A set of exercises and tests for teaching tracing skills using a mastery approach

Martijn Stegeman
University of Amsterdam
The Netherlands
martijn@stgm.nl

ABSTRACT

We present a first implementation of exercises on code evaluation and tracing for use alongside introductory programming courses. The goal of these exercises and accompanying tests is to provide a structure that enables students to fully master a number of common tracing skills. In developing the exercises, we focused on keeping cognitive load as low as possible by gradually introducing new programming language elements, while allowing for repeated practice of previously introduced concepts. The exercises range from evaluating expressions involving integer division to tracing loops with multiple variables. We also generated small tests that students take to show their mastery of the concepts, allowing only a very limited number of mistakes per test and requiring students to take another version if needed. Using this model in several introductory programming courses over the past year shows that it appears to be possible to achieve mastery on these tracing skills for almost all students while maintaining positive attitudes toward the exhaustive training process.

CCS CONCEPTS

• **Social and professional topics → Computer science education**; *CS1*; *Computational thinking*.

KEYWORDS

assessment, tracing, notional machines, mastery learning

ACM Reference Format:
Martijn Stegeman. 2019. A set of exercises and tests for teaching tracing skills using a mastery approach. In *19th Koli Calling International Conference on Computing Education Research (Koli Calling '19), November 21–24, 2019, Koli, Finland.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3364510.3366154

1 INTRODUCTION

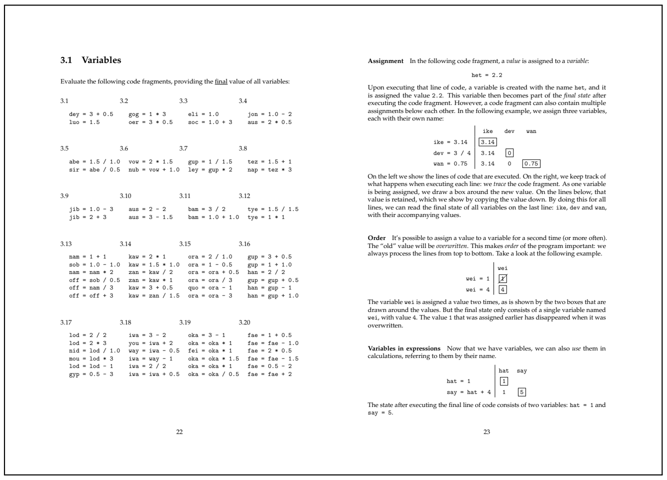
The Lister et al. [3] working group has shown that many students in introductory programming courses lack a good grasp of the basic skills needed to solve programming problems. One such skill is reading code: making sense of a program with respect to the computer that the code will run on. A notional machine is an abstraction of the layers of hardware and software that the programming language is built upon, and students need to form a "robust" mental model [8] to be able to interpret (trace) code correctly. From teaching experience, we know that tracing is an important part of the problem solving process, for example to verify correct composition of programs. Indeed, Lopez et al. [5] and Lister et al. [4] found strong associations between code writing and tracing skills. On the other hand, Du Boulay [1] found that students can learn to program using "bizarre" mental models. Additionally, Vainio and Sajaniemi [10] have found several problems with tracing. For example, students might use overly simple heuristics like keeping only a single value in working memory at a time; also, many students are hesitant to use externalization techniques like drawing a diagram. Such problems might be explained using the cognitive load theory formulated by Sweller and others, who propose that the process of solving problems may dominate working memory, leaving no room for reflection and learning by students [9].

Hence, we would like to get tracing "out of the way" early by teaching it explicitly and separate from programming assignments. Interestingly, studies on teaching how to trace by hand are hard to find. Nelson et al. [7] and Xie et al. [12] have been working on methods for teaching program comprehension early in programming courses, but many other studies exclusively focus on computer-generated traces. Here, we would like to train students to trace using pen and paper. To this end, we have written instructional materials that completely focus on tracing techniques, to be used alongside existing course materials. In our approach, we borrow aspects of a system of mastery learning aimed at higher education: the Keller plan [2]. Core principles are the ability for students to plan tests at their own pace, each covering a small part of the course syllabus, and having to repeat tests that were not completed successfully.

2 TEACHING MATERIALS

We have designed a practice book that is modeled after "All You Need in Maths!" by Van de Craats and Bosch [11]. A typical section (figure 1) contains exercises of increasing complexity, while the opposite page is reserved for an explanation of the concepts involved. In this case, the text introduces *assignment*, *order* and *variables in expressions*. Additionally, it shows a technique for tracing sequences of variable assignments. The practice book contains nine chapters that introduce programming concepts along with evaluation rules and tracing techniques. We have designed the book to use alongside a course in C, but have used an excerpt with a course in Python.

