TEACHING DATA STRUCTURES WITH Ada: AN EIGHT-YEAR PERSPECTIVE

Michael B. Feldman
Professor
Department of Electrical Engineering and Computer Science
School of Engineering and Applied Science
The George Washington University
Washington, DC 20052

(202) 994-5253
mfeldman@seas.gwu.edu

INTRODUCTION

This paper discusses our eight years of experience in teaching a data structures course using Ada as the primary programming language. The recent history of the course is summarized, emphasizing the transition to Ada. Language-to-language comparisons of several particularly attractive Ada features are given.

THE RECENT HISTORY OF CSci 159

CSci 159, *Programming and Data Structures*, is an undergraduate course in the George Washington University Department of Electrical Engineering and Computer Science, required for undergraduate majors in computer science and computer engineering. The course is also populated by would-be graduate computer science majors who have a weak background in modern data structures, and by graduate students from other fields. Typical enrollment is in the neighborhood of 100-150 students per year.

The author has been teaching this course and others like it since 1975. The primary language used was Fortran in 1975; transitions were made to PL/1 (1978), Pascal (1980) and finally Ada in 1985. Recent primary textbooks have been those of Horowitz and Sahni [Horowitz77], Tenenbaum and Augenstein [Tenenbaum81], Aho, Hopcroft and Ullman [Aho83], and finally this author's own text [Feldman85]. At the risk of appearing to "plug" the book, we refer to it as Feldman in the sequel.

Previous papers have focused on our early use of abstract data types (ADT's) as an idea or "design language" [Feldman80] and on our early use of Ada in this course [Feldman84]; other teachers [Lang89 and Owen87, for example] have reported similar experiences. It is of interest to summarize the integration of Ada into the course as follows:

1981 *text:* Tenenbaum, with this author's rough notes on packages;
Ada packages used as "design language" for ADT's;
compiler: Pascal (Ada compilers did not yet exist!)

1982 *text:* Tenenbaum with some typescript chapters of Feldman;
compiler: TeleSoft subset compiler for Vax/VMS;
about 10% of students coded in Ada, others used Pascal.

1983 *text:* Aho *et al* with most chapters of Feldman in preprint;
about 25% of students coded in Ada.
1984 *text:* entire Feldman preprint, Aho *et al* used as backup;
compilers: TeleSoft subset compilers for Vax and IBM 4381;
50% of students coded in Ada.
1985 *text:* Feldman book;
90% of students coded in Ada.
1986 *compilers:* validated (full-language) Verdix compilers on Vax and Sun;
100% of students coded in Ada.
1988 *compilers:* Meridian AdaVantage site license for DOS-based desktop computers;
many students are acquiring their own compilers for home machines;
computer center "help desks" distribute Ada programs for self-study.

WHY Ada HAS BEEN MY FAVORITE DATA STRUCTURES LANGUAGE

Our eight years of experience, with perhaps 100 students per year involved, confirms our view that in the family of widely-available procedural languages, Ada embodies the most effective collection of features to facilitate the teaching of data structures. This is especially true if one holds, as we do, that a primary focus of a modern data structures course should be abstract data types. We shall present this view with reference to other candidate languages, specifically standard (ANSI) Pascal, Turbo Pascal[1], Modula-2, and C.

CSci 159 fits into our curriculum at about the sophomore level; the students have typically had a semester or two of Pascal. The emphasis in the first two courses is necessarily on program control and algorithm development, and the whole complex of issues we call "structured programming." The primary focuses in the third course are data abstraction (or abstract data types) and algorithm performance prediction.

Ada supports data abstraction better than "the competition" in a number of ways. Chief among them are

- functions can return structured objects, not just scalars (alternatively, objects are "first-class" in that they can be passed to and from subprograms with impunity);
- packages impose a separation of specification and body;
- private types exist and there is no restriction on the type classes which can be made private;
- arrays can be "conformant" (to use Pascal terminology) in all dimensions.

Function result types: Ada. That a function may return a value in any type class, including specifically a record or array, is a feature about which little fuss is made in the Ada literature. But it makes a big difference. Consider the standard example of a rational type:

```
type Rational is record
        Numerator: Integer;
        Denominator: Positive;
end record;
```

[1] A serious question of principle is whether, in this age of portability concerns, a single compiler vendor should be able to define the *de facto* standard for a programming language. This is a matter of taste; we maintain that it should not. Using Turbo Pascal sends a message to our students that portability and standardization play second fiddle to bells and whistles. We discuss Turbo Pascal in this paper because it is, for better or worse, so popular.

