in Java, Perl, and Python and interpreting, respectively, small subsets of Python, Java, and Perl. We do not define the small subsets, but instead specify that each of their programs should be at least five pages long, and that no extra credit will be given for longer programs. Our hope, of course, is that the better students will write longer programs anyway, the better to improve the programming skills which they will need after they graduate.

Almost every student in the class handed in all three of the projects. This was true even though our section was given in the summer, for only eight weeks instead of 14, with two sessions every week instead of only one.

16 Logic and Functional Language Paradigms

In order to present the programming language concepts we have mentioned, as well as giving the students four languages to compare, something had to be left out, which is dear to the hearts of many educators, namely functional languages such as Haskell and logic languages such as Prolog. One could argue that the pure C approach (used, for example, in the Palm OS) was also left out; but this can be done in an operating systems course. Similarly, Prolog can be done in an AI course, as is done at our university. This leaves Haskell, ML, Miranda, and other such languages.

We have noticed that their benefits, such as lazy evaluation, have not been incorporated into other languages. There has been no extension of C for these, with a translator into C (which is how C++ got its start). This is unfortunate, because most people will not use lazy evaluation in a program unless they can also use, in that same program, all the other techniques they need. Consequently, their exposure to functional languages in college would appear to be a waste, because most of them will never see such a language again.

Conclusions

We have successfully implemented the study of C++, Java, Perl, and Python in the comparative programming languages course. In the process, we have introduced as wide a variety of programming language features as we ever introduced before. Although we ourselves teach a variety of courses, we plan to return to this method of teaching comparative programming languages in the future, and we recommend it to others.

References

[1] Astrachan, O. L. A computer science tapestry. (1996) McGraw-Hill, New York.

[2] Deitel, H. M., and Deitel, P. J. Java how to program. (1997) Prentice-Hall, Upper Saddle River, NJ.

[3] Denning, P. J., Cross, J. H. II, Engel, G., Roberts, E., Shackelford, R., et al. Computing curricula 2001, Steelman draft. Online. Internet. [Aug. 1, 2001] Available WWW: http://www.computer.org/education/cc2001/steelman/cc2001

[4] Denning, P. J., Cross, J. H. II, Engel, G., Roberts, E., Shackelford, R., et al., Computing curricula 2001, draft, Appendix B: Course descriptions. Online. Internet. [Aug. 1, 2001] Available WWW: http://www.computer.org/education/cc2001/steelman/cc2001/appendixb.htm

[5] Lutz, M., and Ascher, D. Learning Python. (1999) O'Reilly, Sebastopol, CA.

[6] Maurer, W. D. Architecture for typeless languages and generic programs. *Annual Series of Institute Papers, Institute for System Analysis, Academy of Sciences of Russia* (1993). 19, Prospekt 60-Let Oktyabrya, Moscow, 117312 Russia.

[7] Robson, R. Using the STL (2nd ed.) (2000). Springer-Verlag, New York.

[8] Tucker, A. B., Barnes, B. H., et al. (eds.) Computing curricula 1991. Online. Internet. [Aug. 23, 2001] Available WWW: http://www.acm.org/education/curr91/homepage.html

[9] Wall, L., Christiansen, T., and Orwant, J. Programming Perl (3rd ed.). (2000)O'Reilly, Sebastopol, CA.