Foreword

This booklet is based on a challenge Will E. Byrd set himself in January 2024: to overcome procrastination by committing to writing and publishing 11 books and 1000 YouTube videos during the calendar year.

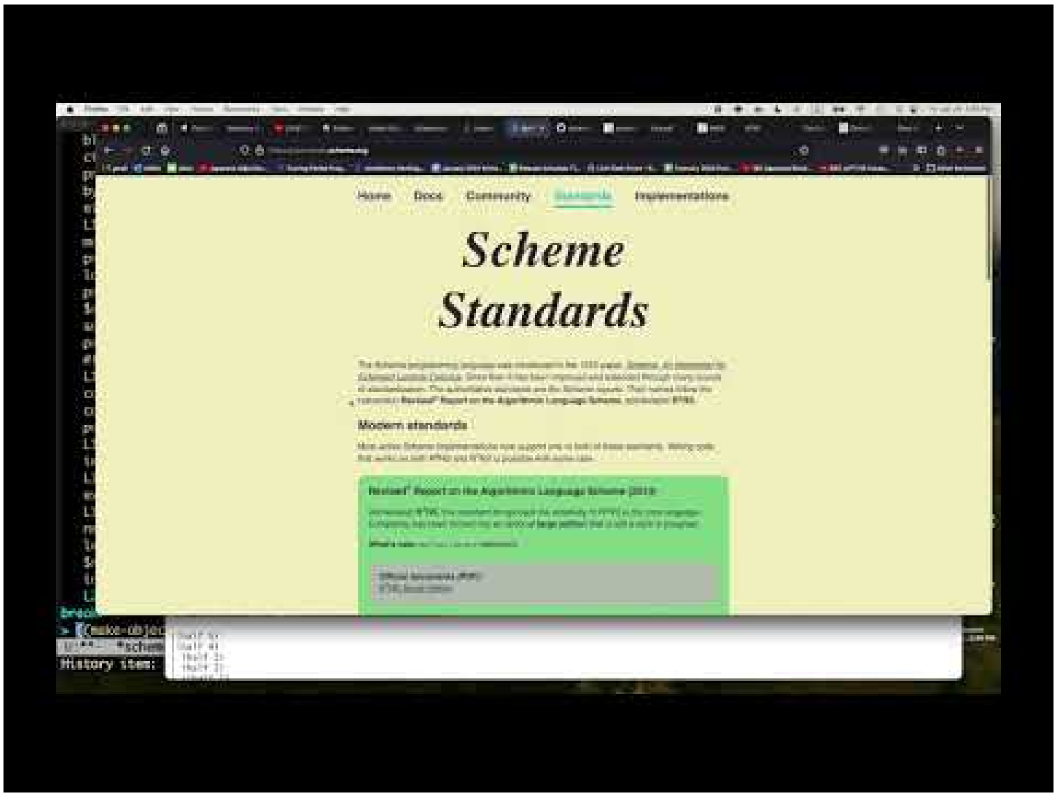
Shownotes

January 2024

January 26th - Scheme Learning Resources

In which Will Byrd shows us some starting points for diving into his favourite programming language

January 26th 2024: Scheme Learning Resources



Watch Will's video at https://youtu.be/iC8eSdoyu9A

Words by Will E. Byrd

The following is a (very) lightly edited version of the transcript of Will's video linked above. Links to content mentioned have been located and included for further reading/viewing.

Intro

Welcome to Episode IV of **Will's Guide**. This year, I've decided to make 1,024 videos - I've upped the number from a thousand, because I think two to the tenth or one kiloTube of videos is more fun!

I've also set up the Discord server, and we've had about a dozen people join. If you're interested in joining the Discord server *The Imperishable Wonderland of Infinite Fun*, please email me.

Today, I'd like to talk about something close to my heart: Scheme, a language which I love. And I'm just going to talk a little bit about how you might learn Scheme. I'm going to have a whole bunch of videos on Scheme (I hope), but how do you learn about Scheme? What are the resources available? Let's just talk about a few of them I found useful..

The Little Schemer by Dan Friedman and Matthias Felleisen

- The Little Schemer (MIT Press)
- Dan's Wikipedia Entry
- Felleisen.org Site, Matthias' Wikipedia and Racket.org

One is called *The Little Schemer* by Daniel P Friedman and Matthias Felleisen. Dan was my adviser at Indiana University for my PhD, and he was also Matthias's adviser. You also might know Matthias from the Racket world, so you have two experts guiding you.

