Functional Programming: More Fundamental than BASIC?

Iain Ferguson

Project MEGSSS, Nova High School
3600 College Ave, Davie FL 33314
Tel.: (305) 370-1691 EMail: 70262.630@compuserve.com

ABSTRACT

The current secondary school computer science curriculum, which culminates in the AP exam, provides students with a misleading picture of the discipline, and many young thinkers are turned off by the rigidity and pettiness of syntax-heavy languages like BASIC and Pascal. This paper describes an introductory course that uses functional programming techniques to emphasize the analytical and problem solving aspects that make computer science so interesting, while avoiding the tedium associated with traditional approaches. A course based on this approach has proven very successful in encouraging students to further their study of the discipline.

1 Introduction

At Project MEGSSS (Mathematics Education for Gifted Secondary School Students) we have since 1985 taught a subsidiary course in computer science to our middle and high school mathematics students. The philosophy permeating our work in the classroom has been that programming is about problem solving, and is only peripherally concerned with managing complex digital machines. In the words of the eminent computer scientist Richard W Hamming of the Naval Postgraduate School, Monterey, CA, "The purpose of computing is insight, not numbers." We wanted to make students aware of the benefits of adopting a computing mode of thought without cluttering their minds with the tedious minutiae of computer control.

The computing side of things was a means to an end, not an end in itself, so we rejected the imperative-style languages such as BASIC and Pascal that pander more to the needs of the computer than to the student, opting initially for LOGO. For reasons that are explained below, we currently use Scheme, a cousin of LOGO that is now widely used at the college level. (For a discussion of the relative merits of these programming styles and languages, see Harvey [1]. A extensive list of universities that use Scheme in introductory and advanced courses is available from [2].) However, we do not introduce programming by discussing the syntactic and semantic rules of our chosen language. Instead, we press into service some intriguing *machine* puzzles based on 'black box' problems of a kind sometimes used by high school mathematics teachers and known to

computer scientists as 'data flow diagrams'. When presented with a problem requiring an algorithmic solution, the students draw **machine diagrams** (such as those depicted later in this paper) consisting of hook-ups of these black boxes. So *before* the introduction of a programming language, the students are able to experiment with and communicate ideas about algorithms.

As the algorithms they engineer become more complex, so the students begin to feel the need for a convenient *written* notation to replace their pictorial solutions. They are now ready to learn a programming language. The students are shown how their diagrams may be translated directly into Scheme, and a transition period begins in which students *think* in terms of the machine diagrams but present their solutions as programs. Of course, an algorithm presented in the form of a program offers an additional significant advantage—it can be run on a computer! So the students now have a means of testing and utilizing their algorithms. It is interesting to note that the first lesson the students spend in the computer laboratory is devoted to testing non-trivial algorithms (involving recursion, for example).

Inspired by the clarity and power of the conceptual model provided by the machine diagrams, the students progress well beyond traditional courses, coming face to face with many fascinating aspects of modern computer science. Moreover, in the author's experience, the level of enjoyment and the degree of motivation expressed by students of both sexes far exceeds that typically displayed by similar students taking more traditional courses.

2 Functional Programming

The course begins with the students being introduced to the **list** data structure. Lists are of two types:

1. the null list, written '`()`', and
2. non-null lists, which begin and end with parentheses and which contain one or more expressions (that may themselves be lists). For example, the following are all non-null lists, the last of which itself contains four lists:
   - `(Albert Betty Carol Donald)`
   - `(1 2 3 4 5)`
   - `( (a) (list of) (lists) () )`

One point that cannot easily be shown in a monochrome paper such as this is that lists (and the words, called **atoms**, that they contain) are always written in *red*, for reasons that are touched on shortly. (In this paper, all symbols that should be interpreted as being written in red are printed in the `typewriter` typeface.)

