TEACHING LINKED LISTS AND RECURSION WITHOUT CONDITIONALS OR NULL*

Stephen Bloch
Dept. of Mathematics & Computer Science
Adelphi University
Garden City, NY 11530
(516) 877-4483, sbloch@adelphi.edu

ABSTRACT

We describe a natural and principled approach to teaching linked data structures and recursion in CS0, CS1 or CS2, and compare the difficulty of using this approach in C++, Java, and Scheme.

1. INTRODUCTION

A first-year programming course can seem to students like a bunch of unrelated topics in a seemingly arbitrary order, including (for purposes of this paper) structures, inheritance, polymorphism, and recursion. The author finds that with care, these topics can be reduced to a small set of concepts, in which one topic flows naturally into the next. In particular, the most natural way to introduce recursion is not on numbers, with unmotivated and inefficient examples like Fibonacci and factorial, but rather on recursive data structures. Recursive data structures, in turn, may be quite difficult or quite easy to teach, depending on the programming language.

The approach described herein has several advantages. It enables students to define a wide variety of data types from a small set of concepts. It catches the most common beginner mistakes in data structures at compile-time, rather than run-time. And perhaps most importantly, it introduces recursion in a completely natural way-indeed, recursion is simply the application of previously-learned techniques to one new data type.

The author has taught this material variously in a CS2 course in C++, in a CS1 course in Java, and in a CS0 course in Scheme (see section 4 for discussion of the differences). However, all the examples in this paper are presented in Java.

For most of the techniques described herein, the author claims no originality: some of them are described in a C++ context in [BD96], some in a Java context in [FF98], and some in a Scheme context in [FFFK01]. My purpose in this paper is to spread awareness of those techniques in the CCSC community.

1.1 What's wrong with conditionals?

Nothing, really. In practice, I usually do teach conditionals before this material. The point is that if one chooses to teach in an object-oriented language in a strongly objects-first manner, with inheritance and polymorphism appearing in the first month or two, one can introduce linked lists and recursion, and write powerful operations using them, before introducing the "if" statement. Only a few of the more advanced examples given below require "if".

1.2 What's wrong with `null`?

Two things are wrong with `null`. First, in Java and many implementations of C++, it's the default value for uninitialized object (pointer) variables. To use it also as a marker for the end of a linked list is to overload it in a confusing and bug-prone way. The `null` pointer has traditionally been used as an end-of-list marker because it was the only nameable pointer that could be guaranteed (platform-independently) not to point to a real record, which in turn is presumably because one could test a pointer for `null`-ness with a single machine language instruction. Such a machine-based argument holds no weight today in teaching a first-year programming course.

A more serious objection from the perspective of teaching OOP is that if an empty list is represented as a `null` pointer, one cannot invoke object-oriented methods on lists without first checking for `null`-ness; the handling of empty lists cannot be encapsulated in a consistent, object-oriented way.

2. A PRINCIPLED APPROACH TO DATA STRUCTURES

When the author first studied data structures in the early 1980's, they tended to be taught as a jumble of unrelated techniques and tricks. But many sophisticated data structures can be built by combining two or three basic concepts. Equally important, *code* to work on these data structures can be written almost automatically, using the principle that *the shape of the data determines the shape of the code*.

2.1 Parts and Choices

In the first few weeks of a first-year course, I familiarize students with two or three primitive data types (numbers, strings, symbols, booleans, etc. depending on language) and the low-level coding patterns to operate on and test them. For example, the subtraction operator makes sense to apply to numbers, the concatenation operator works on strings, and the "and" operator can be applied to booleans. Test cases for booleans must include a "true" case and a "false" case; test cases for code that involves subranges of numbers must include one in each interval and one on each border; etc. Students become accustomed to being asked "what type is this variable?" and "what type is this expression?", and they are trained to write test cases based on data types *before* writing code for the function to be tested.

