Literate Programming in the Large

The 30 Year Horizon

axiom

Jocelyn Guidry
(1) \( u := \text{operator} 'u' \) ; \( f := \text{operator} 'f' \)

(2) \( \text{heat} := -2xt \cdot \text{d}(f(x/\sqrt{t}))/\sqrt{t}, x, 2) - x \cdot x \cdot \text{d}(f(x/\sqrt{t}))/\sqrt{t}, x) - f(x/\sqrt{t}))/\sqrt{t})/(2xt) \)

(3) \( s := \text{rule}(u(x, t) = f(x/\sqrt{t}))/\sqrt{t}) \)

(4) \( t1 := s(\text{heat}) \)

(5) \( t2 := \text{subst}(t1, x = z \cdot \sqrt{t}) \)

(6) \( t3 := t2 \cdot \text{denom}(t2) = 0 \)

(9) \( \text{solve}(t3, f, z) \)

(9) \([\text{particular} = 0, \text{basis} = [\{e/4, e/4 + e^2, e^2 d\}]\]
Axiom

* coded like it was 1970 (it was)
* small files, 4 letter comments
* tree of source code
* multiple authors over many years
* man page documentation style
Axiom

* a 1.2 million line Common Lisp program
* cost over 42 million dollars to develop
* was a competitor to Mathematica
* was written by leading researchers
* was written at IBM Research on AKCL
Axiom

* Scratchpad project at IBM Research
* Sold to Numerical Algorithms Group (NAG)
* Bill Schelter's Death
* Withdrawn from Market (2000)
* Release as Open Source software
Axiom

* What does this code do?
* Removing it breaks the program.
* I wrote it. I know WHAT it does.
* I don't understand WHY I wrote it.
* I failed to communicate to myself.
Why
Axiom

* How do you make code live?
* Is the code worth keeping?
* Does your company depend on it?
Axiom

* How do I make Axiom live?
* Development teams move on.
* Knowledge moves with them.
* Human communication is key.
* Humans communicate with stories.
Literate Programming

Donald E. Knuth
I believe that the time is ripe for significantly better documentation of programs, and that we can best achieve this by considering programs to be works of literature. Hence, my title “Literate Programming”

Let us change our traditional attitude to the construction of programs. Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do.

such that, for each vertex $v \in V$, the neighbours $x_1, x_2, \ldots, x_i$ smaller than $v$ appear in $A_u(v)$ in this order as indicated by a dotted arrow in Fig. 3.9. That is, the $v$’s neighbour embedded around $v$ counterclockwise next to the top entry $x_i$ of $A_u(v)$ is greater than $v$. In particular, the top entry of list $A_u(t)$ for sink $t$ is the source $s(=1)$. Now we present the algorithm “ENTIRE-EMBED” for extending such an upward embedding $A_u$ into an embedding Adj of a given graph. The algorithm executes once the depth-first search starting at sink $t$ on a digraph $D_u$. The algorithm adds vertex $y_k$ to the top of list $A_u(v)$ when the directed edge $\langle y_k, v \rangle$ is searched.

```plaintext
procedure ENTIRE-EMBED;
begin
  copy the upward embedding $A_u$ to the lists Adj;
  mark every vertex "new";
  $T := \emptyset$; \{A DFS-tree $T$ is constructed only for analysis of the algorithm\}
  DFS($t$)
end;
```
6.1.4 Classifying Variables

Those preceding definitions handle only the case of local variables, that is, only variables in lambda forms. There are, of course, global variables, and among them, predefined variables and/or immutable ones, like cons or car. In our current state, the only way of accessing them would be to follow the links between activation records, but that technique makes access to global variables particularly slow since they are located in the ultimate activation record. For that reason, we'll treat these statically classified variables differently.