Next, the students are introduced to certain primitive black boxes, called **machines**. For example,

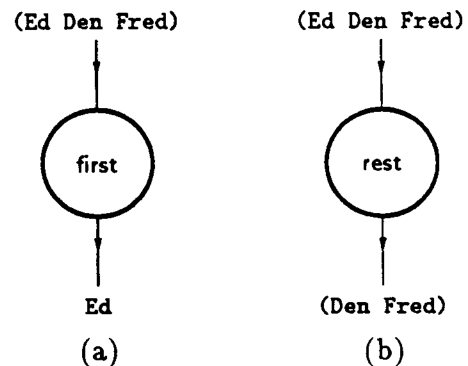

Figure 1: *The* **first** *and* **rest** *machines.*

the first machine, which incidentally is called '**first**', takes as its input any non-null list. When presented with such a list, the **first** machine outputs the first

expression in the list, as shown in Figure 1(a). On the other hand, the **rest** machine, when presented with a similar input, outputs the list obtained by *deleting* the first expression from the input (see Figure 1(b)). The following problem is then posed: Is it possible to engineer a hook-up of **first** and **rest** machines so that the output is the *second* expression of the input list? The solution is given in Figure 2.

To distinguish them from atoms, the names of machines are always written in *black*. In reality, there is little scope for confusion at this stage. However, once the Scheme language is introduced there are significant pedagogical advantages to making this color distinction. (In this paper, black symbols are represented by the **sans serif** typeface.)

Needless to say, many simple but intriguing problems can be posed using just the **first** and **rest** machines described above, and in a very short space of time the students are able to engineer solutions of surprising sophistication. However, after a while it becomes tedious having to draw a hook-up of **first** and **rest** machines each time an algorithm requires the second expression from a list to be extracted. At this point, the students are introduced to the technique of defining new, or **derived**, machines. The process of creating a derived machine is very simple: Take the hook-up concerned and draw a box around it, to create a **twin-focus diagram**. To give the derived machine a name, the new name is written next to the box. For example, we define the **second** machine by drawing the diagram in Figure 3(a). Now, instead of drawing the **first-rest** hook-up, the students may use the **second** machine. For example, the machine in Figure 3(b) outputs the *third* expression from the input list. Of course, the students may go on to create yet another derived machine, called **third** perhaps, based on the algorithm depicted in this figure, and thereafter use it in their machine diagrams as though it were primitive.

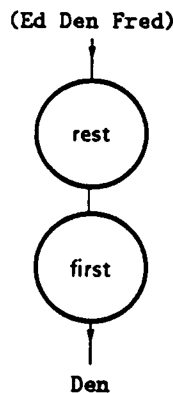

Figure 2: *A hook-up of* **first** *and* **rest**.

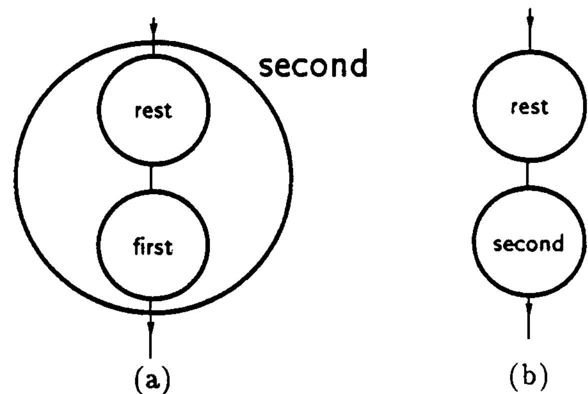

Figure 3: *The* **second** *machine.*

The students are presented with four other primitive machines in addition to **first** and **rest**. They are also provided with certain special devices which help in the construction of more complex machines. A **constant function** is a device that takes any red expression as its input, but always outputs the same expression regardless of the input. These devices are identified by a small inverted triangle; for example, the constant function in Figure 4 always outputs the

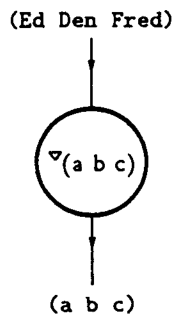

Figure 4: *A constant function*

list '(a b c)', no matter what expression is provided as its input.

