Solving SICP: An Experience Report on Solving the World's Most Famous Programming Problem Set

Scheme Workshop 2020
(The International Conference on Functional Programming)

Vladimir Nikishkin
<lockywolf gmail.com>

<2020-08-28 Tue>

Outline

Introduction. Task and Tools.

The Execution Process.

The Data and the Analysis.

Results and Conclusion.

Outline

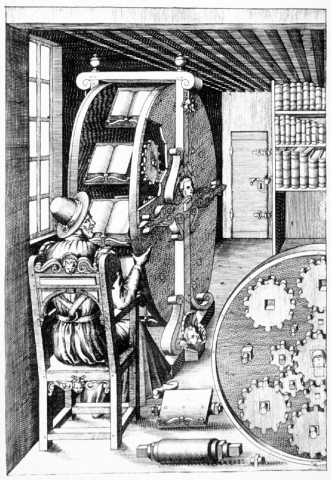
Introduction. Task and Tools.

The Execution Process.

The Data and the Analysis.

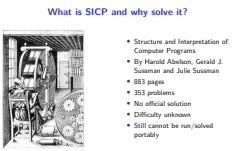
Results and Conclusion.

What is SICP and why solve it?



- Structure and Interpretation of Computer Programs
- By Harold Abelson, Gerald J. Sussman and Julie Sussman
- 883 pages
- 353 problems
- No official solution
- Difficulty unknown
- Still cannot be run/solved portably

+



Note In this talk I want to speak about a very famous book by two very famous MIT professors. It is called "Structure and Interpretation of Computer Programs". This book, as well as the course it is associated with, are considered among the most unconventional among the programming courses to day. (Although HTDP rivals it.) It is still unconventional, even though 35 years have passed, and it would be reasonable to expect that the techniques considered innovative would become an everyday norm. This hasn't happened to the extend expected. Why? To find out why, and also to become a better programmer, was my aim at the start of the project. I once heard that those who passed SICP are among the best programmers in the world.

Who is this report for?

Providers

- Teachers
- Teaching Assistants
- Curriculum designers

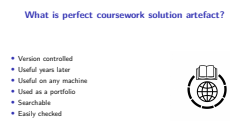
Listeners

- Students
- Time-management enthusiasts
- Self-learners

Note I hope that this report can be useful to people who deal with computer science education in their daily life. Curricula are seldom a subject of independent scrutiny, and the two parties taking part in the education process are likely to suspect each other of dishonesty. Therefore, having an independent assessment that can be trusted will happen to be a valuable contribution. This is especially valid now, when there is an increasing trend on making education a more remote and a more solitary process. Furthermore, it is also becoming increasingly computerised, so both parties can get some benefit from that, if knowing how to.

What is perfect coursework solution artefact?

- Version controlled
- Useful years later
- Useful on any machine
- Used as a portfolio
- Searchable
- Easily checked



Note Naturally, paper is still the most commonly used result of education. In a better case, the paper is written by the students, and they remember a bit by writing. In a worse case, it is handed out to the students by a lecturer, and kept as a memory of the good old days. Students are usually not prepared for the university educational process, and don't know how to behave in order to make this time the most useful. As a consequence, most of the work done by the students during studies becomes worthless right after a deserved mark is written into the transcript. However, if approached open-mindedly, at least two possible applications for the work done in a learning time can be thought of. Firstly, university knowledge can be processed into a "canned brain food" for later queries. Secondly, the coursework and/or lectures can be formatted as a piece of portfolio to be presented to an interviewer at a job position. It's a shame that students are seldom told that before their first lectures. I tried to imagine what a "perfect coursework submisstion format" should be. It should retain the consistency, time and machine independence of a book. But at the same time, it should be searchable, version-controlled and runnable. A so-called "notebook" format similar to Jupyter thus appeared.

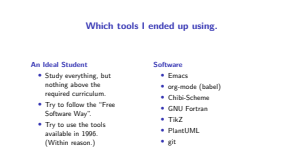
Which tools I ended up using.

An Ideal Student

- Study everything, but nothing above the required curriculum.
- Try to follow the "Free Software Way".
- Try to use the tools available in 1996. (Within reason.)

Software

- Emacs
- org-mode (babel)
- Chibi-Scheme
- GNU Fortran
- TikZ
- PlantUML
- git

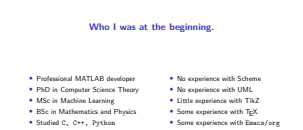


Note

- Of course, TikZ and PlantUML did not exist in 1985.
- Chibi-Scheme also did not exist.
- Scheme existed, obviously, in the form of r4rs. Now we have r7rs.
- TikZ could have been METAPOST.
- git could have been RCS. It is a shame that people still cannot use git while at uni.
- PlantUML did not exist, but I find it very useful to be able to adapt SICP's illustrations to standard diagrams.