We would like students to build a robust mental model of the notional machine, so we aim to gradually introduce new language



3.1 Variables

Evaluate the following code fragments, providing the final value of all variables:

```
3.1              3.2              3.3              3.4

  dey = 3 + 0.5    gog = 1 * 3      eli = 1.0        jon = 1.0 - 2
  luo = 1.5        oer = 3 * 0.5    soc = 1.0 + 3    aus = 2 * 0.5

3.5              3.6              3.7              3.8

  abe = 1.5 / 1.0  vow = 2 * 1.5    gup = 1 / 1.5    tez = 1.5 + 1
  sir = abe / 0.5  nub = vow + 1.0  ley = gup * 2    nap = tez * 3

3.9              3.10             3.11             3.12

  jib = 1.0 - 3    aus = 2 - 2      bam = 3 / 2      tye = 1.5 / 1.5
  jib = 2 + 3      aus = 3 - 1.5    bam = 1.0 + 1.0  tye = 1 * 1

3.13             3.14             3.15             3.16

  nam = 1 + 1      kaw = 2 * 1      ora = 2 / 1.0    gup = 3 + 0.5
  sob = 1.0 - 1.0  kaw = 1.5 * 1.0  ora = 1 - 0.5    gup = 1 + 1.0
  nam = nam * 2    zan = kaw / 2    ora = ora + 0.5  han = 2 / 2
  off = sob / 0.5  zan = kaw * 1    ora = ora / 3    gup = gup + 0.5
  off = nam / 3    kaw = 3 + 0.5    quo = ora - 1    han = gup - 1
  off = off + 3    kaw = zan / 1.5  ora = ora - 3    han = gup + 1.0

3.17             3.18             3.19             3.20

  lod = 2 / 2      iwa = 3 - 2      oka = 3 - 1      fae = 1 + 0.5
  lod = 2 * 3      you = iwa + 2    oka = oka * 1    fae = fae - 1.0
  nid = lod / 1.0  way = iwa - 0.5  fei = oka * 1    fae = 2 * 0.5
  mou = lod * 3    iwa = way - 1    oka = oka * 1.5  fae = fae - 1.5
  lod = lod - 1    iwa = 2 / 2      oka = oka * 1    fae = 0.5 - 2
  gyp = 0.5 - 3    iwa = iwa + 0.5  oka = oka / 0.5  fae = fae + 2
```

22

Assignment In the following code fragment, a *value* is assigned to a *variable*:

```
het = 2.2
```

Upon executing that line of code, a variable is created with the name `het`, and it is assigned the value `2.2`. This variable then becomes part of the *final state* after executing the code fragment. However, a code fragment can also contain multiple assignments below each other. In the following example, we assign three variables, each with their own name:

	ike	dev	wan
`ike = 3.14`	$\boxed{3.14}$		
`dev = 3 / 4`	3.14	$\boxed{0}$	
`wan = 0.75`	3.14	0	$\boxed{0.75}$

On the left we show the lines of code that are executed. On the right, we keep track of what happens when executing each line: we *trace* the code fragment. As one variable is being assigned, we draw a box around the new value. On the lines below, that value is retained, which we show by copying the value down. By doing this for all lines, we can read the final state of all variables on the last line: `ike`, `dev` and `wan`, with their accompanying values.

Order It's possible to assign a value to a variable for a second time (or more often). The "old" value will be *overwritten*. This makes *order* of the program important: we always process the lines from top to bottom. Take a look at the following example.

	wei
`wei = 1`	$\boxed{\cancel{1}}$
`wei = 4`	$\boxed{4}$

The variable `wei` is assigned a value two times, as is shown by the two boxes that are drawn around the values. But the final state only consists of a single variable named `wei`, with value 4. The value `1` that was assigned earlier has disappeared when it was overwritten.

Variables in expressions Now that we have variables, we can also *use* them in calculations, referring to them by their name.

	hat	say
`hat = 1`	$\boxed{1}$	
`say = hat + 4`	1	$\boxed{5}$

The state after executing the final line of code consists of two variables: `hat = 1` and `say = 5`.

23

Figure 1: Extract of the programming book, explaining a trace of sequential assignment statements (zoom in for more detail).

elements, and independently assess these, as suggested by Luxton-Reilly et al. [6]. Hence, in our book, chapters strictly build on earlier chapters. This means that many chapters have some exercises that involve expressions with a modulo operator, while only the final chapters have functions. This is an important aspect of the book, allowing students to focus on a limited number of concepts at the same time, while being able to rehearse those introduced previously. For example, in the excerpt above we find that *integer division* and *automatic conversion*, both from an earlier chapter, are relevant to that section's exercises. Topics for all current chapters are: calculations, logic, variables, conditionals, while loops, for loops, strings, arrays, and functions. For each chapter in the book, we created a set of tests that allow students to demonstrate mastery. The tests contain exercises that are of the same level of complexity as the final exercises in the chapter. Students are expected to spend between 4 and 10 minutes per test.

