VisAST: Generic AST Visualiser for Software Language Education

Ragnhild Aalvik*
Bekk Consulting, Norway

Jaakko Järvi†
University of Bergen, Norway

Abstract

Structural concepts such as *abstract syntax trees* (ASTs) are often best explained through visual representations. Students may, however, struggle with connecting such visual representations with the corresponding program text. To bridge this gap, we developed visAST, a tool for easily visualising ASTs of small languages written in Haskell. To assess the benefits and usability of visAST we conducted a user study in the context of students implementing interpreters. Students reported liking visAST and it being beneficial for learning. The experiment's results were not conclusive, but hint at visAST use improving students' performance.

1 Introduction

This paper presents *visAST*, a tool for visualising *abstract syntax trees* (ASTs) of programs. The purpose of visAST is to help students understand how ASTs and expression evaluation work. Its development was motivated by observing that students of programming language classes often struggle with how the textual representation of a program corresponds to its representation as a syntax tree.

When learning about interpreters, type checkers, and other language processors, students are often presented with ASTs as the internal representations of programs in visual form on the blackboard. When programming assignments on their own, however, they work with textual representations of these abstractions. Our conjecture is that a dynamic visual view of a program's AST helps the students see the connection between the textual and tree representation of a program, and help students learn essential aspects about interpreters and expression evaluation.

It would be unrealistic to expect that students, while learning about ASTs, interpreters, etc., simultaneously implement graphical views of their programs' internal representations. To be practical, such study aids must be at the student's disposal with minimal effort and complexity. VisAST adapts to any (Haskell) data type that represents ASTs, typically without students having to write any additional code for visualisation. To achieve this, we take advantage of Haskell's data type generic programming features.

We investigated the impact of visAST on students' learning and how students perceive its use in an experiment conducted as part of a class on Functional

*ragnhild.aalvik@bekk.no
†jaakko.jarvi@ii.uib.no

This paper was presented at the UDIT-2019 conference; see http://www.nik.no/.

Programming. Students' view of visAST was favorable, and the results of the experiment hint at visAST impacting students' performance positively.

2 Background and related work

Learning styles can be categorized to visual, auditory, and kinesthetic [1]. Visual learners remember best information that they can see, e.g., in pictures, animations, graphs, or films. Auditory learners remember best what they can hear, and they learn better from verbal, rather than visual explanations. Kinesthetic learners learn best through touch, taste, and smell.

Most students are visual learners [2, 3], yet most teaching is verbal. In a traditional lecture, the instructor talks to the class, often also writing textual content on the blackboard, which our brain processes similarly to spoken words [2]. Traditional teaching seems thus most beneficial for auditory learners.

Of course instructors also show visualisations in class, drawings or graphs on the blackboard, videos or animations, and some also using visualisation tools. Visualisation tools for teaching sorting algorithms, curiously, seem to be especially popular [4, 5, 6, 7]. This may be because sorting is somehow a concept that is easy to visualise or because as a topic it is a stable in the curriculum, justifying the effort of developing visualisation tools. There are also many tools for drawing graphs online [8, 9, 10], but such tools seem not to be widely used in teaching.

In the context of programming languages, popular languages have good tools for visualising program executions, but their use in teaching is not mainstream. One example tool is Online Python Tutor [11] (which is not limited to Python). It is essentially a debugger targeted for teaching and collaborative studying. Users can step forwards and backwards in an execution and observe the program state. Another example is the Ohm online editor [12] that can visualise expressions from a user-specified grammar. ExtendJ Explorer [13] lets users explore ASTs built by ExtendJ, relating the ASTs to the Java source code they are generated from.

Multiple studies report students in introductory programming courses liking visualisation tools. VIP, a visual interpreter for C++, was overall rated positively by students [14]. Feedback from students on VIPER, a tool for visualising Pascal code, was positive, particularly among beginners [15]. Daly [16] studied the opinions and impact of Alice, which visualises concepts like loops, objects, and recursion via 3D animations. She reported students liking using Alice and achieving better results and higher self-efficacy than students in a control group. Students with less prior programming experience found Alice significantly more useful than did students with more experience. Kaila et al. [17] investigated the use of the program visualisation tool ViLLE in an introductory programming course, and reported that novice programmers had the most positive views and largest increase in grades.

Ben-Bassat Levy et al. [18] reported a significant improvement in grades of students who used Jeliot, a tool for visualising Java code. In their study, average students benefited the most from the tool: better students did not need it and the weakest students were overwhelmed by it. This is an interesting contrast to our finding, reported in Section 4, that weaker students were the ones who used visAST the most and also perceived it the most useful. These differences might stem from the difficulty of the visualised concept and the usability of the visualisation tool.