This book is really about thinking computationally, and specifically thinking about *recursion* - recursion over lists and trees - those sorts of things, and also a little interpreter work. So, if you think is is going to be a book that's a big thick manual on Scheme - the programming language and the semantics or whatever - well, you might be disappointed. But this really in another way is at the heart of thinking about Scheme computationally and thinking about computation and recursion, so it's a great book to affect your thinking and make sure that you're really comfortable with ideas about recursion.

So this is recommended by me if you want to learn how to think recursively at least a certain way of thinking recursively. If you're not comfortable with recursion, if you read this book and if you learn Scheme in general, you're likely to become very comfortable with recursion, because Scheme is really based on recursion. There's no loop constructs like there are in most languages (or anything does look like a loop, it's actually tail recursion which we'll talk about later)..

So it's sort of the secret that Scheme doesn't really have loops in the way most languages do, although it turns out you can implement loops that are equivalent to a certain type of recursion..

The Structure and Interpretation of Computer Programs by Hal Abelson, Jerry and Julie Sussman

- The Structure and Interpretation of Computer Programs

Another great resource is *The Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald J Sussman with Julie Sussman. This is very famous as maybe the greatest textbook in the history of computer science - it was used as the introductory textbook at MIT for many years. It has all sorts of interesting ideas about computation and interpreters - how to write interpreters, how you write a program that interprets another program, those sorts of ideas, computational models. Great book, full of deep fascinating ideas.

Once again not a manual on Scheme - Scheme is introduced as needed as a computational notation which really gets back to the heart of Lisp, the Lisp family of languages. You need to have some sort of notation to talk about computation, so let's have a very elegant, simple, minimal notation that's compositional, has nice properties, and we can think about that's at the heart of Scheme, and that's one reason I really like Scheme.

MIT 6.001 1986 Lecture Series

- MIT 6.001 1986 Lectures on YouTube

There's a series of videos based on the ideas in the books called the *SICP Lectures* and if you're interested in the book you might like these videos.

I will say that the videos can be - and the book can be - a little mathy, especially in terms of the examples. They aim towards the MIT entering class, or in this case, for these lectures I don't know if these are supposed to be more experienced programmers or whatever, but it can turn people off a little bit. So if you find that this is a little hard going, maybe start with something like *The Little Schemer* first, which is really written for the bright high school student, but the ideas are very deep and powerful, and can benefit anyone. So maybe start with something like *The Little Schemer*, and if you feel that's not enough, you could try *The Reasoned Schemer* which is a much harder book, but in any case you could build yourself up towards the ideas in SICP over time.

Software Design for Flexibility by Chris Hanson and Jay Sussman

- Software Design for Flexibility

There's a follow-on book - intellectually a follow-on book - called *Software Design for Flexibility*: *How to Avoid Programming Yourself into a Corner* by Chris Hanson and Gerald J Sussman. this is metaphorically the advanced version or the advanced follow-up of *Structure and Interpretation of Computer Programs*. It's really about how do you write software, how do you write complicated programs in such a way, that if you want to make a significant change to the program, you don't have to throw away all your code and start over again, which turns out to be quite tricky. So the book uses a bunch of techniques people in the Scheme and Lisp community, and AI communities, have developed over a long period of time to make software more flexible.

OK, so it's full of really interesting techniques should probably talk about that at some point. Once again, not a book on Scheme but it contains a lot of really advanced uses of Scheme - you can learn a lot by looking at this book.

Simply Scheme: Introducing Computer Science by Brian Harvey and Matthew Wright

- Simply Scheme: Introducing Computer Science

If you want something maybe closer to a traditional textbook there's *Simply Scheme* by Brian Harvey and Matthew Wright. Now you can see the second edition is from 1999, so it's, you know, a quarter of a century old - Scheme has changed and so forth, but the basic ideas will be similar.

So there there are some other introductory textbooks on Scheme. You'll find most of them are kind of out of date in terms of the specific language features, but the important thing is how you think about the language. You can learn about the new variants of Scheme over time.

Scheme and Functional Programming Workshop

- Scheme and Functional Programming Workshop

There are also a bunch of papers that you can you can find on Scheme. There's a *Scheme and Functional Programming Workshop*, it's been held for a number of years, since 2000. Hopefully there'll be a 2024 one happening soon. I'm on the Steering Committee for this Workshop and I've organized it several times, so if you look at one of these workshops like from last year, you can find the Program and the Accepted Papers, and this sort of thing. So here's a pre-print and you find this is a recorded talk, on "Designing a Language for Learning Continuations".