Who I was at the beginning.

- Professional MATLAB developer
- PhD in Computer Science Theory
- MSc in Machine Learning
- BSc in Mathematics and Physics
- Studied `C, C++, Python`
- No experience with Scheme
- No experience with UML
- Little experience with TikZ
- Some experience with TEX
- Some experience with `Emacs/org`



Note

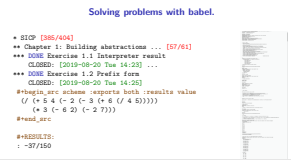
- I am giving this information so that people who are consulting the solution be able to rescale the difficulty to themselves or the target audience.
- Initially I thought that having certain programming experience should make me solve SICP's problem set noticeably faster than a newbie would. Doesn't seem to be the case.
- Full time employment meant that I only had weekends and evenings for work. Still, students usually have classes and other courses.

Solving problems with babel.

```
* SICP [385/404]
** Chapter 1: Building abstractions ... [57/61]
*** DONE Exercise 1.1 Interpreter result
    CLOSED: [2019-08-20 Tue 14:23] ...
*** DONE Exercise 1.2 Prefix form
    CLOSED: [2019-08-20 Tue 14:25]
 #+begin_src scheme :exports both :results value
  (/ (+ 5 4 (- 2 (- 3 (+ 6 (/ 4 5)))))
     (* 3 (- 6 2) (- 2 7)))
 #+end_src

 #+RESULTS:
 : -37/150
```

+



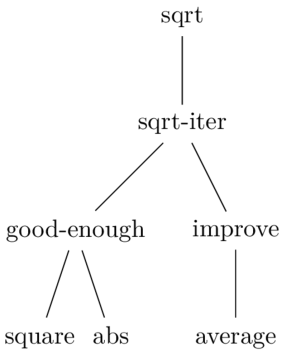
Note This is an example of how the solution looked like. On the right you can see the file's minimap. On the left, there is a typical example of two solved problems. As I mentioned, a "notebook format" had to be found. It is called "org-mode". Look at the code block and the result. An instant benefit, which eventually served as a main substance of this report, is automatic gathering of measure. In particular, progress, completion time, and session start and finish time stamps (in a separate file).

Graphical example with TikZ. (Figure 1.2)

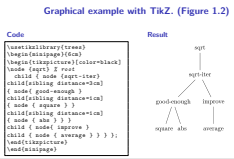
Code

```
\usetikzlibrary{trees}
\begin{minipage}{6cm}
\begin{tikzpicture}[color=black]
\node {sqrt} % root
  child { node {sqrt-iter}
child[sibling distance=3cm]
{ node{ good-enough }
child[sibling distance=1cm]
{ node { square } }
child[sibling distance=1cm]
{ node { abs } } }
child { node{ improve }
child { node { average } } } };
\end{tikzpicture}
\end{minipage}
```

Result



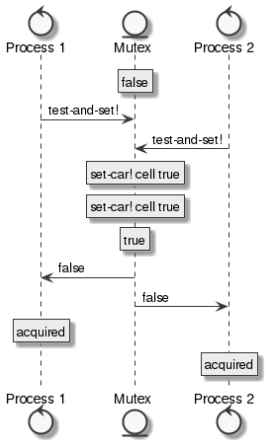
+



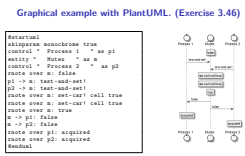
Note TikZ is quite verbose. I used tikz when the exercises required drawing something. I also redrew several figures with Tikz, as I wanted to be able to reproduce the book's narrative in my own classes (if/when I am going to give them). Where possible, I tried to use more specific drawing tools. Note, I did not use TikZ for the "Functional Drawing", the so-called "Picture Language" part of SICP. For it I had to implements my own library, using Imagemagick.

Graphical example with PlantUML. (Exercise 3.46)

```
@startuml
skinparam monochrome true
control "  Process 1   " as p1
entity "   Mutex   " as m
control "  Process 2   "  as p2
rnote over m: false
p1 -> m: test-and-set!
p2 -> m: test-and-set!
rnote over m: set-car! cell true
rnote over m: set-car! cell true
rnote over m: true
m -> p1: false
m -> p2: false
rnote over p1: acquired
rnote over p2: acquired
@enduml
```



+

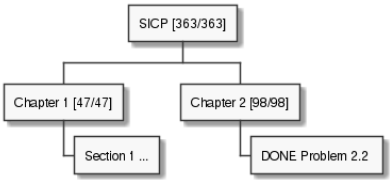


Notes PlantUML's biggest drawback is that its syntax is ugly and fragile. However, the team behind it is working on improving the tool, and it seems that they are learning in the process. UML is not an as pathetic thing as it is usually thought. Standard tools for code visualisation are certainly worth considering. On the other hand, environment diagrams, or essentially debugging interfaces, had to be drawn with TikZ. Which I find amusing. Such a feature is still not possible to implement portably.