I then introduce *structures*-new data types defined by their *parts*, e.g. "a point consists of two numbers, 'x' and 'y'," or "a circle consists of a point, its 'center', and a number, its 'radius'." (In Java, of course, these are classes with-as yet-no methods.) Along with the syntax of defining a structure, I introduce two more coding patterns: how to get at the fields of an existing structure, and how to specify the fields of a new structure (remember, "the shape of the data determines the shape of the code"). These must be introduced together, because test cases for a function that operates on a structure necessarily involve creating one, and judging the correctness of a function that returns a structure necessarily involves looking at its fields. Depending on the instructor's priorities, a Java or C++ course can use public instance variables for simplicity, or introduce coding patterns for constructors, "getter" methods, and perhaps simple output formatting with `toString()` or an equivalent. Students are then assigned numerous simple functions that take in and/or return structure objects; further exercises require students to design a structure, and implement operations on it, from a word problem.

Next, I introduce *variant types* - new data types defined by *choices*, e.g. "a shape is either a circle or a rectangle"-along with coding patterns for operating on and producing instances of such types. In Java and C++, the language-supported way to represent such a variant type is as an abstract base class (or interface, if you prefer) with two or more concrete subclasses. (Unfortunately, neither language allows any of the variants to be a primitive or predefined type.) Code to operate on such an object can be written with at least two different coding patterns, each of which has some pedagogical advantages: using conditionals and **`instanceof`** operators, or using the more "proper" OOP coding pattern, an abstract method in the abstract base class, with concrete method definitions in each of the concrete classes. In either case, to create an instance of such a variant type, one simply creates an instance of one of its variants. Test cases for functions operating on a variant type must include at least one of each variant; test cases for functions producing a variant type must include, for each variant, at least one for which the "correct answer" is that variant. Students are assigned numerous simple functions that take in and/or return variant types, and they become accustomed to being asked "is this type defined by parts or by choices?" Further exercises require students to design a variant type, and implement operations on it, from a word problem; others may involve variant types in which the subtypes are themselves complex structures, or structures in which the fields are themselves of variant types.

2.2 Combining parts and choices

Students know what a list is. A little guided classroom discussion leads to the conclusions that lists can have any number of things on them, that common operations include adding and deleting items, and that even when everything is crossed off a list, what's left can still be considered a list-after all, one could add more items to it. We take this "empty list" as our foundation. A list is either empty or not-a classic example of *definition by choices*. If it's empty, there's not much more to say, but if it's non-empty, it must have a first element, and one can think of the "rest" of the list, everything after the first element, which may be empty or not-i.e. it's a list-and thus the type "non-empty list" is *defined by parts.*

Students already know how to do both of these things, so there's no new syntax to learn, and the already-learned coding patterns for private instance variables and standard constructor and getter methods produce the following:

```
abstract class StringList {
   }

class ESL /* EmptyStringList */ extends StringList {
   public ESL () { }
   }

class NESL /* NonEmptyStringList */ extends StringList {
   private String first;
   private StringList rest;

   public NESL (String firstArg, StringList restArg) {
      this.first = firstArg;
      this.rest = restArg;
      }
   public String getFirst () {
      return this.first;
      }
   }
```

Students are then assigned to create a variety of instances of the new type and examine their parts. This is particularly easy in a development environment that allows interactive testing, such as BlueJ, but it can also be done in a main method or function:

```
StringList plainPizza = new ESL();
StringList pizza1 = new NESL ("cheese", plainPizza);
StringList pizza2 = new NESL ("mushrooms", pizza1);
StringList pizza3 = new NESL ("pepperoni", pizza2);
StringList   fromScratch   =   new   NESL("garlic",   new   NESL("cheese",   new
ESL()));
...
System.out.println   ("pizza1.getFirst()   =   "   +   pizza1.getFirst()   +   "
(should be cheese)");
..
```

2.3 Recursion, the obvious way to work on recursive data

The next step is to write functions that work on lists. Students already know coding patterns for both structures and variants, so we apply those coding patterns straightforwardly: since one of the fields of a non-empty list is itself a list, the obvious thing to do with that field is to call a method in the list class. At the moment, the only method in the list class happens to have the same name as the method in the non-empty list class that we're in the process of writing, but there's no rule against that. (Some sharp students will object to the circularity; most won't see any problem with it.) It remains only to decide how the result for a whole list is related to the result for the rest of the list.

