# Using an Imperative Language to Teach Applicative Methods

Daniel Barrett
Department of Computer Science
The Johns Hopkins University

Robert Strandh
Department of Computer Science
University of Bordeaux

November 17, 1994

**Abstract**

Many universities teach traditional, imperative languages in their introductory programming courses, even though an applicative language may be preferable. In this paper, we present a compromise and show how applicative programming methods may be illustrated effectively using a subset of Pascal.

The approach focuses on order of evaluation, the `if` statement, functions, and recursion. We discuss several program examples actually given in class, and interesting problems encountered by students who had previous imperative programming experience. Although we are just beginning to investigate the results of our approach, we believe that we have been able to cut the time to explain Pascal syntax to a minimum, allowing more coverage of programming techniques.

## 1 Introduction

It can be argued that an applicative programming language offers the best introduction to programming, given its simple syntax and few restrictions on combining expressions [1]. Unfortunately, some universities require that a traditional imperative language be used in the introductory programming courses. Such a university-wide decision may be influenced by:

- conservative faculty;
- students who prefer a language that is "actually used," as opposed to an academic "teaching" language. This problem is very real in places that emphasize practical usefulness of the computer science program.

The purpose of this paper is to show that applicative programming techniques can be used within the framework of an imperative programming language (Pascal [3]).

This approach solves several problems inherent in introductory courses that must use imperative languages. The instructor is no longer forced to spend a large amount of time discussing syntax; instead, he/she can concentrate on programming techniques. More importantly, the students get a more accurate view of computer science, rather than memorizing syntax and writing small, single-purpose ("toy") programs.

In our method, we concentrate on the use of recursion together with the `if` statement. In addition, the notion of *order of evaluation* is stressed, since it is important in both applicative and imperative languages.

When using this approach, instructors may have difficulty inventing Pascal programming assignments that use this restricted syntax. Many introductory programming textbooks are of no

help; "interesting" programming problems generally do not arise until structured data types (arrays, records, pointers) are discussed [2, 4, *et. al.*]. Therefore, an important problem is to find interesting programming problems that require recursion but not structured data types.

We show two such examples that were given to chemistry and physics students. The first program computes the resistance in a circuit, and the second translates a description of an organic molecule into instructions for a hypothetical machine that can build the molecule. The problems illustrate both fundamental and advanced concepts, such as recursive descent parsing techniques and prefix-to-postfix conversion. In addition, characters and integers are the only data types needed. Both examples were well received by the students, especially since they were related to their field of interest.

Our approach has been tested in the courses *Introduction to Computer Programming* (600.107) at The Johns Hopkins University, and the introductory programming course for the second year students at The University of Bordeaux.

# 2 Our Approach

Our method has several features. First, we teach only a subset of the keywords of the language. Second, we de-emphasize iteration in favor of recursion. And third, we emphasize the notion of a *computation*; that is, the *order of evaluation* of an expression, which may be more explicit in an imperative language than in an applicative one. We recommend explaining the full evaluation process, even if the language allows it to be hidden to some extent.

## 2.1 Expression Evaluation

As with any applicative approach, we discuss the notion of an *expression*. Because of the arbitrary syntax of expressions in Pascal (or any other traditional language), we use a two-step method in which we distinguish an expression in prefix notation from a valid Pascal expression.

Thus, an expression is string of symbols that has a value. Expressions are *evaluated* to obtain their value. An expression is always evaluated inside of an *environment*, as defined below.

There are three kinds of expressions:

- A *constant* is an expression. Its value is itself. Examples are 51 and `'R'`.
- A *variable* is an expression. Its value is found by looking in the *environment*. An environment is simply a table (possibly empty) of all known variables and their current values.
- A *function application* is an expression consisting of two parts:
  - The *function name*, which has an associated *function rule*;
  - An ordered list of expressions called *arguments*.

  The value of a function is obtained by a two-step process.

  1. Evaluate the arguments (recursively) to obtain the argument values.
  2. Apply the function rule to the argument values.

This formalism translates to Pascal quite easily: constants are Pascal constants, variables are Pascal variables of simple types, and function applications are the invocations of Pascal functions.