Vegdahl [19] reported on using visualisations in a compiler course. Students could visualise various steps of the compilation process of Java programs, including

the AST. Students reported using the AST visualiser a lot, evaluating the tool from "very helpful" to "absolutely vital" for understanding and debugging their programs, and for gaining an intuition of how executable code relates to source code. This was also a motivation for our project, to help students relate their source code to the visual representations often presented in class.

Mernik and Zumer [20] investigated the impact of LISA that provides animated visualisations of different parts of a compiler. The students viewed the tool use positively; 76 % reported the tool as "important" in understanding compilers better.

Blythe et al. [21] developed LLparse and LRparse, two interactive tools for visualising examples of LL(1) and LR(1) parsing. Students of a programming course, with a focus on automata theory, used the tools for solving and error checking their assignments. Blythe et al. observed that students had fewer errors in their assignments after using the tools.

VAST is an educational tool for visualising *syntax trees* (STs), targeted for compiler or language processing classes. In an educational evaluation in a language processors class, the instructors observed that students were enthusiastic about VAST. Students perceived the tool as positive, easy to use, and that it supported them in their learning. [22]

Naser [23] compared students' exam results with the use of a visualisation tool in an AI searching algorithms course. Students who had used the tool performed better in the exams. Further, there was a strong correlation between students evaluating the visualisation tools as beneficial and their grade improvement in the exams.

Naps et al. [24] reviewed multiple experimental studies conducted on the effectiveness of visualisation tools. They divided the tools into *passive observation*, which includes observing different representations of the learning material through animations, and *active engagement* which includes learner involvement, e.g., by making the users construct their own data sets or program the algorithm to be visualised. This meta-analysis found that tools in the active engagement category were consistently more effective for learning than passive observation tools. Of the tools that required active engagement, 83 % produced significant positive results. Our visAST tool is clearly in the active engagement category: it visualises the evaluation steps of a program that a student writes.

In summary, past studies show that students like using visualisation tools, and that these tools can improve learning. Further, different types of students benefit differently from visualisations, but whether it is the stronger or weaker students that benefit the most varies.

3 visAST

VisAST is a web application, available at https://vis-ast.netlify.com. It can visualise ASTs of expressions generated by an arbitrary abstract grammar, and step through a sequence of ASTs, e.g., the evaluation steps produced by an interpreter.

VisAST offers two modes. The *Get familiar* mode visualises expressions of a simple, fixed, expression language. This mode is meant to make the tool easy to approach. It presents the user with a grammar and a field to type in expressions. VisAST parses the expression with a built-in parser, records all the evaluation steps of a built-in evaluator, draws the initial AST (see Figure 1), and lets the user step forwards and backwards through the steps. The interpreter implements small-step operational semantics, which makes the intermediate steps of expression evaluation

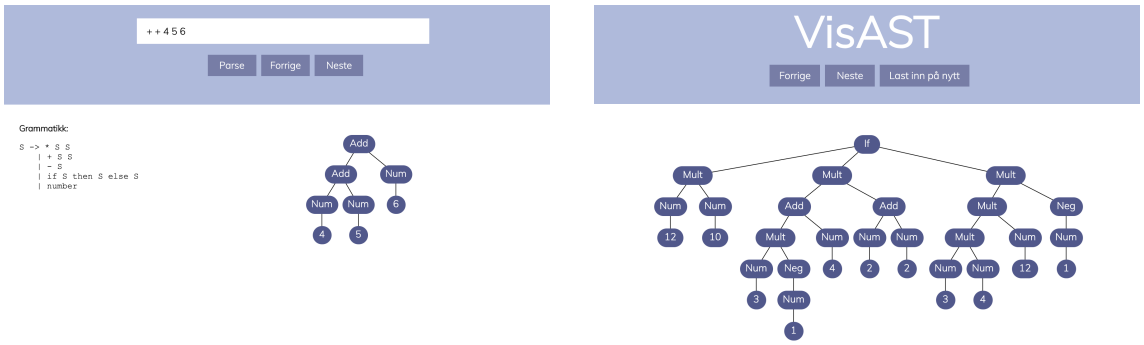


Figure 1: Two use modes of VisAST: *Get familiar* on the left shows the view of the initial AST after parsing an expression and *Advanced* on the right shows an AST saved and retrieved for a given lookup key.

explicit [25]. The *Advanced* mode lets the user visualise her own code. The user implements her own language, which means defining the abstract syntax and a small-step evaluator, and possibly a parser. VisAST's client library provides a function for sending the evaluation steps to the VisAST server, together with a lookup key that identifies the user. When the key is entered on the VisAST website, VisAST retrieves the ASTs saved for that key and visualises them as in the Get familiar mode. Figure 1 displays a stored and retrieved expression in the Advanced mode; no grammar is displayed because the grammar is defined in the user's source code.