Since I usually teach in a computer lab, I demonstrate the development of one such function, e.g. `length`, then tell the students to develop another, e.g. `concatenateAll` (for Strings) or `sum` (for numbers), and we continue taking turns in this fashion. In the Appendix, I present a variety of methods in approximately the order that I would introduce them in the classroom. The point is that recursion is *not a new, scary technique*, but rather the *most natural* way to operate on fields of a self-referential data structure. (If you don't tell students that recursion is hard, they won't think it is.)

Note also that two sorting algorithms fall almost automatically out of the already-learned coding patterns-insertion-sort from a pattern for taking apart the input, and its dual, selection-sort, from a pattern for putting together the output-and that they require respectively only twelve and fifteen lines of code (half of which are function headers).

2.4 Compile-time detection of common beginner errors

One powerful advantage of this approach (in Java or C++) is that the most common beginner errors in list processing – e.g. dereferencing the "next" pointer without first checking whether it's null – correspond to mistakes that are caught by the compiler, rather than at run-time. (According to [Fel02], the "principles of programming languages" community has been pointing this out for twenty years, but with little impact to date on first-year pedagogy.)

For example, if a student refers to the `first` element of the `rest` of a non-empty list, the compiler will complain that `rest` is of type StringList, which has no `first` member. If the student is certain that there *is* such a first element in a particular context, an explicit cast tells the compiler "Trust me; it's really a NESL" and allows reasonably painless access to the element. If the student is incorrect, the result will be a ClassCastException, which at least is more informative than a NullPointerException.

Similarly, if a student completely fails to write code for the "empty" case, a Java compiler will complain that an abstract method was not defined in the subclass and the subclass should therefore be declared abstract. A C++ compiler will complain that a pure virtual method was not defined in the subclass, the subclass *is* therefore abstract, and it therefore cannot be instantiated. Perhaps a confusing error message in either case, but students should already be familiar with it from their previous exercises on variant types.

I wrote Java code illustrating both of these common errors, once with a "traditional" null-terminated implementation and once with a polymorphic implementation, with the following results:

Mistake	Null-terminated	Polymorphic
calling `getFirst()` on an empty list	Exception in thread "main" java.lang.NullPointerException	cannot resolve symbol: method getFirst() location: class StringList
not checking for empty case	Exception in thread "main" java.lang.NullPointerException	EmptyStringList should be declared abstract; it does not define concatAll() in StringList

2.5 Formatting and convenience

The way of building lists in the Appendix is conceptually simple, but admittedly long-winded and inconvenient for use in complex algorithms. Briefer and more convenient ways to build lists are easy:

```
abstract class StringList {
   ...
   public static ESL empty() {
      return new ESL();
      }
   public NESL add(String firstArg) {
      return new NESL(firstArg, this);
      }
   ...
   }
...
pizza1 = plainPizza.add("cheese");
pizza2 = pizza1.add("mushrooms");
pizza3 = pizza2.add("pepperoni");
fromScratch = StringList.empty().add("cheese").add("garlic");
```

If the proliferation of ESL objects bothers the instructor or the students, it can motivate discussion of the "singleton pattern":

```
class ESL {
   ...
   private static ESL only = new ESL();
   public static ESL empty() {
      return only;
      }
   private ESL () { }
   ...
   }
...
if (mysteryList == ESL.empty()) // should work correctly, since there
                                // is only one
```

Output is a trickier problem. The `toString()` method in the Appendix has the advantage that the output looks exactly like the code students would use themselves to create the corresponding list, but it *is* verbose and involves deeply-nested parentheses, and students may be annoyed at this. In general, the author finds output-formatting code a nearly-infinite sink of classroom time, and one which teaches few transferrable skills, but here's an alternate definition of `toString`, whose development in class affords an opportunity to discuss the design of auxiliary functions, postconditions, etc. in a setting motivated by students' desire to be able to read their lists more easily.

```
abstract class StringList {
   ...
   public abstract String toString();
   // produces output of the form "list(cheese,garlic,pepperoni)"
   protected abstract String toStringHelper();
   // toStringHelper concatenates all the elements of the list,
   // with a comma before each one, e.g. ",cheese,garlic,pepperoni".
   ...
   }

class ESL extends StringList {
   ...
   public String toString() {
      return "list()";
      }
   protected String toStringHelper() {
      return "";
      }
   ...
   }

class NESL extends StringList {
   ...
   public String toString() {
      return "list(" + toStringHelper().substring(1) + ")";
      }
   protected String toStringHelper() {
      return "," + this.first + this.rest.toStringHelper();
      }
   ...
   }
```