How to measure progress and motivate yourself.

Measures

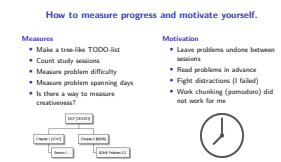
- Make a tree-like TODO-list
- Count study sessions
- Measure problem difficulty
- Measure problem spanning days
- Is there a way to measure creativeness?



Motivation

- Leave problems undone between sessions
- Read problems in advance
- Fight distractions (I failed)
- Work chunking (pomodoro) did not work for me

+



Note

- In high school we cross out the tasks as they appear in a problem set.
- We feel better as things progress.
- Measuring problem difficulty requires sequentiality.
- Org can be compiled into a wbs-chart (not implemented)
- Noise can be fought with headphones.
- Pomodoro did not work because I could not fit problems in chunks reasonably.

Looking for help.

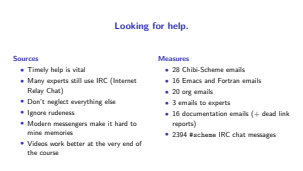
Sources

- Timely help is vital
- Many experts still use IRC (Internet Relay Chat)
- Don't neglect everything else
- Ignore rudeness
- Modern messengers make it hard to mine memories
- Videos work better at the very end of the course

Measures

- 28 Chibi-Scheme emails
- 16 Emacs and Fortran emails
- 20 org emails
- 3 emails to experts
- 16 documentation emails (+ dead link reports)
- 2394 `#scheme` IRC chat messages

+



Note

- I didn't manage to collect and reprocess memory from modern messengers.
- The videos are like "ahh, that's what they actually meant by what they said."
- In general, seeing "how much I have already done!" and "there is a limited amount of things to do" is a great feeling that makes you get back to work.
- IRC is still a useful tool.
- But the modern ones are still better to not be neglected.
- Communication is important, and getting questions answered fast is a great thing.

Measured data examples.

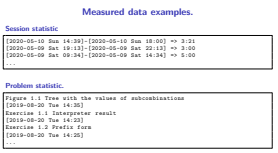
Session statistic

```
[2020-05-10 Sun 14:39]-[2020-05-10 Sun 18:00] => 3:21
[2020-05-09 Sat 19:13]-[2020-05-09 Sat 22:13] => 3:00
[2020-05-09 Sat 09:34]-[2020-05-09 Sat 14:34] => 5:00
...
```

Problem statistic.

```
Figure 1.1 Tree with the values of subcombinations
[2019-08-20 Tue 14:35]
Exercise 1.1 Interpreter result
[2019-08-20 Tue 14:23]
Exercise 1.2 Prefix form
[2019-08-20 Tue 14:25]
...
```

+



Note Problem statistic is indicative. I first solved the Exercises 1.1 and 1.2, and then turned to the Figure 1.1. But it is displayed earlier, because it is earlier in the book.

Solving problems has a little bit of robustness to non-sequentiality. In general, SICP tries to enforce sequentiality by making problems depend on one another, and this gives the noweb-like features of SICP a great value.

Having time tracking data in a machine-accessible format made it possible to do the analysis.

Data analysis with Emacs Lisp.

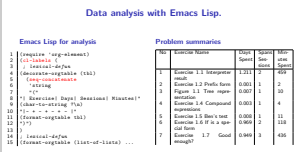
Emacs Lisp for analysis

```
1  (require 'org-element)
2  (cl-labels (
3   ; lexical-defun
4  (decorate-orgtable (tbl)
5    (seq-concatenate
6     'string
7     "("
8  "| Exercise| Days| Sessions| Minutes|"
9  (char-to-string ?\n)
10 "|- + - + - + - |"
11 (format-orgtable tbl)
12 ")")
13 )
14 ; lexical-defun
15 (format-orgtable (list-of-lists) ...
```

Problem summaries

No	Exercise Name	Days Spent	Spans Sessions	Minutes Spent
1	Exercise 1.1 Interpreter result	1.211	2	459
2	Exercise 1.2 Prefix form	0.001	1	2
3	Figure 1.1 Tree representation	0.007	1	10
4	Exercise 1.4 Compound expressions	0.003	1	4
5	Exercise 1.5 Ben's test	0.008	1	11
6	Exercise 1.6 If is a special form	0.969	2	118
7	Exercise 1.7 Good enough?	0.949	3	436