Each object of this type is a record. In languages with unrestricted function return values, one can define operations of the form

```
function Add(left, right: Rational) return Rational;
function Mult(left, right: Rational) return Rational;
```

and given four objects `R1, R2, R3, R4`, of type `Rational`, one can write statements of the form

```
R1 := Add(R2,R3);
R4 := Mult(R1,Add(R2,R3));
```

The advantage of this functional notation and composition should not be underestimated: many applications require manipulation of programmer-defined mathematical structures and the notation used by programmers should model as closely as possible the notation used by mathematicians and engineers. If Ada did not allow functions to return structured types, our operations would have to be procedures, e.g.

```
procedure Add(Result: out Rational; left, right: Rational);
procedure Mult(Result: out Rational; left, right: Rational);
```

and a use of the operation would be written as a procedure call, which cannot be composed. Our nice composed expression above would have to be written

```
Add(TemporaryResult, R2, R3);
Mult(R1, TemporaryResult);
```

which is much more cumbersome and surely does not look mathematical.

A work-around in Pascal and Modula-2 is to pass *pointers* to the structured objects as function arguments and results. This technique creates problems such as aliasing and dynamic allocation. Such excessive use of pointers is poor software engineering; it is also difficult to explain to students why it should be necessary.[2]

We note that Ada also provides for operator symbol overloading, so that e.g.

```
function "+"(left, right: Rational) return Rational;
function "*"(left, right: Rational) return Rational;
```

is permitted, with corresponding use

```
R4 := R1 * (R2 + R3);
```

making for a very mathematical-looking expression. This feature falls into the category of convenient "syntactic sugar;" it is less fundamental or necessary than the unrestricted function return value.

Ada also allows array objects to be returned from functions, so that one can write and use vector and matrix operations very conveniently and intuitively. This is related to the general Ada array capabilities, about which more below.

Function result types: the Competition. *Standard Pascal* does not permit records or arrays to be returned from functions. Neither do the Pascal derivatives *Turbo Pascal* and *Modula-2*. The proposed *C* standard allows records—but not arrays—to be returned. In the present example, C would allow the rational type but not the vector or matrix.

---

2 Even where pointers are necessary in Ada (in linked lists, for example) they are easier to use. Variables and record fields declared as pointers are *always* initialized to the null value. Students are thus robbed of the unwelcome learning experience of program crashes due to uninitialized pointers; we are quite happy to postpone this experience until the students learn C in upper-division courses.

Ada's unrestricted function return values makes Ada compilers undoubtedly more difficult to implement; we think the price is worth paying.

Packages: Ada. The separate package specification introduces the student to the idea of a "contract with the user." Students trained in (standard) Pascal tend to focus on "getting an answer" rather than "building a product." Using packages encourages a student to design a software component and carefully implement this contractual relationship with the component's user. The contract idea is reinforced by the separation of specification and body into separate files, separately compiled: students can see clearly that if something is not written in the spec, it's not visible to a client. Separate compilation means that programs dependent on a package need not be re-compiled if only the body , not the spec, is changed.

In CSci 159, programming assignments often require just the building of a package, with *no* client program at all except a test driver to validate the package. This is often not easy for students whose intuition drives them to focus on pretty interfaces and *getting an answer*, as opposed to developing a component intended for use by *another programmer* and not an end user. The grading system for projects must place heavy weight on the contractual relationship: the contract must describe *how a package is to be used, not the details of what it does*. CSci 159 allocates 30% of the grade to the quality of the package specification and its supporting user document.

Packages: the Competition. *Standard (ISO or ANSI) Pascal* has, of course, no notion of a package. *Turbo Pascal* provides a package-like structure called the "unit" (borrowed from UCSD Pascal), but the interface (specification) and implementation (body) must be in a single file. This diminishes the abstraction value—the student does not see the two sections as physically distinct—and also requires recompilation of dependent program segments every time something is changed, even if the change is only a detail in the implementation. A disadvantage of Turbo Pascal in general is that it is not available on Unix and other shared machines, and also that, at least until now, version k+1 has differed *significantly* from version k. And the IBM-PC and Macintosh versions are not even compatible: even if one ignores special operations for graphics, etc., there are *syntactic* differences between the two.

Modula-2 provides the *library module*, with definition (specification) and implementation (body) modules (files), separately compiled. This capability is quite similar to Ada, in spite of differences in the way import and export directives are written. Compilers are widely and inexpensively available and support a (generally) common language. A serious liability is the treatment of private types (see below).