3. DIFFERENT KINDS OF RECURSION

One benefit of introducing recursion in this context is that it's a strictly limited form of recursion-"explicit structural recursion" - which *cannot* go infinite, because recursion is always on an explicit sub-part of a finite data structure. Once students are comfortable with this, one can introduce a Peano-style definition of "natural number":

> A natural number is either 0 or a positive natural number.
> A positive natural number has a predecessor, which is a natural number.

Clearly, this definition is almost isomorphic to the above definition of a list: the "natural number" datatype is defined by choices, of which one choice is primitive and the other is defined by parts, of which one part is again a "natural number". And if time allowed, one could implement this data type polymorphically in exactly the same way as above. On the other hand, since students think they already know what a natural number is, it may be more intuitive to use the built-in 0, 1, and +, with explicit conditionals in place of method dispatch. In either case, recursion on natural numbers is just as "natural" as recursion on linked lists: since the predecessor is a natural number, one will presumably call on the predecessor some method that works on natural numbers, e.g. the one we're in the process of writing. A good example (although it shouldn't be the first) is the `removeFirstN` method in section 2.3.

The next, slightly less "natural", form of recursion might be called "implicit structural recursion": recursion on a proper sub-part of a finite data structure, but in which the data structure itself doesn't dictate what to recur on. For example, one can define a wide variety of string functions by recursion on Strings, looking at the first character and recurring on the rest of the String, an obvious analogue of list recursion. (The author's favorite example of this is pattern-matching with "?" and "*" wildcards-a fairly simple task using recursion, but difficult and non-intuitive to do iteratively.) Next one would generalize to recursion on other strict substrings, and then to strict subranges of arrays, as in a mergesort.

As described in [FFFK01], the next form of recursion to introduce might be "accumulative recursion", which is simply structural recursion with the addition of an "accumulator" parameter which is built up on the way *down* the recursion, and returned at the bottom, rather than an answer being built up on the way *back* as in pure structural recursion. However, since recursion is still only on strict sub-parts of a finite data structure, it is still guaranteed to terminate. The `smallestHelper` method in the Appendix is a simple example.

Finally, one can introduce the most general recursion scheme, "generative recursion", in which the new problem to be solved is generated on the spot, rather than being a strict sub-part of the old problem; quicksort is a classic example. To be sure that such a recursive function terminates, one must define a suitable "smaller" relation on problems, prove that the problem thus generated is always strictly "smaller" than the old problem, and furthermore prove that the "smaller" relation has finite descending chains! This requires more theorem-proving background than most first-year CS students have.

4. C++, JAVA, AND SCHEME

I've used this approach in several different first-year programming courses, variously in C++, Java, and Scheme. The choice of language impacts the ease of teaching this approach, at what stage it can be used, and how it fits in with other topics in the course.

[BD96] describes a similar approach in C++, to be incorporated into a CS2 course (as I have also done it). The approach cannot easily be used any earlier in C++ because it requires run-time polymorphism, which requires pointers, dynamic allocation and deallocation, all of which are difficult to fit into a first-semester programming course. As a result, the extremely

natural form of recursion this technique suggests cannot be used to introduce recursion in a C++-based course unless recursion itself is delayed to the second semester.

[FF98] describes the approach in Java. Since polymorphism in Java requires less syntax than in C++, and Java's garbage collector frees the programmer from concern about deallocation, the approach becomes viable in a CS1 course, and can therefore be used to introduce recursion. (The garbage collector makes a big difference: the reader is invited to design a safe deallocation protocol for the `append` and `removeFirst` methods in section 2.3, and then to explain it to first-semester students!)

[FFFK01] describes a similar approach in Scheme (although using conditionals and type-tests in place of the method-dispatch in the above Java examples). Doing all this in Scheme requires even less syntax than in Java, and memory allocation and deallocation issues are even more hidden than in Java. As a result, one can successfully teach recursion on linked lists and other self-referential data structures halfway through a CS0 course for non-majors (as I have done numerous times), or in a first programming course at the high school or even middle school levels, as attested by hundreds of teachers in the TeachScheme! project [FKF+].