You can find the video online - lots of good resources there. This is going to be more advanced, of course, than a textbook, but this is where a lot of the ideas get percolated - or more like disseminated - through the community, is through papers or researchers talking to each other.

If you don't have access to researchers talking to each other, if you're not in grad school, if you're not a working academic, if you're not a Scheme implementer, then reading papers is probably the best thing to do.

Lambda the Ultimate

- Lambda the Ultimate (papers page)

Certainly one of the best things to do now, because of the internet, and the web, there are lots and lots of videos you can watch and learn things that way as well. And of course, communicate with people via forums and all the amazing things that we have now. Speaking of forums, you know there's *Lambda the Ultimate*, the Programming Languages Web Log, which used to be more active. I don't know to what extent it is active these days, but there's a period in time where this was a really important resource for people trying to understand functional programming. And there's a group of papers here called Lambda the Ultimate papers, where the website got its name, and you see *Lambda the Ultimate Imperative*, for example (Uh, I don't know what will happen if I try opening that, let's see how about that one yeah let's try it - I have no idea - oh, okay)..

Well, anyway these are papers with Guy Steele and Jerry Sussman, these the original Lambda the Ultimate papers: Lambda the Ultimate "X". These are papers that were describing Scheme and using Scheme or using Lisp to solve different types of problems and looking at Lips or Scheme from different perspectives, so these are classic papers.

"Introduction to Lambda Calculus," ok, "GOTO Statement Considered Harmful" by Dykstra, a whole bunch of interesting papers, "Essence of Compiling Continuations," so, you know, these are classic papers in functional programming, in programming languages, that are worth taking a look at, and many of these are involving Scheme or some version of Lisp.

Scheme.org

- Scheme.org

There's also this *scheme.org* website, there at least used to be a website called "Read Scheme", I don't know if that still exists, but you know Scheme's been around long enough that websites come and go. The scheme.org website seems to have lots of useful resources, and you can see that there are different standards and implementations, and so forth, great things about Community, books...

SchemeDoc Bibliography

- SchemeDoc Bibliography

Here is a Scheme bibliography on GitHub and this lists all sorts of lambda papers, and papers on macros (well maybe it doesn't... Like I said Scheme's been around a long time, so I don't know why these things aren't working...) So, you know, there are lots and lots of papers on different topics like reflection, a bunch of really interesting papers on reflection in Scheme, and we'll talk about those too.

R5RS, R6RS, R7RS Pages

- R5RS Page
- R5RS Standard PDF
- R6RS.org
- R7RS.org

All right, another way if you really want to learn Scheme, is to start looking at the standards - like R5RS: that's the *Revised Revised Revised Revised Revised Report on the Algorithmic Language Scheme*, so there's this tradition of having "Revised" repeated a bunch of times, so R6RS - add one more "Revised" at the beginning, so be Revised to the sixth power.

R7RS, which is the standard currently being developed has "Revised" to the seventh power so that's if you hear people talk about R5RS, R6RS, they might just call it R6, they're talking about different versions of the Scheme specification standard. R5RS was the standard that was in place when I started graduate school, and when I started studying Scheme in a serious way.

Normally you hear about people talking about R4RS, R5RS, R6RS, R7RS. R7RS is broken into two parts - a "large" and a "small" - but those are the standard things you hear about, so if someone's talking about R5RS Scheme, that's what they're referring to - they're referring to that version of Scheme.

And you can get the report which is pretty short - it's 50 pages long - this is a beautiful document, the R5 document, oh I love it, one of my favorite documents in computation. And you know, it has an index, it has a denotational semantics which has some problems, but it contains a complete semantics for the programming language, and an index and descriptions of the core features of language, and the syntax, and all that, and examples, and the whole thing's 50 pages long!

So you can read 50 pages, it's not that long, there'll be things you don't understand, but there'll be lots of things you do understand, because this is written actually very well, and if you take a Scheme interpreter, or Scheme compiler, and you try going through the examples with your Scheme compiler, you can get a lot of mileage, just from reading through this document.