3 EXPERIENCES

In 2018 and 2019, we have used the practice book and tests in three undergraduate courses at the University of Amsterdam. The courses had traditional lectures (mostly on video) as well as regular larger programming assignments with a focus on problem solving practice. Tests were compulsory but didn't count towards the final grade. Some notable observations are summarized below.

Attitudes A surprising aspect of the tests is that students find them to be quite motivating, even if they have to retake several. The way we integrated the tests into the courses apparently made students feel "safe" to do them repeatedly and a sense of accomplishment was regularly noticeable.

Prerequisites The first half of the book starts out with arithmetic and logic, and we build on that until we introduce strings in chapter 7. This does not seem to pose real problems, even with non-science students. A few students reported having some trouble with "maths" initially, and they still progressed fine on the tests (as did other students).

Answer sheets We expected students to practice during lab hours and discuss exercises with other students. Immediately from the first week there were persistent calls to have answer sheets for the book. Students like to practice on off-hours, often at home, and need answers to check their comprehension and progress.

Language independence It seems that using slightly abstracted pseudocode doesn't bother students when learning to trace code, for example when we leave out explicit block markers `{}` from the C language. However, in our Python course we saw that some aspects of our C-inspired syntax were indeed confusing.

Impact on teaching It appears that students learn the programming vocabulary better from the book than from our lectures and assignments. We have felt more comfortable asking students to "try to make a trace" when fazed by an annoying bug. And post-test discussions have allowed us to more systematically correct the use of the nonexistent word "`if`-loop".

4 FUTURE WORK

The book and tests are in active development and use, and we encourage teachers to take part in testing an researching the usefulness for goals like we set out in the introduction. Code, book chapters and various sample tests may be downloaded or contributed to at `http://stgm.nl/basics`.

ACKNOWLEDGMENTS

Many thanks to Jelle van Assema, Simon Pauw, Jill de Ron and Wouter Vrielink for contributing to the development of the book, tests and software, and to our many teaching assistants for contributing corrections.

REFERENCES

[1] Benedict Du Boulay. 1986. Some difficulties of learning to program. *Journal of Educational Computing Research* 2, 1 (1986), 57–73.
[2] Fred S. Keller and John G. Sherman. 1974. *PSI, the Keller Plan Handbook: Essays on a personalized system of instruction.* W.A. Benjamin, Menlo Park, CA, U.S.A.
[3] Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, et al. 2004. A multi-national study of reading and tracing skills in novice programmers. In *ACM SIGCSE Bulletin*, Vol. 36. ACM, 119–150.
[4] Raymond Lister, Colin Fidge, and Donna Teague. 2009. Further evidence of a relationship between explaining, tracing and writing skills in introductory programming. In *ACM SIGCSE Bulletin*, Vol. 41. ACM, 161–165.
[5] Mike Lopez, Jacqueline Whalley, Phil Robbins, and Raymond Lister. 2008. Relationships between reading, tracing and writing skills in introductory programming. In *Proceedings of the fourth international workshop on computing education research.* ACM, 101–112.
[6] Andrew Luxton-Reilly, Brett A. Becker, Yingjun Cao, Roger McDermott, Claudio Mirolo, Andreas Mühling, Andrew Petersen, Kate Sanders, Jacqueline Whalley, et al. 2018. Developing Assessments to Determine Mastery of Programming Fundamentals. In *Proceedings of the 2017 ITiCSE Conference Working Group Reports.* ACM, 47–69.
[7] Greg L. Nelson, Benjamin Xie, and Amy J. Ko. 2017. Comprehension first: evaluating a novel pedagogy and tutoring system for program tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research.* ACM, 2–11.
[8] Juha Sorva. 2013. Notional machines and introductory programming education. *ACM Transactions on Computing Education (TOCE)* 13, 2 (2013), 8:1–8:31.
[9] John Sweller. 2016. Story of a research program. *Education Review/Reseñas Educativas* 23 (2016).
[10] Vesa Vainio and Jorma Sajaniemi. 2007. Factors in novice programmers' poor tracing skills. In *ACM SIGCSE Bulletin*, Vol. 39. ACM, 236–240.
[11] Jan van de Craats and Rob Bosch. 2014. *All You Need in Maths!* Pearson Education Benelux, Amsterdam, Netherlands.
[12] Benjamin Xie, Greg L. Nelson, and Amy J. Ko. 2018. An Explicit Strategy to Scaffold Novice Program Tracing. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education (SIGCSE '18).* ACM, 344–349.