5. TAKING IT FARTHER

5.1 Other self-referential data structures

[FF98] and [FFFK01] go on to define binary and n-ary trees, in a manner closely analogous to the above definition of lists; students can easily write functions traversing these trees by using the same principles of "parts" and "choices" that they learned early in the course. These are left as an exercise for the reader (and, of course, are described in [FF98] and [FFFK01]).

5.2 Functional and stateful programming

The reader will have noticed that the list operations in the Appendix are all *purely functional*: they do not modify their arguments, but rather produce completely new lists. The functional approach seems to work well with polymorphic lists, but eventually one must introduce mutation, either for efficiency in operating on simple linked lists and trees, or for cyclical linked data structures such as doubly-linked lists.

A problem with mutating a polymorphically-implemented linked data structure is that both C++ and Java forbid a method to modify the class of the object on which it was invoked: one cannot write, for example, "`this = new NESL(...);`" The same problem applies to "traditional" null-terminated linked data structures, if they are implemented in an object-oriented way: a method cannot replace all references to the object on which it was called with `null`. The solution in both cases is the same: hide the real data structure inside a "front end" class. This can be done with top-level classes at first, and later (depending on the instructor's priorities) the functional part can be moved into inner classes or non-public parts of a package, so only the "front end" class is visible. Doubly-linked lists, for example, can be implemented polymorphically in a type-safe way, using two interfaces "HasPredecessor" and

"HasSuccessor", and three types of nodes: "LeftEnd" (which implements HasSuccessor), "RightEnd" (which implements HasPredecessor), and "DataNode" (which implements both), all hidden behind a front-end "DoublyLinkedList" class. This data structure is slightly cleaner in C++ than in Java, as it's one of the rare examples that actually benefit from true multiple inheritance.

6. CONCLUSIONS

A guiding principle throughout this work has been the use of *coding patterns tied to data types*; the principle that "the shape of the data determines the shape of the code". We use this principle to teach students to work with structures, then with variant types (or *vice versa*, as in [FF98]). When the notions of "structure" and "variant type" (or, as I put it in the classroom, "parts" and "choices") are combined in a particular way, the natural outcome is a self-referential data structure such as a linked list or a binary tree. And the *inevitable* result of following the coding patterns on this data structure is a safe, easy-to-understand, recursive function.

After students gain familiarity with this form of recursion, they can be introduced to less self-evident forms, such as recursion on natural numbers or sub-ranges of an array. And, like data types, these different forms of recursion can be seen not as a hodgepodge of unrelated tricks and techniques, but as an organized taxonomy that follows a few simple rules.

BIBLIOGRAPHY

[BD96] A. Michael Berman and Robert C. Duvall. "Thinking about Binary Trees in an Object-Oriented World". In Proceedings of the 27th SIGCSE Technical Symposium on Computer Science Education, pp. 185-189, 1996.

[Fel02] Matthias Felleisen, 2002. Electronic mail.

[FF98] Matthias Felleisen and Daniel P. Friedman. A Little Java, A Few Patterns. MIT Press, 1998.

[FFFK01] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, and Shriram Krishnamurthi. How to Design Programs: an Introduction to Programming and Computing. MIT Press, 2001.

[FKF+] Matthias Felleisen, Shriram Krishnamurthi, Matthew Flatt, Robeert Findler, and Kathy Fisler. "The TeachScheme! Project". Web page http://www.teach-scheme.org

