Documentation for TexiWebJr

SEPTEMBER 2018

LOLH

Published by:

LOLH
Address Line 1
Address Line 2
etc.
Email: user@example.com
URL: http://www.example.com/

This is Edition 0.1 of *Documentation for TexiWebJr*, covering version 0.0.2 of the `twjrdocs.twjr` program.

Table of Contents

1 Introduction and Apology

`TexiWeb Jr` is "a system that provides literate programming on top of the `Texinfo` markup language." It's source documentation can be found at `http://github.com/arnoldrobbins/texiwebjr`. Check it out; it has a wealth of information about literate programming in general and this system in particular.

It is not my intent to supersede or duplicate or attempt to improve anything written by the author of the `TexiWeb Jr` system in this document; rather, this document and its various forms of output are all designed primarily to help me be a better literate programmer and student. I suppose first, then, that I need to offer some justification as to why I am doing this, if only for myself. Here, then, is my little apology.

I believe in the value of the literate programming paradigm, and I believe in `TexiWeb Jr` to that end because it allows me to produce quickly and easily various forms of output, such as an `Info` file with fully outlined, hypertexted, and (most importantly) indexed concepts, a TeX PDF output, which is fully outlined, hypertexed, and (again) indexed, produce runnable, hypertexed, indexed, and documented code using any language or combination of languages, and HTML output with all of these attributes that can be quickly and easily placed into the Cloud. All of these combined mean that I can be very productive very quickly and retain the benefits of what I have learned and produced far into the future, which is very much more than plain vanilla programming allows me to do. Because everything is both hypertexted and indexed (by me the author), I can literally find and jump to any topic in a matter of seconds in any medium (Info file, HTML web page, PDF document). Literate programming is probably first and foremost a way for me to learn, refresh and retain my learning into the future. I can decompose a subject into an `Info` file in a matter of minutes, and then return any time to review and update that `Info` file whenever I come across a similar subject or have a new thought. There is no other system that I am aware of that allows me to learn and review as quickly and concretely as the combined resources of TeX, `Info`, HTML, PDF, and `TexiWeb Jr`.

Therefore, this document presents the salient features of the `TexiWeb Jr` system (i.e., those things that are unique to it) while leaving the `Texinfo` system to fend for itself. Certainly, learning the `Texinfo` system is by far the more important thing to do (and certainly is no easy task) but there are some things that are different or new in `TexiWeb Jr` than in `Texinfo` that need to be addressed in order to be productive.

There is without a doubt a steep learning curve to being a productive literate programmer; there are numerous systems that must be learned, and learned well, such as `Emacs`, `Texinfo`, `Info`, and TeX, to name the most obvious. There is also the mental training and attitude that is required to transform one's methodology from whatever it was to that of literate programming. And there is of course the need to be able to write. I am old enough to have gone through several of these paradigm shifts in my lifetime (moving from writing papers on note paper to typewriters to computers using word processors; and moving from reading bound books to computer screens to tablet screens to book readers for another). If there is one thing having lived through these transitions has taught me, it is the value of striving to keep my mind plastic (*neuroplasticity*) and receptive, and not getting stuck in a paradigm simply because it has become comfortable. Shifting paradigms and learning new systems is hard; but there are enormous advantages on the other side.

1.1 System and Knowledge Requirements

In order to process a `TexiWeb Jr` source file, and use its `Info` and PDF output, you will need just a couple of requirements:

- Emacs (to be truly productive using `Texinfo` and `Info`)
- GNU AWK (`gawk` version 4.0 or later) because that is the program that powers both `jrtangle` and `jrweave`
- `Texinfo` of at least version 6.1 (the `TexiWEB Jr` source code contains a version that will work)
- knowledge of how to navigate through an `Info` file (the `Info` program has a great interactive tutorial that teaches everything in a short amount of time)
- a $\TeX$ engine(see Download a $\TeX$ System)

1.2 Literate Programming

For some general introductions and information about the Literate Programming world, see, for example:

- Wikipedia's Wiki on Literate Programming
- LiterateProgramming dot com
- Normal Ramsey's Literate Programming Site
- Ben Pfaff's `TexiWEB` system for `C` language literate programming;
    - see `http://adtinfo.org` for the HTML output;
    - for a PDF of the textbook, `https://ftp.gnu.org/gnu/avl/avl-2.0.2.pdf.gz`
    - see `https://github.com/CoryXie/libavl` for source code on GitHub;
    - see `http://savannah.gnu.org/projects/avl` for source code on Savannah.
    - see `http://adtinfo.org/poster/index.html` for a diamond exposition of both what literate programming is and where it can be most effective, i.e., in retaining knowledge

1.3 Ben Pfaff's `TexiWEB` System

Literate programming has never caught on in the mainstream. This is probably because programmers don't want to write essays every day, or at least most don't. After all, there is a reason they became programmers, not journalists. But there are proponents of the paradigm who are professional programmers. One is Donald Knuth. Another is Ben Pfaff. He turned to literate programming on one occasion and has given his rationale for this decision in an essay he wrote, part of which I have reproduced below. I reproduce this because it is really both what got me into literate programming, but also because it exactly mirrors my own need for literate programming.