We'll assume that the global variable g.current contains the list of mutable global variables, while g.init contains the list of predefined, immutable ones, such as cons, car, etc. The function compute-kind classifies variables and returns a descriptor characterizing variables.

\[
\begin{align*}
\text{(define (compute-kind r n)} & \\
\text{\quad (or (local-variable? r 0 n))} & \\
\text{\quad (global-variable? g.current n)} & \\
\text{\quad (global-variable? g.init n))} & \\
\text{(define (global-variable? g n)} & \\
\text{\quad (let ((var (assq n g)))} & \\
\text{\quad \quad (and (pair? var) (cdr var)))}) & \\
\end{align*}
\]

We test g.current before g.init so that we can mask predefined variables, if need be. Considering primitives as values of immutable global variables is a safe practice. However, certain implementations of Scheme allow a program to redefine such a global variable (car for example) on condition that the redefinition modifies only this variable and not any other predefined function (not even those, like map, for example, that seem to use car). Only the functions explicitly in the program will see the new value of car. You can get this effect simply by compute-kind, but there is still a problem of knowing how to insert such a variable in the mutable environment. We could also invent a new special form, redefine, say, for this purpose. [see Ex. 6.5]
Implementing Elliptic Curve Cryptography

- Working code in C
- Understandable Math
- Mathematicians’ Secrets Translated
- IEEE P1363 standard
The next set of parameters is:

```c
#define NUMBITS    158
#define NUMWORD    (NUMBITS/WORDSIZEx)
#define UPRSHIFT   (NUMBITS%WORDSIZEx)
#define MAXLONG     (NUMWORD+1)
```

NUMBITS is used more in the following chapters; it is the total number of bits we expect to be working with. The exact value can be anything you like; the value shown above just happens to be what I was working on when I wrote this. The more advanced large integer packages do not fix this length. For modular math and for the mathematics described in chapters 3 and 4, a fixed number of bits will work fine.

NUMWORD is the maximum index into an array of machine words we need to represent NUMBITS. MAXLONG is useful in for loops; it is the number of machine words used in an array that holds NUMBITS. The parameter UPRSHIFT is not used by the large integer package; I'll describe that in the following chapter.

Specific parameters for the large integer package deal with the hardship of not having access to the carry bit. This implementation uses unsigned machine words and divides them in half to fake a carry. For example, a 32-bit machine will use 16 bits of an unsigned long in each array entry to store the bits; then, when two 32-bit machine words are added together, the 17th bit can be used to see if a carry occurred. This is really crude, but it should work on just about any processor that has a C compiler.
Axiom

* 1 million lines of Literate latex
* 12 years of rewriting so far
* built directly from the documents
* 250,000 lines of code still to be merged
* 1.3 million lines of tests to document
Combined Table of Contents

- Volume 0: Axiom Jenks and Sutor -- The main textbook
- Volume 1: Axiom Tutorial -- A simple introduction
- Volume 2: Axiom Users Guide -- Detailed examples of domain use (incomplete)
- Volume 3: Axiom Programmers Guide -- Guided examples of program writing (incomplete)
- Volume 4: Axiom Developers Guide -- Short essays on developer-specific topics (incomplete)
- Volume 5: Axiom Interpreter -- Source code for Axiom interpreter (incomplete)
- Volume 6: Axiom Command -- Source code for system commands and scripts (incomplete)
- Volume 7: Axiom Hyperdoc -- Source code and explanation of X11 Hyperdoc help browser
  - Volume 7.1 Axiom Hyperdoc Pages -- Source code for Hyperdoc pages
- Volume 8: Axiom Graphics -- Source code for X11 Graphics subsystem
  - Volume 8.1 Axiom Gallery -- A Gallery of Axiom images
- Volume 9: Axiom Compiler -- Source code for Spad compiler (incomplete)
- Volume 10: Axiom Algebra Implementation -- Essays on implementation issues (incomplete)
  - Volume 10.1: Axiom Algebra Theory -- Essays containing background theory
  - Volume 10.2: Axiom Algebra Categories -- Source code for Axiom categories
  - Volume 10.3: Axiom Algebra Domains -- Source code for Axiom domains
  - Volume 10.4: Axiom Algebra Packages -- Source code for Axiom packages
  - Volume 10.5: Axiom Algebra Numerics -- Source code for Axiom numerics
- Volume 11: Axiom Browser -- Source pages for Axiom Firefox browser front end
- Volume 12: Axiom Crystal -- Source code for Axiom Crystal front end (incomplete)
- Bibliography: Axiom Bibliography -- Literature references
Quotes
Step away from the machine. Literate programming has nothing to do with tools or style. It has very little to do with programming. One of the hard transitions to literate programming is “literate thinking”.
The effect of this simple shift of emphasis can be so profound as to change one's whole approach to programming. Under the literate programming paradigm, the central activity of programming becomes that of conveying meaning to other intelligent beings rather than merely convincing the computer to behave in a particular way. It is the difference between performing and exposing a magic trick.