More recently, there's the *Revised Revised Revised Revised Revised Revised* (I think I got that right) *Report on the Algorithmic Language Scheme* Revised to the sixth power or R6RS, and you can find it on R6RS.org. There are different versions of the specification. Now for R6, they broke up the specification into both the language specification, and the standard libraries, so this is a bigger version of Scheme, significantly bigger than R5RS.

It's somewhat controversial that it was so big. Now, compared to a language like Java, it's tiny, but it's still much bigger than R5, which was bigger than R4, and there are some people in the Scheme community who love the tiny tiny tiny core language that's easy to implement, easy to teach, easy to understand, and there's some people who want a pragmatic practical language for everyday programming practical tasks - they want a bigger language. In R6, they found something of a middle ground, that didn't make everyone happy. Maybe it made no one happy! But that was the approach taken for R6, and so they broke the standard libraries into a different document, and they also have these Non-normative Appendices, and Rationale.

This is great if you want to understand how a language is designed, so this is a discussion about *why* they did certain things. Ok, so not only is there a a specification, and in this case a different type of semantics, this I believe is operational semantics, but they also talk about why they made these decisions. I think that's pretty interesting.

Now, if you want to get your hands on a specification, and understand it, what I probably would recommend is starting with R5, just because it's so small. R5RS doesn't really describe things like hygienic macros in a lot of depth and there are some things that are frankly mistakes in or or oversights in the spec, so I wouldn't say that this is the final word, even for the R5 version. However, it's so small and it's so well written, that I think it's a great way to get a handle on what a spec looks like.

Or you could just dive into R6, but just be aware that the this is a bigger language and and specifies a lot more, and this libraries part is large as well.

So, ok, and then there's an R7RS, which I was talking about which is still being developed. There's a "small" language and a "large" language. So like I said, there was some controversy over R6RS, because it was a bigger standard than R5, and yet it didn't include things like threads, it didn't include lots of information about dealing with the operating system and the web. So people who wanted to use Scheme as a a teaching language, or as a language where they could easily hack up an implementation and play around with ideas - that way - some of those people weren't so happy with R6, because it was bigger and more complicated than R5, harder to implement, harder to teach... And at the same time, people who want to do web programming, let's say, or create a video game in

Scheme, and interact with graphics or whatever complicated thing, they weren't happy, because R6RS didn't specify that.

But what R6RS *did* specify was a library system where people could (at least in theory) write code that could be modularized, and maybe ported between different implementations. Before R6RS, there really wasn't a chance to port code easily between Scheme implementations so R6 was a step in that direction.

And R7RS is a reaction to some extent (I think) against R6 where they say: "Ok, let's have a small language for the people who want a tiny jewel-like gem of a language, that's easy to implement, easy to understand, they can hack it to their heart's content for, you know, programming languages research, or whatever they want to do; they can have the small language".

And for the people who want to have a really big language, they have the big language. So they split up the spec, which is an interesting idea. What's happened in practice is that the small language design was completed in 2013. The large language design is still going on, so, 11 years after the small language was finished, and I guess this was sort of, almost a guaranteed outcome when you're trying to make a much bigger version of the language, that's not supported by a bunch of industry, and government research labs or whatever...

So this spec is still going on. You will see for any individual Scheme implementation, it's likely to say which version of the spec is supported: they support R4RS, they support R5RS, R6, R7, "R7 small".

Nothing supports "R7 large" completely because that spec is still being developed...

Chez Scheme currently supports R6RS, and even for something like R6RS or R5RS, you'll often see that an individual implementation will describe differences from the spec. So in fact, for some versions of the specification, the specification itself is internally inconsistent, and it's impossible to implement a correct version, or at least people have made that argument: that it's impossible to create a fully conforming implementation...

Also there are some subtleties in the spec that make a little bit of a design and implementation tradeoff. To decide to implement every single thing in the spec so many implementations will say they implement RNRS whatever N is, modulo this one little feature...

Ok, so those are the the language specs and it's really important if you're going to use Scheme, or any other language, to be comfortable with the definition of the language, I think.

A nice thing about Scheme is this the the specification is actually pretty small - if you want to learn C++ or Java those specifications are quite complicated although C++ they seem to be (maybe) simplifying. Java still seems to be getting more complicated in a way, although they are adding abstractions that hopefully are helpful.