APPENDIX

Following is sample Java code for a variety of simple methods, in roughly the order that I would introduce them in the classroom. To save space, I've omitted comments and test cases, and used a compressed indentation style.

```
abstract class StringList {
   public abstract int length();
   public abstract String concatenateAll();
   public abstract String toString();
   public abstract boolean containsOnion();
   public abstract boolean contains(String toFind);
   public abstract StringList toUpperCase();
   public abstract StringList prefixAll(String toAdd);
   public abstract StringList append(StringList other);
   public abstract StringList stutter();
   public abstract StringList stutterOnion();
   public abstract StringList removeAll(String toRemove);
   public abstract StringList removeFirst(String toRemove);
   public abstract StringList removeFirstN(int n, String toRemove);
   public abstract StringList insertionSort();
   protected abstract StringList insertInOrder(String toInsert);
   public abstract StringList selectionSort();
   protected abstract String smallestHelper(String anElement);
   }

class ESL extends StringList {
   public ESL () { }
   public int length() { return 0; }
   public String concatenateAll() { return ""; }
   public String toString() { return "new ESL()"; }
   public boolean containsOnion() { return false; }
   public boolean contains(String toFind) { return false; }
   public StringList toUpperCase() { return this; }
   public StringList prefixAll(String toAdd) { return this; }
   public StringList append(StringList other) { return other; }
   public StringList stutter() { return this; }
   public StringList stutterOnion() { return this; }
   public StringList removeAll(String toRemove) { return this; }
   public StringList removeFirst(String toRemove) { return this; }
   public StringList removeFirstN(int n, String toRemove) {
      return this; }
   public StringList insertionSort() { return this; }
   protected StringList insertInOrder(String toInsert) {
      return new NESL(toInsert, this); }
   public StringList selectionSort() { return this; }
   protected String smallestHelper(String anElement) {
      return anElement; }
   }

class NESL extends StringList {
   private String first;
   private StringList rest;

   public NESL (String firstArg, StringList restArg) {
      first = firstArg;
      rest = restArg; }
   public String getFirst () { return this.first; }
   public int length() { return 1 + this.rest.length(); }
   public String concatenateAll() {
```

```
      return this.first + this.rest.concatenateAll(); }
   public String toString() {
         return "new NESL(\"" + this.first + "\"," + this.rest.toString()
+ ")"; }
   public boolean containsOnion() {
            return this.first.equals("onion") || this.rest.containsOnion();
}
   public boolean contains(String toFind) {
            return this.first.equals(toFind) || this.rest.contains(toFind);
}
   public StringList toUpperCase() {
                        return new NESL (this.first.toUpperCase(),
this.rest.toUpperCase()); }
   public StringList prefixAll(String toAdd) {
          return new NESL (toAdd + this.first, this.rest.prefixAll(toAdd));
}
   public StringList append(StringList other) {
      return new NESL (this.first, this.rest.append(other)); }
   public StringList stutter() {
      return new NESL (this.first,
                       new NESL (this.first,
                                 this.rest.stutter())); }
   public StringList stutterOnion() {
      if (this.first.equals("onion"))
         return new NESL(this.first,
                                                        new NESL(this.first,
this.rest.stutterOnion()));
      else
         return new NESL(this.first, this.rest.stutterOnion()); }
   public StringList removeAll(String toRemove) {
      if (this.first.equals(toRemove))
         return this.rest.removeAll(toRemove);
      else
               return new NESL (this.first, this.rest.removeAll(toRemove));
}
   public StringList removeFirst(String toRemove) {
      if (this.first.equals(toRemove))
         return this.rest;
      else
              return new NESL (this.first, this.rest.removeFirst(toRemove));
}
   public StringList removeFirstN(int n, String toRemove) {
      if (n <= 0)
         return this;
      else if (this.first.equals(toRemove))
         return this.rest.removeFirstN(n-1, toRemove);
      else
                    return new NESL (this.first, this.rest.removeFirstN(n,
toRemove)); }
   public StringList insertionSort() {
      return this.rest.insertionSort().insertInOrder(this.first); }
   protected StringList insertInOrder(String toInsert) {
      if (toInsert.compareTo(first) < 0)
         return new NESL (toInsert, this);
```

```
      else
                                      return    new    NESL    (this.first,
this.rest.insertInOrder(toInsert)); }
   public StringList selectionSort() {
      String min = this.smallest();
      return new NESL(min, this.removeFirst(min).selectionSort()); }
   private String smallest() {
      return this.rest.smallestHelper(first); }
   protected String smallestHelper(String anElement) {
      if (anElement.compareTo(first) < 0)
         return this.rest.smallestHelper(anElement);
      else
         return this.rest.smallestHelper(this.first); }
   }
```