--Ross Williams, FunnelWeb Tutorial Manual, pg 4
Imagine giving a new team member a copy of your literate program. Send them to Hawaii for 2 weeks, on an all expenses paid vacation. If, when they return, they can modify and maintain the program as well as the original authors then you have written a truly literate program.

Strive to pass the Hawaii test.
Motivation is the key difference between documentation and literate programming.
Literate Programming is not Documentation
The hardest part of literate programming is the documentation
Lessons Learned
Avoiding Fructer
Integrating Documentation and Development
From an IBM 1970s Study:
Hour for hour, the most effective way to get bugs out of code is to sit and read the code, not to run it and not to write unit tests. Code review is the best technique known for fixing bugs.

Get somebody else to read your code. 60 to 90 percent of the errors can be removed before the very first run of the program.

And hour for hour, if you have a choice of writing unit tests and reading code, read the code.

Editor-in-chief on each team
Editor in code review
* require section text before checkin
Changes to development
* larger files (7000+ pages)
* rearranged code organization
* automatic XREF – who calls X, who X calls
* TOC / index / bibliography
* no need for an IDE
Reproducible Research
An article about computational science in a scientific publication is not the scholarship itself, it is merely advertising of the scholarship. The actual scholarship is the complete software development environment and the complete set of instructions which generate the figures.

– D. Donoho on Reproducible Research Standards
Tangle
\section{Tell the story}
This is where you write the story about why you wrote the code.
\begin{chunk}{a chunk of code}
    public static void main(String[] args) {
        ......
    }
\end{chunk}
Now you understand the code above.
Interior nodes of Clojure's Red Black trees can have a value. A BlackBranch node is a leaf node with a null value and children, one of which could possibly be null. This is constructed by PersistentTreeMap's `refToBlack` method.

```java
\defsubclass{PersistentTreeMap}{BlackBranch}
\extends{BlackBranch}{Black}
\begin{chunk}{PersistentTreeMap BlackBranch Class}
static class BlackBranch extends Black{
    final Node left;
    final Node right;

    public BlackBranch(Object key, Node left, Node right){
        super(key);
        this.left = left;
        this.right = right;
    }

    public Node left(){
        return left;
    }

    public Node right(){
        return right;
    }

    Node redden(){
        return new RedBranch(key, left, right);
    }
}
```

```java
static class BlackBranch extends Black{
    final Node left;
    final Node right;

    public BlackBranch(Object key, Node left, Node right){
        super(key);
        this.left = left;
        this.right = right;
    }

    public Node left(){
        return left;
    }

    public Node right(){
        return right;
    }

    Node redden(){
        return new RedBranch(key, left, right);
    }
```
Clojure Language Demo

* generates interpreter
* generates new PDF
1.0 Literate Programming - A Gentle Introduction
1.1 What is the problem?
1.2 Is there a better way?
1.3 Literate Programming -- The basic idea
1.3.1 Literate Programming is NOT documentation
1.3.2 Literate Programming is a change in mindset
1.3.3 Literate Programming reduces bugs
1.4 Understanding the literate programming tangle tool
1.4.1 Motivation, or getchunk
1.4.2 All of the literate programming tools
2.0 The Tangle Program
2.1 Understanding the story of the program
2.2 The main task
2.3 A complication with HTML
2.4 Finding the chunk
2.5 Finding the next line in the buffer
2.6 At last, finding the chunk
2.7 Printing the chunk we found
2.8 The simple case, print the line
2.9 Checking chunks for getchunk tags
2.10 Getting a new chunk name from the getchunk tag
2.11 Finding the end of the chunk we are printing
2.12 Final cleanup and housekeeping. The preamble
3.0 The tangle source code
4.0 What have we learned?
5.0 Credits
Summary

Make code live

Avoid Fructer

Make documentation primary

Write literate programs
Tim Daly

daly@axiom-developer.org
http://daly.axiom-developer.org
axiom-developer.org/axiom-website/litprog.html
The 30 Year Horizon

axiom