The Scheme Programming Language by R. Kent Dybvig

- The Scheme Programming Language

Ok, another way to learn about Scheme is to read this book. This is a book I refer to all the time, which is called *The Scheme Programming Language* (4th Edition) by Kent Dybvig, who's creator of Chez Scheme, and who also was the editor of R6RS, that spec. And so this is describing in the 4th Edition, you know, basically R6RS Scheme, but instead of describing it in a very formal way, it is describing Scheme from the standpoint of examples and motivation, and just gives a lot more detail.

So it's not really a textbook or tutorial (although it has aspects of that), if you didn't know how to program at all I think this would be a hard book to learn programming from, but it's a great book to go to, to try to understand what the specification is trying to say. So if there's part of R6RS that's complicated, or you just don't know about a feature at all, if you read about it in this book, and then look at the spec - then look at R6RS - then that will give you a much better idea of what's going on, and since Kent was the editor for the R6RS process, he understands what the motivation was, and he's also an implementer, so he knows inside and out not just the motivation from the specification process, but how at least Chez Scheme implements those parts of the language, or at least how how Chez Scheme implemented it when he wrote this book! Very, very useful resource; the book's very well written.

Chez Scheme

- The Chez Scheme User's Guide
- Chez Scheme on Github
- Chez Scheme Github Repo

Another companion guide that goes to that is the *Chez Scheme User's Guide*, so like I said Kent Dybvig is the creator of Chez Scheme and he has a User's Guide which now, I guess, other people have contributed to this guide. Certainly, originally I think this was written by Kent and it ties to the *Scheme Programming Language* book - these are cross referenced with each other - very helpful.

So if you're using Chez Scheme, which is the implementation (along with Racket) that I use most often, of Scheme or Scheme-like language, then this is a very helpful guide. And because it cross references with this book, talking about R6RS, you can learn a lot about the language, the spec, and about how Chez does things and where Chez differs from the specification. So, two really useful resources.

One thing that's changed since when I started grad school, let's say, at Indiana, was that Chez Scheme, which has always been known as a high-quality Scheme implementation, used to be commercial software that you had to pay an expensive license to be able to run. Cisco ended up purchasing Chez Scheme, and getting Kent and other Chez hackers to work with them on a project for a number of years, so you can see that this *User's Guide* is actually from Cisco. And Cisco eventually released Chez Scheme Version 9 as open source software under an open source license! So now you can run the full version of Chez, the full compiled version, whereas before you could only use the interpreter. And Chez Scheme is now available on GitHub.

So I use Chez Scheme a lot and will be using Chez Scheme at least some of these videos and so here you can see the cisco.github.io Chez Scheme page, there's "Get Chez Scheme", "Documentation" and so on...

Scheme Related Papers by CollectRobot

- Scheme Related Papers by CollectRobot
- How to Debug Scheme Programs
- Generation-Friendly Eq Hash Tables

So, last night, when I was preparing for a video or preparing to make a bunch of videos, I started going through some of this documentation in the Chez Scheme Guide, and in particular, I wanted to learn how to use a Chez Scheme debugger better. Chez Scheme has an integrated debugger, it also has tracing facilities, all sorts of nice features, but I've never really been comfortable using a debugger, partly because when I was in grad school, the debugger didn't work that well, because I only had access to the interpreter, not the compiler.

The debugger really is much better if you had the compiler for a Chez Scheme, which I have access to now, because it's open source, so I want to go back and learn about the debugger. When I was reading the debugging section, I noticed that - well probably because this was written 10 or more years ago, maybe 15 years ago now - there's this dead link to "How to Debug Chez Scheme Programs". Fortunately someone has found Kent's instructions on how to debug Chez Scheme programs, and you can find it online on GitHub.

And I found some other little typos, I think this is a typo, "`trace-let` may be used to trace ordinary `let` expressions as well as `let` expressions" - that I think is a typo. I don't know, I'm a little confused.

Here's something looks like a copy and paste error: "box inspector objects: box inspector objects contain Chez Scheme boxes" - okay - and then "TLC inspector objects: box inspector objects contain Chez Scheme boxes". That looks like a copy and paste error to me and I didn't even know, by the way, what a TLC inspector object was - I'd never heard of that - it looks like that's coming from this paper - my friend Aziz Ghuloum and Kent wrote on *Generation-Friendly Eq Hash Tables*, so Scheme `eq` is pointer-equality; generation-friendly has to do with generational garbage collection.