The function rule is the body of the Pascal function declaration, and arguments are Pascal parameters.

The concept of an environment also translates nicely. At any instant during the execution of a Pascal program, there is an unambiguous set of variables whose scope includes the current block, and their values are well-defined (though the variables might not be initialized). The environment at that instant is a table containing those variables and their values.

## 2.2 Operators and Prefix Form

We also consider the use of Pascal operators (arithmetic, boolean) to be function applications. This introduces a small problem with applying this formalism to Pascal, because many operators are infix rather than prefix. For consistency, students are taught how to write all Pascal expressions in prefix form. For example, the expression `Cube(3 * number - 7)` would be written as `Cube( -(*(3, number), 7) )`.

Students were cautioned that this syntax was not part of the Pascal language, but was purely for consistency in our expression formalism.

## 2.3 A Teaching Tool: The Human Stack

To illustrate the process of expression evaluation, we use a method we call the *human stack*. In this method, students stand in the front of the classroom and simulate the evaluation process itself. To do this:

1. Write a simple Pascal program that contains expressions. It is best if the program contains no conditional statements, so it has only one possible output. For example:

```
program ExpressionExample(output);
var
        number : integer;

function Cube(x : real) : real;
begin
        Cube := x * x * x
end;

begin
        (* The environment contains only the variable "number". *)

        number := 5;
        writeln(Cube(3 * number - 7))
end.
```

2. Make a set of large, cardboard flash-cards. These is exactly one flash-card for each function application (function call or arithmetic/boolean operation) that will occur in the execution of the program. Side one of the card contains the name of the function, with the names and current values of its operands (in the environment) beneath. Side two contains the value computed by this function.

   For example, given the card for "$3 * number$, with $number = 5$", side one lists the function '$*$' with arguments 3 and *number* (value 5). Side two simply says 15.

3. Sort the cards into the order that the function applications occur during the program execution. You are now ready to begin.

4. In front of the class, on the blackboard, trace the program. As each expression is encountered, write it in prefix form, and say, "A new environment comes into existence!" At this cue, a student walks to the front of the room, takes the next card from your set, stands next to the "previous" student, and displays side one. Later, when the expression is fully evaluated (perhaps creating other environments), the student flips the card to reveal the value. That student now returns to his/her seat.

   As the functions invoke other functions, students will line up across the front of the room. Their appearance and disappearance exactly mirror the actions occurring on the system stack as functions are called (push) and return (pop). When the very last function exits, the stack is empty, and the program is over. (The term "stack" does not need to be mentioned.)

This exercise helps to teach that every expression has a local environment, and that expression evaluation in a correct program is unambiguous.

# 3 Example Homework Problems

An important goal of our approach is to give the students an idea of the problems facing a computer scientist. The traditional approach to teaching programming can give the impression that computer science is merely a collection of rules like *you must not put a semicolon before an else*, or that programming is little more than summing the elements of an array of integers.

To achieve this goal, we use examples that require and illustrate recursive programming techniques. This is relatively easy using an applicative language such as Scheme, but much harder in Pascal. Examples beyond *fibonacci* and *hanoi* typically use complicated pointer structures which are difficult both to understand and to teach. In addition, we would like examples based on familiar topics rather than straight computer science.

In this section, we describe two homework problems given to chemistry and physics students at the University of Bordeaux. These programs use no complicated data structures: just a few integers, reals and characters.

## 3.1 Resistor Networks

The first problem illustrates the evaluation of expressions in prefix notation. To disguise the computer science aspects of the example, we say that the expression represents a network of resistors connected in series or parallel.

### 3.1.1 Problem Statement

Resistors networks arise naturally in electrical engineering applications. For our purposes, a resistor network is either

- a single resistor,
- two resistor networks connected in parallel, or
- two resistor networks connected in series.