In order to construct machines whose behavior changes according to the inputs provided, the students use an **if-switch**, a device that channels its input down one of two outputs, the selection being determined by an expression passed as a special 'control input' to the if-switch. When the control input is the atom '#t' (representing *true*), the if-switch directs its input to the left-hand output. When the control input is the atom '#f' (representing *false*) the input is directed to the right-hand output (two if-switches appear in Figure 5).

Armed with these tools the students are able to construct highly sophisticated machines, such as the lat? machine depicted in Figure 5. This machine outputs the atom '#t' if its input is a list that contains only atoms, otherwise (if the input list itself contains one or more lists) it outputs the atom '#f'. Note that this machine is recusive, being defined in terms of itself. The students are introduced to recursion after about fifteen hours of instruction. Thanks entirely to the machine diagrams, they find it a simple and intuitive technique. Moreover, initially the students do not distinguish between tail

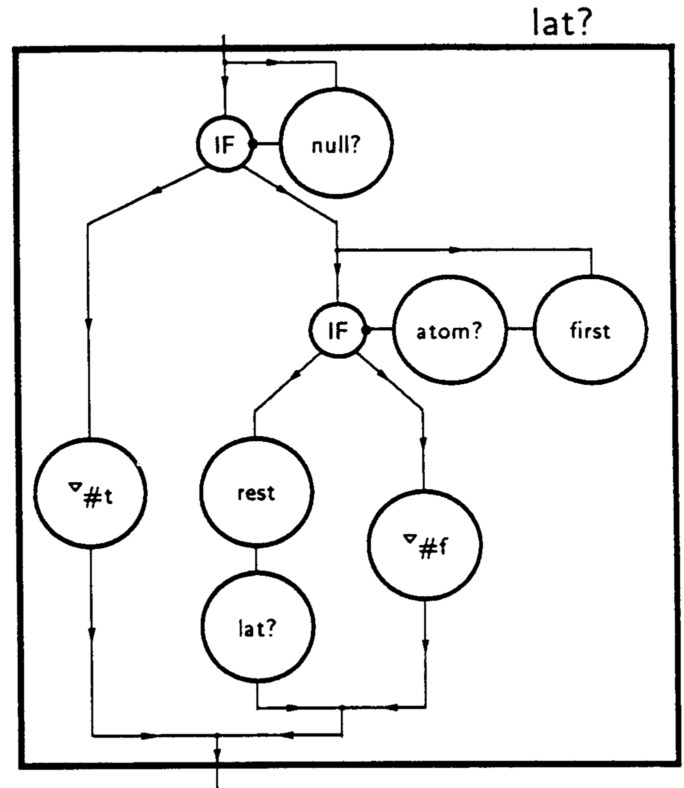

Figure 5: *The* lat? *machine.*

recursion—describing an iterative process—and full recursion, being equally at home with both and blissfully unaware that fully recursive processes are, by conventional wisdom, significantly harder to understand. Even more remarkably, the students are instinctively able to construct diagrams of machines that recur at more than one location (describing tree-recursive processes, for example). In fact, as far as the students are concerned, recursion simply means employing the machine within its own twin-focus diagram, and not until much later is it pointed out that if the machine appears at the bottom of its twin-focus diagram—as with the lat? machine in Fig-

ure 5—then the algorithm being depicted involves a simple 'looping' process rather than a true recursive process. (Shortly after the introduction of recursion, the students take a test which involves engineering several tail and fully recursive machines, and of the sixty or so students that take this test each year, about two thirds score 90% or more.)