The frontend of visAST is written in Elm and the backend server in Haskell. The server runs on Heroku and stores its data in a PostgreSQL database. The source code of the implementation is available as three GitHub repositories under https://github.com/aaalvik: visast-frontend and visast-backend are the frontend and backend implementations, visast-client-example implements a client that demonstrates how to use visAST in the Advanced mode.

The frontend speaks with the backend through a RESTful API. In the *Get familiar* mode, expressions typed in by the user are sent to the backend as a POST request, and parsed and evaluated on the server. As a response, the server sends back a list of ASTs. In the *Advanced* mode, the user code passes a list of ASTs to the visualise function defined in the VisAST client library. This function serializes the ASTs and sends them together with a user-defined key to be stored on the server, and opens the visAST web interface for the user.

Aalvik's thesis [26] gives a detailed description of the implementation and how a student uses visAST from her code. Here we show just enough code to demonstrate that visualising ASTs of a language that a student creates herself requires practically no programming. Listing 1 shows parts of an implementation of a simple language that a student might write. The data type Expr defines the abstract syntax. The parse function parses program text to ASTs of type Expr. The evaluate function then reduces the AST to a normal form, returning all reduction steps, as type [Expr]. These functions are written by the student, and not shown here. In order to visualise programs, nothing but a small amount of boilerplate is needed: 1. visAST client library must be imported (line 1); 2. the AST type must derive from Generic (line 3); 3. the AST type must be declared an instance of Generalise (line 4); and 4. parsed and evaluated ASTs must be sent to be visualised using the visualise function (line 5).

The user can invoke visualise with practically any AST types. Such generality is

```
import Client -- imports visualise, Generalise, etc.
data Expr = Num Int | Add Expr Expr | Mult Expr Expr | Neg Expr | If Expr Expr Expr
          deriving (Eq, Show, Generic)
instance Generalise Expr
main = visualise (evaluate (parse "(4 + 5) + 6"))
```

Listing 1: Using the visualise function to send ASTs to visAST.

attained with Haskell's Generics features that support writing algorithms over any type. The generics features serve for a similar purpose than Reflection in, say, Java. In particular, we inspect the user-defined AST data type to extract the names of each different AST node and the number of their arguments, and construct a general purpose AST representation that can be easily serialized and then visualised.

The trees are drawn by generating SVG, to be rendered by the browser. The generation is obtained with a top-down recursive traversal over the tree. For placing the nodes, visAST works hard to produce a visually pleasing outcome. The layout algorithm is described in Aalvik's thesis [26].

The reasons to implement the visualisation on the server, rather than entirely on the web client, was primarily to be able to collect data on how the students interact with visAST, but also to make the adoption as easy as possible and without the need to install additional software. VisAST records every request to the server. Concretely, it logs an event when a user parses an expression in the *Get familiar* mode, requests the ASTs of a specific lookup key in the Advanced mode, or invokes visualise. We could in principle also track when the user flips back and forth through the reduction steps, but we did not do that in our experiments.

4 Methods and results

In order to assess the usefulness of visAST, we conducted a user study. The subjects were students in the third semester undergraduate class on Functional Programming (INF122) at University of Bergen (UiB). The study took place in the context of an obligatory assignment, in which students implemented a simple expression language. They were asked to use visAST to visualise ASTs of their language, and later to fill out a questionnaire about how they perceived visAST.

We also wanted to measure whether the use of visAST had an impact on students' performance. Due to time constraints and the need to treat all students equally (the course is compulsory for most of the students), a controlled study was not possible—all students received the same assignment and the same instructions on using visAST. We can, however, draw some conclusions by comparing the students' answers in the questionnaire and their performance in the assignment.