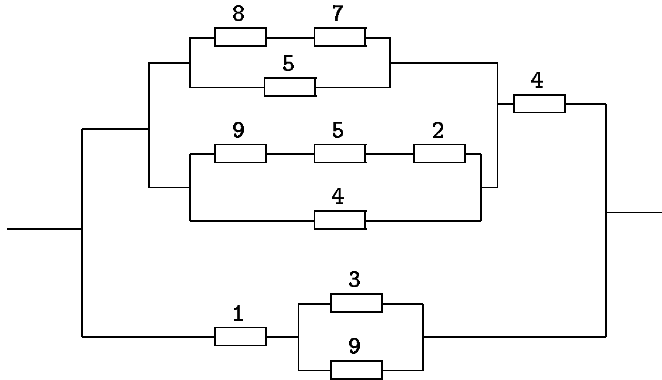
Many (though not all) real-life networks can be represented in this manner. Give an example of one that cannot be represented this way!

The rules for computing the resistance of a network are the usual ones. If $A$, $B$, and $C$ are resistor networks, $r$ is a resistor and $R(A)$ is the resistance of $A$ then

| | |
|---|---|
| $R(A) = R(r)$ | if $A$ is a single resistor, $r$. |
| $R(A) = R(B) + R(C)$ | if A is B and C connected in series. |
| $R(A) = (R(B) * R(C))/(R(B) + R(C))$ | if A is B and C connected in parallel. |

Why can we not use the formula: $R(A) = 1/(1/R(B) + 1/R(C))$ for parallel-connected networks?

Of course, figures are the best way to represent resistor networks; but in order to make it easier for a computer to manipulate them, we describe them as sequences of characters. A resistor is represented by a single digit that indicates the resistance (in Ohms) of the resistor. If $B$ and $C$ are two networks (of arbitrary size) connected in series, we write S$BC$. If $B$ and $C$ are connected in parallel, we write P$BC$. For example, the network:




Can be represented by

```
PS1P39SPP5S87P4S9S524
```

This becomes somewhat easier to read when parentheses are inserted:

```
(P(S1(P39))(S(P(P5(S87))(P4(S9(S52))))4))
```

**A grammar for resistor networks**

Write a formal grammar for the description (with or without parentheses) of a resistor network. Suggested symbols:

| | |
|---|---|
| Goal symbol: | Network |
| Other non-terminals: | Parallel, Serial, Resistor |
| Terminal symbols: | `P`, `S`, `1-9` |

Write a program to compute the resistance of the network.

Suggested structure: Write a function for each non-terminal symbol. A function computes the resistance of its corresponding entity, calling the other functions if necessary.

Notice that, although all the resistors are integer valued, the final resistance is a real number.

### 3.1.2 Proposed Solution

It is important to note that local variables are needed to hold temporary results. Without them, the program would be dependent on the order in which Pascal evaluates the arguments of a function application.

```
program resistances(input,output);
var
        c : char;

function resistance: real;
begin
        resistance := ord(c) - ord('0');
end;

function network : real; forward;

function parallel: real;
var
        r1, r2 : real;
begin
        r1 := network;
        r2 := network;
        parallel := r1 * r2 / (r1 + r2);
end;

function serial: real;
var
        r1, r2 : real;
begin
        r1 := network;
        r2 := network;
        serial := r1 + r2;
end;

function network;
begin
        read(c);
        while (c = '(') or (c = ')') do read(c);
        if c = 'P' then network := parallel
        else if c = 'S' then network := serial
        else network := resistance;
end;

begin
        writeln(network);
end.
```

## 3.2 Organic Molecules

The second problem illustrates prefix-to-postfix conversion and compilation techniques, disguised as a problem in organic chemistry. The assignment is to translate the formula of an organic molecule into a sequence of instructions for a hypothetical machine that builds the molecule.

### 3.2.1 Problem Statement

Organic molecules are naturally occurring recursive structures. We define a *radical* as a charged particle with a valence of -1; that is, it has one "free slot" available for bonding to something else. A radical is made of carbon (`C`), oxygen (`O`) and/or hydrogen (`H`) atoms. An *organic molecule* consists of two radicals bonded together. To simplify the problem, radicals have only simple bonds and no circular structures.

