DRAWING AND UNDERSTANDING

RECURSIVE FUNCTIONS

Gregory D. Weber
Indiana University East
2325 Chester Boulevard
Richmond, IN 47374
gdweber@iue.edu

ABSTRACT
Sifflet is a visual, functional programming language and environment, developed as an aid for understanding recursive functions. Sifflet’s explanations of recursive computations provide visual reinforcement of the copies model of recursion. They are driven by student demand, supporting active learning and avoiding information overload.

1 THE CHALLENGE OF RECURSION
Recursive functions are not easy for beginning programmers. Kahney [3] distinguished between the loop and copies models of recursion, and presented evidence that most novice programmers hold the loop model, while experts possess the copies model.1 In the loop model, a recursive call causes control to return to the beginning of the procedure; new parameter values simply replace the old, without saving the old values on a stack and restoring them later as the recursion unwinds. Students with the loop model can correctly interpret tail-recursive procedures, such as factTR (Figure 1 (a)). In a tail-recursive procedure, the recursive call is literally the last thing that happens, so that it is semantically equivalent to a loop; an optimizing compiler can in fact convert it to a loop. However, students holding the loop model cannot correctly interpret functions with embedded recursion, where there is more work to do after the recursive call returns, as in factE (Figure 1 (b)); the copies model is required for understanding such functions.

1 About a third of the students seemed not to have grasped either model firmly.

© 2013 by the Consortium for Computing Sciences in Colleges. Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the CCSC copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Consortium for Computing Sciences in Colleges. To copy otherwise, or to republish, requires a fee and/or specific permission.
Kurland and Pea [4] reported similar misconceptions among children aged 11-12 years in their second year of Logo programming.

Diagrams, in the style of The Schemer's Guide [2], can help explain how recursive functions work, but they require an instructor or textbook to provide the diagram; students with a weak grasp of recursion cannot reliably make their own diagrams.

Some software tools show recursive computations step by step using tracing and animation. With the tracer in WinHIPE [6], the user can request a single step, N steps, or stepping to a breakpoint. WinHIPE can also produce an animation to be “played” with a VCR-like control. Similarly, SRec [9] produces animations of recursive Java methods. Many interactive debuggers offer similar, if less graphically rich, facilities. These tools have a few drawbacks. Anything with a “play” button makes it easy to become a passive viewer rather than an active learner. Since computations have many steps, there is a danger of information overload. Finally, the tools make an arbitrary choice of “next” step which is foreign to the nature of pure functions.²

This paper presents an alternative, designed for active learning, with information presented on demand: the visual, functional programming language Sifflet.³ Developed with the primary goal of helping students understand how recursion works, Sifflet

---

²The term pure function is explained in the next section.

³Simple vIsual Friendly Functional Language of Expression Trees. Sifflet is also the name of a brilliant, high-pitched organ stop and is the French word for a whistle.
provides graphical support for the copies model. Sifflet functions are drawn as expression trees and evaluated as data-flow diagrams. Roughly, Sifflet is a pure functional subset of Lisp with expressions represented as diagrams and no parentheses. The remaining sections cover basic features of Sifflet (Section 2), using lists in Sifflet (Section 3), future work (Section 4), and how to get involved with Sifflet (Section 5).

2 SIFFLET BASICS

Sifflet is a functional language: it’s about programming with pure functions. A pure function is a procedure that, given the same arguments, always returns the same value, like a function in mathematics. Functional programming contrasts with the imperative style of programming, which is rife with side effects, such as mutation of variables: pure functional programs may not contain assignment statements such as $x = x + 1$. Side effects are also common in object-oriented programming. While recursion is important in imperative and object-oriented programming, it is especially important in the functional style, where there are no loops. It also occurs there in its simplest form, since there are no side effects.

Sifflet is a visual language based on dataflow diagrams. The diagrams show how inputs (arguments) pass into processors (functions, operators), which produce outputs (values), which become the inputs of further processors. To establish connections with text-based languages, Sifflet can export function definitions to some of them (currently, to Python, Scheme, and Haskell).

To define a Sifflet function, the programmer selects “New function” from the menu, enters the function and parameter names, and builds an expression

![Figure 2: A call to the factorial function.](image)

The if node evaluates its first argument (leftmost subtree); if the result is True, it evaluates and returns the second argument (middle subtree); otherwise, the third argument (rightmost subtree). The zero? function tells whether its argument is equal to zero; sub1 subtracts one from its argument; * multiplies its two arguments. The output value of a node (except for the literal nodes 1) is shown in italics above the node label; for example, 120 is the output of the * node.
tree for the function body. Nodes in the tree represent functions, literals, parameters, and the conditional processor “if.” There are tools to insert, connect, disconnect, move, and delete nodes.

After defining a function, the programmer can drop a copy of it onto the Sifflet Workspace window, click to call it, and enter the actual argument values for the call. The workspace then displays the result of the function call and intermediate results as in Figure 2.

But how is the value (24) of the recursive call computed? Clicking on the node pops up another copy of the function for the recursive call. Sifflet's display of visual copies of the recursive function reinforces the copies model of recursion. In the new copy, again, how is the recursive call's output determined? Clicking on this node, too, reveals the process— and so on until reaching the base case. But if the student catches on to what is happening before getting down to the base case, it is not necessary to go that far — the expansion of recursive calls is shown one level at a time, on demand (Figure 3). Since a mouse click is required to display each copy, the student can browse through as many or as few levels of recursion as needed, avoiding information overload.