The assignment

The focus of the assignment was parsing and evaluating expressions, which is one notable component of INF122. Students were given a grammar of a simple expression language consisting of arithmetic expressions formed with `*`, `+`, and unary $-$, along with if then else-expressions:

$\langle S \rangle ::=$ '`*`' $\langle S \rangle$ $\langle S \rangle$ | '`+`' $\langle S \rangle$ $\langle S \rangle$ | '`-`' $\langle S \rangle$ | '`if`' $\langle S \rangle$ '`then`' $\langle S \rangle$ '`else`' $\langle S \rangle$ | $\langle digit \rangle^+$

Based on this language description the students were asked to (1.1) implement a parser (program text → AST), (1.2) pretty-printer (AST → program text), and (1.3) small-step evaluator for the language; (1.4) write a function that can run user commands, such as running the pretty printer; (2.1) implement the command for visualising ASTs using visAST; (2.2) fill out a questionnaire about visAST; and (3.1) a task unrelated to visAST. Before starting, students were instructed to download the visAST client and check that it worked on their machine.

As a voluntary subtask of 1.1, students were asked to test their parser using visAST, concretely to call `visualise` with their parsed AST. In Task 1.3, students could use visAST to show their interpreter's intermediate ASTs. Task 1.4 was to create the command that launched the visualisation, essentially to write the code that passes the list of ASTs to `visualise`. The questionnaire in Task 2.2 was given out as an online form; the questions are described below.

Participants and procedures

The subjects were all students in the INF122 class. There were originally 214 students signed up, 93 turned in the assignment, and 79 answered the questionnaire. 48 students answered Question 8 in the questionnaire; this question asked for free-form feedback. The questionnaire was part of the mandatory assignment, but students were informed that it had no effect on the assignment grade. The assignment was to be completed individually and handed in in two weeks' time; some collaboration was allowed, except for in the questionnaire.

Standard curricula at UiB do not include learning about parsing and interpreters prior to INF122. Most students thus knew little about these topics before this course, but during the course they had already been discussed to an extent. None of the participants had used visAST before this experiment.

The questionnaire was in Norwegian. Translated to English, the questions were:

- Q2 How well did you understand ASTs before this assignment? (1=understood little, 5=good understanding)
- Q3 How much did you use visAST in this assignment? (1=as little as possible, 5=a lot)
- Q4 How useful was visAST for your understanding of ASTs in this assignment? (1=not useful, 5=very useful)
- Q5 How useful do you think visAST would have been for your understanding the first time you were introduced to ASTs? (1=not useful, 5=very useful)
- Q6 Did visAST help you understand parsing better? (yes/no)
- Q7 Did visAST help you understand evaluation of expressions better? (yes/no)
- Q8 We would like your thoughts about visAST, in your own words. What was useful/not useful? What was easy/not easy? Any suggestions for improvements? Other thoughts?

Q1 was for identifying information we needed for aligning students' answers, visAST usage logs, and assignment and exam grades.

Research questions

To understand benefits of visualisation tools for learning about language processors, we were interested in how students evaluate their knowledge of ASTs and related topics, and their experiences with visAST. We formulated two research questions for the experiment: 1. *Do students find visAST helpful in understanding ASTs*

better? 2. *Do students find visAST helpful in understanding parsing and expression evaluation better?*

Q2, Q3, and Q4 deal with the students' previous understanding and how much they used the tool in the experiment. These are all relevant for answering the first research question. Q2, Q3, Q6, and Q7 are relevant for answering the second research question. Q6 and Q7 address it directly, and Q3 is also relevant for it has to do with how much the subjects used the tool. Q2 is relevant because an understanding of ASTs is a prerequisite to understand parsing and evaluation.

Q5 was included because of the timing of the experiment: students had already learned some about ASTs in the class, possibly rendering visAST less useful.

The purpose of Question 8 was to get feedback for improving visAST.

Results

We analysed the results first for all students, then divided into three groups based on their final exam score, graded from 0 to 105 points, as follows: *Stronger students* (80 points or more), *Average students* (50–80 points), and *Weaker students* (less than 50 points). The stronger group had 24 students, average 30, and weaker 15, amounting to 69 students in total. We excluded from the groups ten students who did not take the final exam. The reason for the grouping was to investigate how different types of students experienced visAST and how much they benefited from it. As discussed in Section 2, many studies show that visualisation tools benefit different types of students differently.

Questions Q2 through Q5 had an ordinal scale with labels on the lower (1) and upper (5) bound of the scale. To compute averages of this ordinal scale, we converted it to its numerical equivalent. In general, one should be careful when converting an ordinal scale to numerical [27]. Our data is clearly single peaked and we have no basis for any other assumption than that the data is approximately normally distributed. Under these assumptions, the risk of misanalysis is low [27].