A carbon atom has valence -4, oxygen -2, and hydrogen -1. Thus, a hydrogen atom is a radical by itself, but the others are not. To turn an oxygen atom into a radical, a second radical must be bonded to it. To turn a carbon atom into a radical, three other radicals must be bonded to it, leaving one "free slot."

We represent our radicals as strings of the characters `H`, `O`, and `C`. In the case of an `O`, there is another radical following the `O`, and in the case of a `C`, there are three radicals following. Thus, the methyl radical is written `CHHH`, and the ethyl radical is written `CHHCHHH` or `CHCHHHH` or `CCHHHHH`, depending on the order in which the carbon atoms are listed. To make this more readable, one could insert parentheses as follows: `(CHH(CHHH))`, `(CH(CHHH)H)` and `(C(CHHH)HH)`.

A molecule is written simply as two consecutive radicals. For example, ethane could be written in several ways, such as `H(CHH(CHHH))` and `(CHHH)(CHHH)`. A well known molecule of great importance to Bordeaux (which one?) would be written `(CHHH)(CHH(OH))`.

#### A grammar for organic molecules

Write a formal grammar for organic molecules. Suggested symbols are:

| | |
|---|---|
| Goal symbol: | Molecule |
| Other non-terminals: | radical, `C`-radical `O`-radical `H`-radical |
| Terminal symbols: | `C`, `O`, `H`. |

#### Genetic engineering

We shall now imagine a machine that can produce molecules according to a specification. The machine consists of four *tubes* and a *work area*. The tubes are the `C`-tube, the `O`-tube, the `H`-tube and the radical-tube. The `C`-, `O`-, and `H`-tubes contain an infinite supply of carbon, oxygen and hydrogen atoms, respectively. The machine can be instructed to move an atom from either of these three tubes into the work area. For this purpose, we use the `GRAB` instruction. The format is

```
GRAB <atom>
```

where `<atom>` is `C`, `O`, or `H`.

The work area contains an atom that is being worked on. If the atom in the work area is a complete radical, it can be moved to the radical-tube with the SAVE instruction. The format is simply:

```
SAVE
```

The radical-tube is an infinitely long tube. This tube is so thin that only the last radical saved can be accessed. Thus, radicals must be used in the reverse order that they were saved.

To connect a radical to the atom in the work area, the machine has the instruction CONNECT, which takes the radical last saved and connects it to the next free binding of the atom in the work area. The format is simply:

```
CONNECT
```

Finally, there is an instruction to connect the two top radicals in the radical-tube to each other, forming a molecule. The instruction is:

```
GLUE
```

In order to fabricate the ethanol molecule H(C(CH(OH)H)HH), the instruction sequence would be as shown below. In order to make it easier to read, we show the contents of the radical-tube and the work area as we go along. Also, to get the order right, we always show the last connected radical immediately to the right of the atom.

| instruction | radical-tube | work area |
|---|---|---|
| GRAB H | : | H |
| SAVE | H : | - |
| GRAB H | H : | H |
| SAVE | H H : | - |
| GRAB H | H H : | H |
| SAVE | H H H : | - |
| GRAB O | H H H : | O |
| CONNECT | H H : | OH |
| SAVE | H H (OH) : | - |
| GRAB H | H H (OH) : | H |
| SAVE | H H (OH) H : | - |
| GRAB C | H H (OH) H : | C |
| CONNECT | H H (OH) : | CH |
| CONNECT | H H : | C(OH)H |
| CONNECT | H : | CH(OH)H |
| SAVE | H (CH(OH)H) : | - |
| GRAB H | H (CH(OH)H) : | H |
| SAVE | H (CH(OH)H) H : | - |
| GRAB H | H (CH(OH)H) H : | H |
| SAVE | H (CH(OH)H) H H : | - |
| GRAB C | H (CH(OH)H) H H : | C |
| CONNECT | H (CH(OH)H) H : | CH |
| CONNECT | H (CH(OH)H) : | CHH |
| CONNECT | H : | C(CH(OH)H)HH |
| SAVE | H (C(CH(OH)H)HH) : | - |
| GLUE | ****H(C(CH(OH)H)HH)**** | |

As you can see, the sequence of instructions for the machine produced the desired molecule. However, to produce the sequence of instructions, given the description of a molecule, is time consuming. We would like a program that reads the description of a molecule and produces the instructions for the machine.