In the case of tree recursion, as in the slow Fibonacci function fib1 (Figure 4), the student has a choice of clicking to expand either the left or the right recursive call
(Figures 5, 6) or both (not shown). There is no need to go through the left recursive call before the right, or vice-versa. With pure functions, the order of evaluation is irrelevant: there are no side effects, so either way produces the same results; and with a parallel machine, both might be evaluated at the same time.

There are two key differences between Sifflet's “explanation” of the value of $\text{fib} 1 5$ and a conventional trace or animation of an algorithm:

![Diagram](image)

**Figure 4:** A call to the slow Fibonacci function $\text{fib} 1$. The nodes labeled `==`, `+`, and `-` represent the usual binary operators for equality, addition, and subtraction.
CCSC: Midwestern Conference

Figure 5: Expanding the left recursive call to fib1

1. Sifflet's explanation is driven by student questions, promoting a more active form of learning and avoiding the risk of bombarding the student with inessential details. It does not explain how a particular function call computes its value unless the student asks. Consequently, the student can decide which paths of computation to explore. If the function call makes two or three other function calls, a tracer will show the details of all. But the student may be interested in only one of them. Also, the student can decide how far to go down each path of computation. Some students may need to venture all the way down to the base case; others may catch on sooner.

2. Sifflet's explanation is pictorial. Within a particular call frame, the entire computation is laid out. All details within the call frame are shown; students can focus on the data paths that are most interesting.

3 LISTS IN SIFFLET

The list data type is another interesting domain for the study of recursive functions. The data structure itself is recursively defined:

A list is either empty, or it is composed of a head (the first element) and a tail (a list containing the rest of the elements).
In Sifflet, lists are constructed with : (“cons”), which builds a new list out of a head element and a tail list.\(^4\) The predicate \texttt{null} is true of the empty list, \([\,\,]\); the functions \texttt{head} and \texttt{tail} extract the parts of a non-empty list.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure6}
\caption{Expanding the right recursive call to \texttt{fib1}}
\end{figure}

The recursive structure of many functions on lists naturally follows from the recursive definition of the list data type: the base case occurs when the list is empty; the recursive step involves applying the function to the tail of the list and usually combining the result of that application with the value of the head.\(^5\) For example, \texttt{sum} (Figures 7 and 8) computes the total of a list of numbers.

\[^{4}\text{The : notation is from Haskell.}\]

\[^{5}\text{Applying a function to } x \text{ means calling it with argument } x.\]
4 FUTURE WORK

Further work for Sifflet should include formal studies of its effect on student learning, additional language features, system and user libraries of functions (packages, modules), and "toys."

While informal observations suggest that Sifflet has helped students understand recursion, objective measurements of student learning are needed.

Language enhancements include type inference, higher order functions, let expressions for local variables, and additional data types. Type inference will enable Sifflet to export programs to languages that, like Java, C, and C++, are statically typed, but lack their own type inference mechanisms. Type inference, like type checking with type declarations, can also detect or prevent many programming errors before running the program.\(^6\) Higher order functions are important tools, along with recursion, in functional programming. An example is the map function which takes a function parameter \(f\) and a list \(xs\) and constructs a new list by collecting the results of applying \(f\) to each element of \(xs\). For example, map sub1 \([2, 4, 6]\) has the value \([1, 3, 5]\). Local variables, of course, avoid the need for duplicating computations where the value is used in more than one place. Trees would be an interesting additional data type for students to learn about recursion. Allowing users to define their own data types is also desirable.

Toys, such as 2D and 3D graphical objects and musical output, contribute enormously to the appeal of the visual languages Alice and Scratch \([1, 7]\), as well as the non-visual Logo. Sifflet needs toys! But there is a significant gap between the world of pure functions and the world of toys: while repeated calls to a function \(f(x)\) always have the same value as long as \(x\) is the same, repeated commands to toys have different effects. For example, the repeated command move 10 to a robot places it first 10 steps away from

---

\(^6\)Type inference is actually already implemented in Sifflet, but not in a good way, because the type error messages are confusing to students.
where it started, then 20 steps, 30, and so on. Nevertheless, there are ways to connect functional programs to toys. The function could generate a list-like stream of commands which are sent one by one to the toy. If reaction to sensor data is needed, it could be provided by a function which takes sensor data and previous state as arguments, producing a pair of values: the new state and the current command to the toy. A “universal controller” similar to the one Papert and Solomon described for Logo [5] would be useful.

5 GETTING INVOLVED

Sifflet provides diagrams that explain the computation of functions, with an emphasis on explaining recursive functions, providing visual reinforcement for the copies model of recursion. It enables students to draw rather than write function definitions. Both aspects can be appealing and helpful for visually oriented learners.

The Sifflet website (http://mypage.iu.edu/~gdweber/software/sifflet/) provides instructions for installing and using Sifflet, including an extensive tutorial, and links to downloadable source code.

Collaborators are welcome in various roles, including:

- Studying the effects on student learning.
- Contributing binary packages and installers for various operating systems.
- Further development of Sifflet (in Haskell).

REFERENCES