Does literate programming make one a better programmer? I don't know. I believe that if everyone produced literate programs, programming would be easier, because so much time is spent trying to understand someone else's code and literate programming would make that chore more manageable, albeit with additional time perhaps spent developing the essays to go with the programs. As Donald Knuth said way back when he coined the phrase, he knew it was a loaded term, because who wants to be accussed of being a proponent of Illiterate

Programming? Donald Knuth said he was a better programmer because of it, and others have said the same. For me, it is as much about learning a system that helps me retain what I have studied and torn apart as it is about being a better programmer. In the end, if I do not know how to program well, being literate about it is really meaningless. So, literate programming is a system that I believe will help me become a better programmer through a great deal of introspection and self-reflection through explaining myself in writing (*meta-programming?*), not make me a better programmer in and of itself. That is good enough for me.

Because Ben Pfaff's exposition on literate programming influenced me so much to pick up literate programming, I take the privilege of placing several portions of his writings on the subject here. Pay particular attention to the bolded text in the second section. In the end he needed a literate programming system to help him be a better programmer. These are both from `http://adtinfo.org/poster/index.html`.

What is a Literate Program? (by Ben Pfaff)

A *literate program* is intended to be useful, as is all software. But a literate program is also intended to be a work of technical literature that explains the program's inner workings. A literate program is at the same time a functional design and an essay that describes that design.

The ideas behind literate programming date back to at least the 1960s, but credit for its popularization is due to eminent computer scientist Donald Knuth, who coined the name itself in his 1984 article "Literate Programming."

The benefits of literate programming are numerous. Literate programs tend to have good organization and design because of the need to justify decisions to readers. They can be easier to maintain because reasons for design decisions are clear. They can be easier to write because of the enhanced ability, available with most literate programming systems, to write code in the order most convenient for the programmer, instead of that most suitable for the compiler.

Literate programming does have some drawbacks. The most apparent is that it requires additional skills on the part of the programmer. Not every programmer is also a talented essayist, but some writing ability is necessary to be an effective literate programmer.

The most well-known examples of literate programs are $\TeX$ and METAFONT by Knuth himself. Taken as literature, both of these programs are published in book form as well as available electronically. Taken as software, they are commonly used for publishing articles and textbooks, especially those containing mathematics.

Libavl 2.0: A Literate Balanced Tree Library (By Ben Pfaff)

The original version of `libavl` worked well, as evidenced by its use in several programs.

But over time it turned out that the knowledge that it embodied was as important as the actual code. That is, the code itself is a realization of one way to manipulate a few kinds of balanced trees, but it gives few hints of the reasons for trade-offs made during its design or how to construct similar libraries given different priorities. Reasons to build new, similar libraries kept cropping up, but each time much of the rationale behind libavl had to be rediscovered, even by its original author. The needed information was scattered among books, papers, and online sources.

Eventually, a solution presented itself: rewrite `libavl` as a literate program! Upon investigation, this idea came to look better and better. Early in 2000, the rewrite began. . . .

The first step in writing `libavl 2.0` was to decide on a literate programming system. Several of these exist. Eventually, it was determined that none of them exactly met `libavl`'s needs. As a result, `libavl` uses its own custom literate programming system, called `TexiWEB`. This name is taken by combining 'Texinfo', the publishing system that TexiWEB is based on, with 'WEB', Knuth's own literate programming system.

1.4 The Problem with Literate Programming

Literate programming is difficult; it requires learning a lot of new tools, being able to decompose a problem, perhaps in a number of different ways, and then being able to express one's reasons for one's decisions in prose. That all takes time and energy. One would also need to do the same with the tests, and writing tests themselves is a tooth-pulling experience to begin with. My only response to these difficulties is that I believe the end result to be superior, and so the extra effort should be worth it. If the extra effort does not produce superior results, then it would truly be a waste of time.

> Without wanting to be elitist, the thing that will prevent literate programming from becoming a mainstream method is that it requires thought and discipline. The mainstream is established by people who want fast results while using roughly the same methods that everyone else seems to be using, and literate programming is never going to have that kind of appeal. This doesn't take away from its usefulness as an approach.
>
> —*Patrick TJ McPhee*

2 Next Chapter Title

Appendix A First Appendix Title

Appendix B Code Chunk Summaries

This appendix presents alphabetical lists of all the file definitions, the code chunk definitions, and the code chunk references.

B.1 Source File Definitions

B.2 Code Chunk Definitions

B.3 Code Chunk References

Index