### 3.2.2 Proposed Solution

The proposed solution is procedural rather than functional. Instead of values, the subprograms produce side effects in the form of `writeln` statements.

```
program organic(input,output);

procedure radical; forward;

procedure Hradical;
begin
        writeln('GRAB H');
        writeln('SAVE');
end;

procedure Oradical;
begin
        radical;
        writeln('GRAB O');
        writeln('CONNECT');
        writeln('SAVE');
end;

procedure Cradical;
begin
        radical;
        radical;
        radical;
        writeln('GRAB C');
        writeln('CONNECT');
        writeln('CONNECT');
        writeln('CONNECT');
        writeln('SAVE');
end;

function readchar : char;
var
        c : char;
begin
        read(c);
        while (c = '(') or (c = ')') do read(c);
        readchar := c;
end;
```

```
procedure radical;
var     c : char;
begin
        c := readchar;
        if c = 'H' then Hradical
        else if c = 'O' then Oradical
        else if c = 'C' then Cradical
end;

procedure molecule;
begin
        radical;
        radical;
        writeln('GLUE');
end;

begin (* main *)
        molecule;
end.
```

## 4 Problems With The Approach

Overall, the Johns Hopkins courses were overwhelmingly enjoyed by the students.[1] However, some students had interesting problems understanding parts of the approach, and we describe them here.

Despite frequent reminders that Pascal uses infix notation for arithmetic operators, some students used prefix form in their first programs. This error was quickly remedied; when the Pascal compiler refused to accept a student's code, a Teaching Assistant pointed out the problem, and the student did not make the same mistake again.[2]

Another problem is that students could not see a reason for learning prefix form in the first place. Without knowledge of several languages, novice students found it difficult to appreciate the simplicity and generality of this expression model.

Students with previous experience in Pascal, BASIC, or FORTRAN were confused during the first part of the course. Their idea of "programming" was very different from the material being presented. They felt frustrated because they wanted to use familiar loops and gotos instead of recursion. However, this confusion lessened as the course progressed, and many of these students became "converts" to the new methods.

Students complained that the lectures did not follow the book. As one student wrote, "I got [functions and procedures] hopelessly messed up for two and a half weeks [because we] reversed the order that the book covered them." Unfortunately, we have encountered no Pascal textbook that emphasizes applicative programming (although one author is considering revising his textbook in this manner[5]). To combat this problem, printed lecture notes were distributed every day in class.

Students complained that they did not learn all of the Pascal language in the course. We did not cover enumerated ordinal types, sets, variant records, file operations, and other Pascal-specific topics. Our argument was that this was a course in learning how to program, as opposed to learning

[1] Rated 4.7/5.0 or higher in the *Johns Hopkins University Course Guide*.

[2] Perhaps this misunderstanding illustrates that the students found the prefix form to be more consistent and "natural" than Pascal's actual syntax; or perhaps they simply did not pay attention in class!

Pascal, and that the students were free to read about these topics in the textbook.

## 5 Conclusions

By concentrating on only a subset of Pascal's syntax, we have created a method for introducing students not only to coding, but also to computer science. This is in direct contrast with methods that emphasize the syntactic details of an entire imperative language. We believe that our method is advantageous for universities that teach an imperative language in the first undergraduate programming course. It allows applicative techniques to be taught, and it illustrates that simple computer programs can be used for solving realistic problems. As a result, we believe that this approach gives students a good preparation for further computer science courses that do not emphasize syntax, such as computation models and compiler theory.

## References


[1] Harold Abelson and Gerald Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, New York, 1985.

[2] Doug Cooper and Michael Clancey. *Oh! Pascal!, 2nd edition.* W. W. Norton & Company, New York, 1985.

[3] Kathleen Jensen and Niklaus Wirth. *Pascal User Manual And Report, 3rd edition.* Springer-Verlag, New York, 1985.

[4] Elliot Koffman. *Pascal, 3rd edition.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[5] Elliot Koffman, 1990. Personal communication.