Programming without variables?

Michael Spivey, University of Oxford
mike@cs.ox.ac.uk

If you wanted to get someone started with programming, what would you show them first? Perhaps a one-line program with a command that prints a greeting; then a program that uses a couple of variables and assignment statements to do some simple calculations; and maybe next a program that contains a loop, so it can perform a sequence of actions that's not limited by the length of the program. Those are the basics, aren't they?

Well, not necessarily, for there's a style of programming, and programming languages to support it, where none of these things are used. In *functional programming*, there are no commands, no variables (not ones that change their value, anyway), no assignments, and no loops. By losing these features of conventional languages, a functional language gains other things that sometimes have greater value: a way to treat data without worrying how storage for it is allocated, and the ability to build programs from kits of parts that interact in productive but manageable ways.

Let's consider a very simple problem: summing the squares of the first n natural numbers. A functional program to solve this problem is written,

$$\begin{array}{l} \mathit{sum}\ (\mathit{map}\ \mathit{square}\ [1..n]) \\ \textbf{where}\ \mathit{square}\ x = x * x. \end{array}$$

The program begins with a list $[1..n]$ containing the first n numbers; applying the function *map square* to this list makes a new list in which each number has been replaced by its square; then the function *sum* takes this list of squares and adds them all up. A conventional program for the same task would have to introduce two assignable variables, one to range from 0 up to n, and another to keep a running total of the squares:

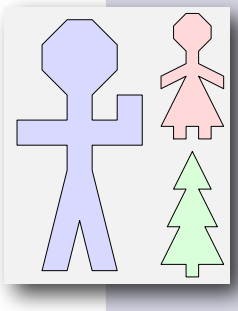
$$\begin{array}{l} k := 0;\ \mathit{sum} := 0; \\ \textbf{while}\ k < n\ \textbf{do} \\ \quad k := k{+}1;\ \mathit{sum} := \mathit{sum} + k{*}k \\ \textbf{end.} \end{array}$$

This example is very simple, but it still shows that whilst the loop in the conventional program is all of a piece and cannot be split into smaller components, the functional program contains several reusable parts. Only the function *square* is specific to this program, and the rest is put together from general-purpose components such as *sum*, which adds up any list of numbers, and *map*, which may be used to apply any function uniformly to all members of a list.

At first sight, it seems wasteful to create a list containing the n numbers, then make another list containing their squares, and finally add them up. But the answer from a functional program is the same whatever order the calculations are carried out, and a compiler is free to interleave the processes of generating the numbers, squaring them, and summing the squares, so that the program actually carries out the same actions in the same sequence as its conventional equivalent. It's this opportunity to view programs as more than an explicit sequence of actions that makes functional programming such a valuable tool for teaching students to think about programming, and also provides the inspiration behind Google's MapReduce system for carrying out immense calculations on clouds of computers working in parallel.

GeomLab

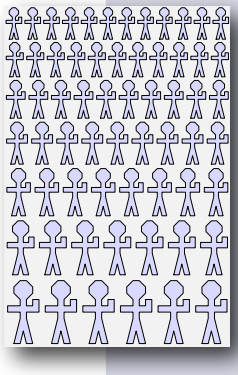
At Oxford, we think functional programming is so important that we make it the basis for the very first programming-related course taken by our undergraduates. To give students a taste of functional programming before they come to us, we've put together an online activity called GeomLab that combines functional programming with graphics. The basic functions of the GeomLab language assemble pictures from pre-defined tiles by allowing them to be placed side-by-side or one above another. The first picture shows the result of the expression



$$\mathit{man}\ \$\ (\mathit{woman}\ \&\ \mathit{tree})$$

where a stick-figure man is placed beside ($) a picture in which a woman appears above (&) a tree.

Participants are guided through a sequence of exercises where they begin to describe more complex pictures using recursion. They create the second picture from a varying series of rows, each made up of several copies of the *man* picture, describing it with a pair of nested recursive functions. Before long, they are investigating how more striking pictures, like the Escher image shown below, can be assembled from a handful of basic tiles.



We have used GeomLab successfully to run whole-class extension activities for a wide range of ages from Year 10 up. An hour is enough to get an idea of what is possible, but the website provides sufficient material for an exploration that lasts one or two days, and others have used the materials as the basis of a computer club meeting weekly over the course of a term.

http://www.cs.ox.ac.uk/geomlab