C provides only a very rough equivalent to packages, namely the separation of groups of subprograms and type declarations into different files. Compilers are legion; the language supported is reasonably standard. Enforcement of interfaces, however, is strongly compiler-dependent.[3]

Private types: Ada. The private type, with its hidden implementation, is of course intimately related to the package. Ada allows any type to be made private or limited private; in particular, structured types can be private, and this forms the basis for an abstract data type scheme.

---

3 C++, the recently-developed extension to C, provides an object-oriented programming language more similar to Smalltalk than to Ada. C++ may become an important competitor, but is not yet widely available. A disadvantage for students is the less-than-obvious syntax.

The software-component philosophy embodied in the package and the private type pays off handsomely in more advanced courses, *even if the student goes on to develop programs in other languages*. Private types are an important subject in CSci 159; we see anecdotal evidence that CSci 159 graduates who choose to use C, for example, in senior projects, write better C because of their Ada exposure.

Private types: the Competition. *Standard Pascal* provides no private types. *Turbo Pascal* allows a unit to export a type, but its internal structure is visible to clients. One could hide, e.g., the fraction record type definition in a unit whose existence is not advertised, then make the fraction type itself a pointer to the hidden record type. This dodge is unsatisfying : it requires an extra unit, spreading the code for a single abstract type into two units, and carries along all the disadvantages of pointers.

Modula-2 improves the situation, but only a bit. A private type may be declared in a definition module, but its type is *required* to be a pointer to another type declared in the implementation module. At least the code for a single abstraction appears in a single library module, but the pointer difficulties persist.

C provides no notion of a private type. A work-around similar to the one described for Turbo Pascal could be invented, but it would surely be cumbersome.

An important consequence of the generality of function results and private types is that *access types (pointers) are unnecessary except to implement linked structures.* We believe that it is inappropriate to have to trade the niceness of functional notation for the forced clumsiness of pointers, solely because of a language limitation.

Array handling: Ada. Ada provides the "unconstrained array type" for an arbitrary number of dimensions. While the *number* of dimensions of an array must be specified in the type declaration, the bounds may be left unspecified until variables are declared. Further, unconstrained array types may be used in subprograms as formal parameters and function results. This facilitates a very natural implementation of vector and matrix packages, an important application often studied in data structures courses. For example, consider a package exporting a matrix type

```
package Matrices is

   type Matrix is
      array(Integer range <>,          -- bounds left open
            Integer range <>)          -- till variable
            of Float;                  -- is declared

      ...

   function "+" (left,right: Matrix) return Matrix;

   Conformability_Error: exception;

end Matrices;
```

Here we have combined many of the capabilities of Ada: the package, the unconstrained array type, overloaded operator symbols, unrestricted function result types, and the definition of application-dependent exceptions. In the package body, below, the code for the addition operator is given. Note the use of the attribute functions First, Last, and Range, which give the low bound, high bound, and bounds range, respectively, for the two dimensions. The subprogram can simply ask its actual parameters what their bounds are, then operate accordingly—in the event, create a temporary matrix sized according to the bounds of the inputs, fill it with values, then return this new matrix to its caller. Given three matrix objects

```
M1, M2, M3: Matrix(-5..5);
```

then the statement

```
M1 := M2 + M3;
```

can be written in the natural mathematical style. Note in the body of the addition operator that Conformability_Error is raised if the addition of the two matrices would be mathematically meaningless.

```
package body Matrices is
   ...
   function "+"(left,right: Matrix) return Matrix is

      Temp: Matrix(left'range(1), left'range(2));
      -- size of result gotten from size of input

   begin

      if left'First(1) /= right'First(1) or
         left'Last(1)  /= right'Last(1)  or
         left'First(2) /= right'First(2) or
         left'Last(2)  /= right'Last(2
      then
         raise Conformability_Error;
      end if;

      for row in left'range(1) loop
         for col in left'range(2) loop
            temp(row,col) := left(row,col) + right(row,col);
         end loop;
      end loop;

      return temp;                          -- array!

   end "+";
   ...
end Matrices;
```

Array handling: the Competition. Neither *Standard Pascal* nor *Turbo Pascal* nor *C* has any equivalent at all to the unconstrained array type (which actually resembles a feature in PL/1). *Modula-2* provides the "open array parameter" for subprograms, in which a *one-dimensional* array parameter may be passed without knowing its bounds; there is a rough equivalent to the attribute functions in this case. But this is permitted only for one-dimensional arrays, so the ability

to create a general matrix package in a natural way is severely limited.[4]

Following the body of this paper is a chart comparing, in summary form, the various features we have discussed here. We have concentrated here on a selected few Ada features we believe are especially useful in teaching data abstraction. We have not paid particular attention to linked data structures, as these are essentially the same in all modern languages. For brevity we have not included a discussion of generics; this subject warrants a paper in its own right.