I mean we talk about that, but it turns out that these TLC things are *Transport Link Cells* - I never heard of that term before, it has to do with the way they implement these hash tables, so this documentation is a Transport Link Cell inspector object. I think there's just a little copy and paste mistake and I think I found one or two other typos because Chez Scheme is now open source, and you can find, not just the code for Chez Scheme, but also the *Chez Scheme User's Guide*, and along with that the *Scheme Programming Language* 4th Edition, all these files are here, we can go, and we can look at `debug.stex` - which is like a Scheme version of TeX, I believe, that Kent came up with, this is like a Scheme TeX - and it turns out that, sure enough, these typos (what I believe are typos) are in the documentation. And I don't know if Kent's still maintaining this or not, but there are people still working on Chez Scheme, and in particular the Racket people are rebuilding - they've rebuilt - Racket on top of Chez, so they're active in keeping Chez up to date.

And then some of the people who worked on Chez at Cisco, like Andy Keep, are still, you know, keeping an eye on it, and making updates, so I emailed Andy to ask what should I do? Should I do a pull request, or, you know, send him or someone else the typos I've found?

So that, especially when you're part of a small community, but reading books as well, it's a service that I think is important, so not only do you (hopefully), feel comfortable reading specifications, and reading formal documentation, reading books, but if you find an error, it's really helpful to the authors if you could report that error somehow. So that's what I'm going to do, to try to be a good citizen in the Chez Scheme world; to try to report these little typos. You know, there's nothing here I couldn't easily understand, so I don't think it's a any sort of deep error, but let's go ahead and try to correct all those. So, if you're reading documentation and you think something's an error...

Now, if you don't know the language well, if you don't know the documentation well, then maybe it's hard to tell, so maybe ask someone else first. But, if you're pretty sure it's just a typo, maybe go ahead and email the person who created it, or send a pull request. And that's one way we can try to make these languages, and documentation, and tutorials, books, better over time...

So that is just a quick overview of some resources for learning Scheme. And, you know, the importance I think of learning a spec from the official specification; trying to read those..

The Scheme spec is very short, so if you're interested in Scheme, I recommend maybe starting with R5, or just starting with R6, or you could do R7 small. And trying to read those specs, you could read all of them if you if you're really into it, trying to see how far you can get understanding it.

And in the intro to R5 - and I don't know how how old this intro is - there's this marvelous philosophical statement about the design of Scheme how the philosophy behind Scheme's design. In particular - I love this sentence - "programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary."

I love it! "Scheme demonstrates a very small number of rules for forming expressions with no restrictions on how they or compose suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today."

Oh a very strong sense of design - I love it - this is very different from say the design of Java! Okay, very different.

So, it's interesting when you're learning languages or thinking about languages, compare and contrast the philosophies of language, and then how the languages change over time, are they keeping up with this philosophy, or are they changing somehow?

I think certainly through R5 Scheme had this philosophy. Some people I think said that R6 maybe went too far away from being this minimum language, with the smallest number of features; R7 small is more like this philosophy...

Whether or not the larger versions are more useful? That's an interesting question.

In any case um I hope you will take a look at the R5 report and see if you can make sense of that. Parts of it are very easy to read, parts of it are kind of complicated... And if you're not familiar with denotational semantics - well, it's going to be really hard to read; if you don't know anything about hygienic scheme macros, it's going to be hard to read.

But we can talk about some of those topics in these videos.

All right see you soon, I got 1,020 videos to make!

Bye-bye!

Appendix A - Dev Notes for this Booklet Project

To upload my repo to Github, I ran:

```
$ gh repo create
```

To create a `gh-pages` branch in Github, I followed this advice:

```
git checkout --orphan gh-pages
git reset --hard
git commit --allow-empty -m "Initializing gh-pages branch"
git push origin gh-pages
git checkout main # was master
```

I added entries to the Makefile provided here

Now, I run

```
git commit -a -m "publish"
git push
make deploy
```

Index

Andy Keep, 1
Chez Scheme, 1
Chez Scheme User's Guide, 1
Chris Hanson, 1
Daniel P Friedman, 1
Gerald J Sussman, 1
Guy Steele, 1
Harold Abelson, 1
Kent Dybvig, 1
Lambda the Ultimate, 1
Matthias Felleisen, 1
Racket, 1
Scheme and Functional Programming Workshop, 1
scheme.org, 1
SICP Lectures, 1
Simply Scheme, 1
Software Design for Flexibility, 1
The Little Schemer, 1
The Reasoned Schemer, 1
The Scheme Programming Language, 1
The Structure and Interpretation of Computer Programs, 1