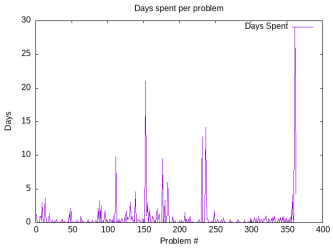
+

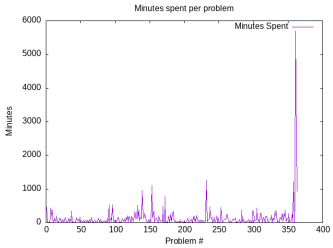


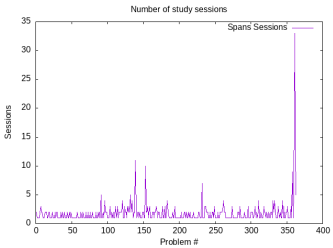
Note

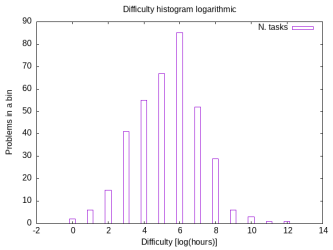
- This is called "Reproducible Research".
- seq-* functions help elisp be like scheme.
- cl-labels are like lexical defuns.
- The ten problems on the right are the example of the table that the code generates, and the table can be further analyzed by Emacs Lisp.
- The three measures are "raw time", "wall clock time" and "total days in memory".
- They all are not totally dependent.
- It is possible to offload some thinking into the unconscious.
- It is easier to return to the work when you have something undone.
- Problems were never ended at the same time as sessions.

Data demonstration.

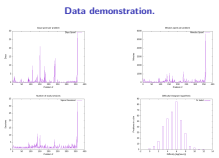








+



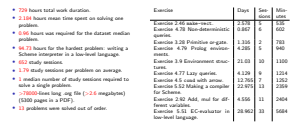
Notes Not really readable graphs depict the distribution of the three measures of the dataset. Problems that take more days than sessions are essentially the days when I needed a holiday. But students also have holidays. Why exactly the problem set obeys a log-normal distribution? The two most difficult problems take most of the time. To me it would mean that those have to be broken into smaller bits. Even though is says "translate scheme to a low-level language line by line", the runtime support is huge, and not very well explained in SICP. Input-output is not explained at all. This is not what students usually do in the university, but perhaps it is not such a bad thing.

Statistics and ten hardest problems.

- 729 hours total work duration.
- 2.184 hours mean time spent on solving one problem.
- 0.96 hours was required for the dataset median problem.
- 94.73 hours for the hardest problem: writing a Scheme interpreter in a low-level language.
- 652 study sessions.
- 1.79 study sessions per problem on average.
- 1 median number of study sessions required to solve a single problem.
- >78000-lines long .org file (>2.6 megabytes) (5300 pages in a PDF).
- 13 problems were solved out of order.

Exercise	Days	Sessions	Minutes
Exercise 2.46 `make-vect`.	2.578	5	535
Exercise 4.78 Non-deterministic queries.	0.867	6	602
Exercise 3.28 Primitive or-gate.	1.316	2	783
Exercise 4.79 Prolog environments.	4.285	5	940
Exercise 3.9 Environment structures.	21.03	10	1100
Exercise 4.77 Lazy queries.	4.129	9	1214
Exercise 4.5 `cond` with arrow.	12.765	7	1252
Exercise 5.52 Making a compiler for Scheme.	22.975	13	2359
Exercise 2.92 Add, mul for different variables.	4.556	11	2404
Exercise 5.51 EC-evaluator in low-level language.	28.962	33	5684

+



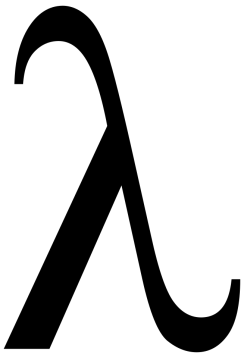
Note

- make-vect – writing the whole picture language
- non-deterministic – whole rewrite
- primitive or-gate – assemble a simulator
- prolog-env – open-ended
- environment structures – TikZ
- lazy-queries – a large architectural piece
- cond with arrow – assemble a metacircular evaluator
- scheme compiler – huge
- add+mul different variables – huge work with normalization
- ec-evaluator – learning fortran
- Three working months (>700 hours)

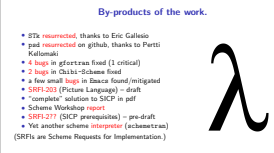
By-products of the work.