After an initial period in which students construct only pictorial algorithms, they are introduced to Scheme notation. First, students learn to describe hook-ups; for example, the output from the hook-up of the first and rest machines shown in Figure 2 is described by the following (black) **functional expression** (note how a variable '*r*' is introduced to represent the input to the hook-up):

```
(first (rest r))
```

Next, the students are shown how to write expressions that describe the process of creating a derived machine. To construct a new machine, they first draw a box around the appropriate hook-up. This step is mirrored in the written notation by wrapping a '(lambda ...)' around the functional expression that describes the hook-up, together with a (black) list giving the variables that are being used to represent the inputs. So drawing a box around a hook-up of the first and rest machines is equivalent to writing the functional expression

```
(lambda (r) (first (rest r)))
```

Finally, this **derived function** is given a name (a process that corresponds to writing the new name beside the twin-focus diagram) using 'define':

```
(define second
  (lambda (r) (first (rest r))))
```

A similar process leads to the LOGO definition

```
TO SECOND :R
OP FIRST (BUTFIRST :R)
END
```

As mentioned earlier, initially we used LOGO as the language for describing machine diagrams, since this was the only appropriate language available on the Apple IIe's to which we had access. However, students soon reached the point beyond which LOGO cannot go. We then introduced them to the language Scheme, and were forced to forgo laboratory sessions. Now we are in the happy position of having access to IBM compatibles and Macintosh computers, and use the Scheme language (see [3]) from the beginning.

3 Conclusion

Because of the method by which a programming language is introduced, the students view a program not as a means for controlling a computer, but as a notation for describing an algorithm. Thanks to this deep and sophisticated insight into the nature of computer programming, our students are able to progress much further into the algorithmic aspects of computer science than students who follow more traditional courses, such as those leading to the College Board Advanced Placement Exam. For example, in addition to the usual fare (which is covered in a remarkably short time) students are able to explore such important concepts and techniques as

1. procedural abstraction (a technique that allows often-used algorithms to be generalized);
2. object-oriented programming (a technique used for encapsulating related procedures); and
3. artificial intelligence (a simple learning strategy is implemented as part of a tenth grade project).

All this is achieved with an absolute minimum of the syntactic distractions that take up so much time in most high school courses. In other beginning computer science courses taught by the author, in which BASIC and Pascal were the languages studied, students wasted a great deal of energy, and generated a equal amount of frustration, struggling with the seemingly petty and pointless syntactic rules imposed by the language. Yet in over six years of teaching this course, the author has yet to hear a single student express frustration as a result of syntactic issues. Instead, the students direct their unfettered creative energies toward devising elegant algorithms.

In May of 1991 we gave the Part B, Section II (programming) paper of the College Board Advanced Placement exam to our ninth grade students, without warning or preparation, and asked them to complete the test, writing their answers in Scheme. The students found the test to be easy, in some cases trivial, and most of the students had finished 15 minutes before the 45 minutes allowed. At the time of taking this test, the students had received less than forty hours of instruction in computer science, compared to the three hundred or more hours that most students who take the AP test must endure.

Unfortunately, to have received credit the students would have had to have written their answers in Pascal. However, by communicating directly with universities, we are able to ensure that our students are properly placed, and many of our former students are now studying computer science at college, either as their major or as a subsidiary discipline. Moreover, their background enables them to progress much faster than might otherwise be the case; for example, two of our former students have taken (and excelled in) a graduate level course in lambda calculus at Rice University, despite being only in their freshman year.

Acknowledgements
The diagrams in this paper are reproduced from *The Schemer's Guide* [4] with permission from the publisher.

References

[1] Brian Harvey, "Symbolic Programming vs. the A.P. Curriculum", in *The Computing Teacher*, February 1991, pp. 27–29, 56.

[2] A list of universities and colleges who teach Scheme may be obtained by sending a stamped self-addressed envelope to Project MEGSSS, Nova High School, 3600 College Ave, Davie FL 33314. Many of these, including some of the most prestigious colleges in the USA, have indicated a strong desire to attract students who have taken the course described in this paper.

[3] Project MEGSSS uses the EdScheme implementation of Scheme on IBM PC compatibles with 640K and one floppy disk drive. EdScheme is also available for Atari computers, and for the Macintosh (from September 1991). The implementer is Schemers Inc, 4250 Galt Ocean Mile, Suite 7U, Ft Lauderdale, FL 33308.

[4] Iain Ferguson, Edward Martin & Burt Kaufman, *The Schemer's Guide*, Schemers Inc. Fort Lauderdale, FL, 1990.