HOW DO THE STUDENTS TAKE TO Ada?

Our undergraduate curriculum encourages students to learn a number of programming languages, because we believe that multilingual graduates are more openminded and accepting of change than those steeped in a single language with only the most superficial exposure to others.

Recently we have made the syntactic transition to Ada a bit easier by distributing a diskette of about fifty "small" Ada programs which cover the inner syntax of the language and the structure of the input/output libraries. Some of these programs are "booby-trapped" with deliberate compilation errors. The students are asked to compile and try these programs; if they can understand them all, including the reasons for the various errors, they know the rudiments of the Ada "Pascal subset" and are ready to dive into writing packages. These small programs also serve as templates for writing other programs, especially those using various kinds of input loops. A diskette of these programs is available from the author.

After a bit of grumbling about having to learn a new language for CSci 159, our students take readily to Ada once they begin to sense its power for building systems. Once students have picked up the rudiments, they often comment that syntactically, Ada is *easier* than Pascal; we tend to agree. And increasingly they choose Ada for upper-division projects where they are given a choice of language.

AVAILABILITY AND COMMONALITY OF COMPILERS

Recall that *compiler validation* means testing the compiler for conformance to a standard. In the case of Ada, neither subsets nor supersets are permitted by the standard [Nyberg89]. This gives a teacher confidence that all current Ada compilers will treat any reasonable classroom example or project in the same way. This unusually high degree of commonality stands in sharp contrast to the well-known difficulties with Pascal and Modula-2.

There are currently nearly 300 Government-validated Ada compilers in existence, many of which are available on systems commonly used in computer science education. There are, for example, at least four validated Ada systems for the IBM PC family; educational prices for three of these are within the budgets of typical insitutions or even individuals. Environmental support (editors, debuggers, etc.) has greatly improved in the last two years, as has performance of both compilers and resulting executable programs.

Vendors of Ada compilers have recently become aware of the needs and budget constraints of

[4] Rumor has it that this restriction to one-dimensional arrays will be relaxed. If the rumor is true we applaud the progress.

educational institutions, and are developing educational price lists and site-license arrangements that have begun to be quite competitive with each other and with compilers for other languages. The result is that integrating Ada into the computer science curriculum is now both technically feasible and economically interesting.

Our decision to introduce Ada progressively starting in 1981 may have been a bit of a gamble; looking back from 1990 we have no cause to regret the decision.

BIBLIOGRAPHY

[Aho83] Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *Data Structures and Algorithms*, Reading, Mass.: Addison-Wesley, 1983.

[DoD83] U.S. Department of Defense. *Reference Manual for the Ada Programming Language*. ANSI/MIL-STD 1815A, 1983.

[Feldman80] Feldman, M.B., "Teaching Data Abstraction to the Practicing Programmer," *Proc. 11th SIGCSE Tech. Symp. on Computer Science Education*, Kansas City, Feb. 1980.

[Feldman84] Feldman, M.B., "Packages, Abstract Types, and the Teaching of Data Structures," *Proc. 15th SIGCSE Tech. Symp. on Computer Science Education*, Philadelphia, Feb. 1984.

[Feldman85] Feldman, M.B., *Data Structures with Ada*, Englewood Cliffs, NJ: Prentice-Hall, 1985.

[Horowitz77] Horowitz, E., and S. Sahni, *Fundamentals of Data Structures*, Potomac, Md., Computer Science Press, 1977.

[Lang89] Lang, J.E., and R.K. Maruyama, "Teaching the Abstract Data Type in CS2," *Proc. 20th SIGCSE Tech. Symp. on Computer Science Education*, Louisville, Feb. 1989.

[Nyberg89] Nyberg, K.A., ed., *The Annotated Ada Reference Manual*, Vienna, VA.: Grebyn Corporation, 1989.

[Owen87] Owen, G.S., "Using Ada on Microcomputers in the Undergraduate Curriculum," *Proc. 18th SIGCSE Tech. Symp. on Computer Science Education*, St. Louis, Feb. 1987.

[Tenenbaum81] Tenenbaum, A.M., and M.J. Augenstein, *Data Structures Using Pascal*, Englewood Cliffs, NJ: Prentice-Hall, 1981.