- STk resurrected, thanks to Eric Gallesio
- psd resurrected on github, thanks to Pertti Kellomaki
- 4 bugs in gfortran fixed (1 critical)
- 2 bugs in Chibi-Scheme fixed
- a few small bugs in Emacs found/mitigated
- SRFI-203 (Picture Language) – draft
- "complete" solution to SICP in pdf
- Scheme Workshop report
- SRFI-2?? (SICP prerequisites) – pre-draft
- Yet another scheme interpreter (schemetran)

(SRFIs are Scheme Requests for Implementation.)



+



Note This project gave birth to a healthy amount of by-products. The most universally useful is probably the bug fix in gnu fortran, which I used for the last two exercises. Twice I got a remark from old software writers that "software never dies". I didn't manage to port PSD to a modern Emacs, but this probably can be done eventually.

Let's talk about portability. Even though SICP is believed to be a book about scheme, it is still not possible to finish it with a scheme system supporting only the base standard, even if it is r7rs. The critical points are multi-threading, randomness, and most prominently, graphics. I think that this is a drawback, and should be addressed. I already submitted an srfi on the most unportable part. The second, one on the parts that are implementable with the help of other SRFIs, is in the plans.

Applications and Further Work.

Applications

Teachers monitor dropout
Teachers make marking simpler
Students make portfolios
Students monitor work
Designers make coursework templates
Designers get feedback

We Need More Data!

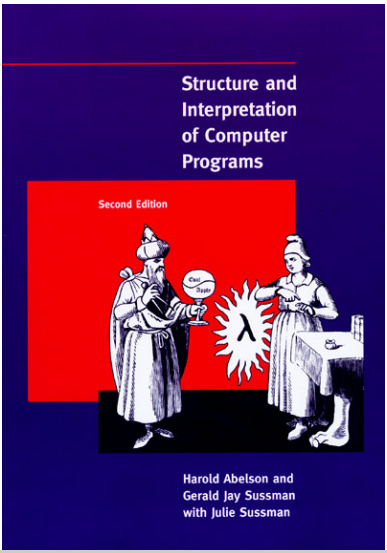
- Get more data points for SICP!
- Behavioural analysis of the existing data.
- Measure other course work!
- Profile-guided optimisation.
- Social/cloud service.
- Effort tracking? (e.g. window switching)

+

Notes On this slide I want to once again restate that for me this project can be seen as a model according to which university coursework could be considered. Having an established notion of a "standard coursework" would allow to calibrate one's own perception. Am I late or ahead? Do I have enough time scheduled for the work? Where am I in the process. How do I recall the learned topics in the future?

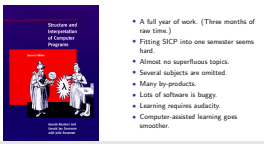
Same things apply to the teachers, only regarding their students. In addition, it may be mildly nice for the teachers to have an example of a "well-done coursework" that can be shown to the students. Obviously, a single-point estimate is not very good. Teachers have a power over their student, at least in the form of exposing them to the existence of this report.

Review.



- A full year of work. (Three months of raw time.)
- Fitting SICP into one semester seems hard.
- Almost no superfluous topics.
- Several subjects are omitted.
- Many by-products.
- Lots of software is buggy.
- Learning requires audacity.
- Computer-assisted learning goes smoother.

+



Notes This course is hard. Can I say that it is needlessly hard? It certainly requires spending a lot of time on learning things that are not even mentioned in the book. In fact, the book looks more like a companion to the MIT course than a standalone work. I hope that the SRFIs and this Report can compensate for that a little bit, and help those trying to solve SICP in the future by sawing off the sharp corners. All of this took roughly one year, including the Report and the SRFIs. Learning while being employed is certainly possible, but hard. Computing models presented in SICP seem not obsolete. ?

Credits.

Contacts

- `http://gitlab.com/Lockywolf`
- `http://lockywolf.wordpress.com`
- `lockywolf at gmail.com`
- `https://t.me/unobvious`
- `http://lockywolf.net`
- https://paypal.me/independentre-search

Thank you

- John Cowan
- Alex Shinn
- Eli Zaretskii
- Patrick Volkerding
- evets @ `stackoverflow`
- `irc.freenode.net/#scheme` contributors
- all my friends and relatives

Notes I want to say thank you to the `#scheme` freenode channel users. Personal thank you to the people responsible for the scheme standard development, for the implementation development, for Emacs development, Fortran developers, Slackware developers.