Figure 2 shows the distributions of the answers to the questions Q2–Q5, jointly for all students and separately for each group, as bar charts. It also lists the averages in the numeric scale and standard deviation (for the "all students" case).

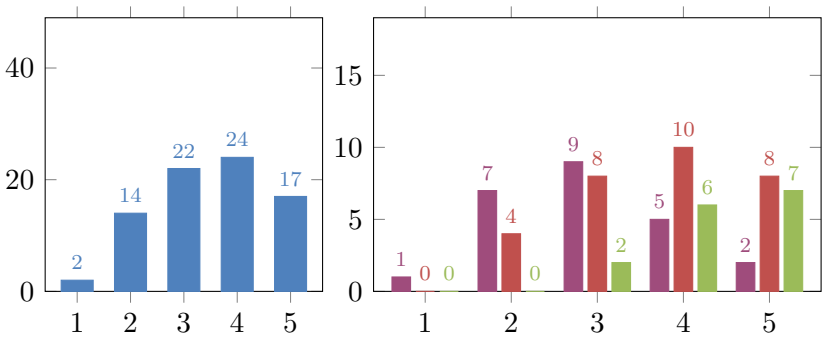
The mean of 3.5 of answers to Q2 with standard deviation 1.1 suggests a varying but moderately good understanding of ASTs prior to the assignment. There were sixteen students who answered that they had very little or little understanding, all of which belonged in the weaker or average group. Even the weaker group, however, had a mean of 3.0, indicating a moderately good understanding on average.

In Q3, no student reported using visAST "a lot" in the assignment, but that a substantial number of students reported moderate or more use. The weaker group used the tool somewhat more than the average and stronger groups.

In Q4, a majority (58%) of all students found visAST moderately or more useful for learning about ASTs, confirming our first research question above. The weaker students found visAST the most helpful—not a single student in this group answered "not useful". On average the stronger group found the tool the least useful, but there were many in that group too who found visAST either "useful" or "very useful".

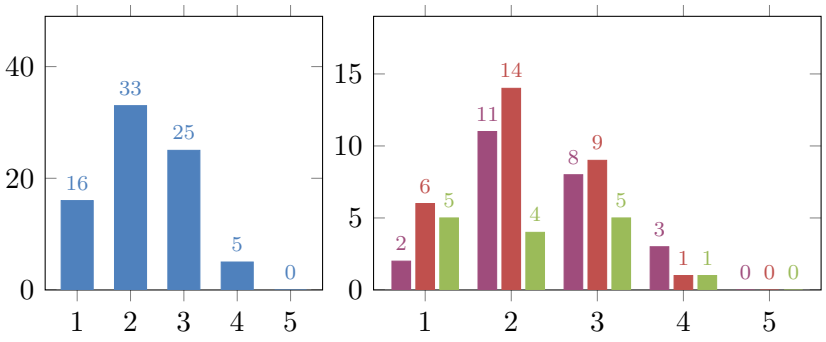
Answers to Q5 confirm our suspicion that we introduced visAST too late in the topic sequence of the class. All groups overwhelmingly agreed with this: 54.4% of the the students answered that visAST would have been "very useful" if introduced when ASTs were first introduced, and no students answered "not useful".

μ_{all} 3.5
s_{all} 1.1
μ_{weak} 3.0
μ_{avg} 3.7
μ_{strong} 4.3



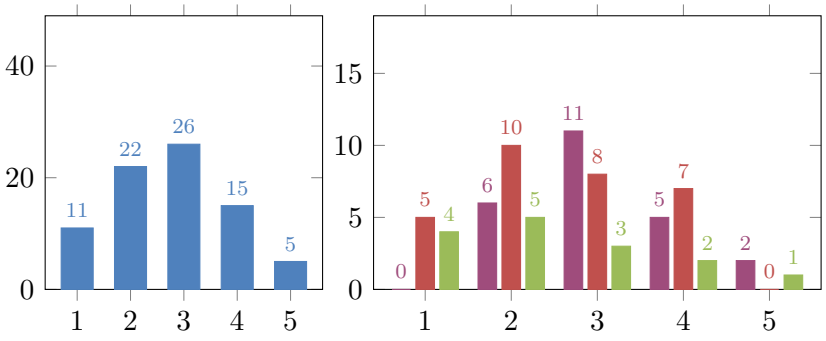
Q2: How well did you understand ASTs before this assignment? 1=understood little, 5=good understanding.

μ_{all} 2.2
s_{all} 0.9
μ_{weak} 2.5
μ_{avg} 2.2
μ_{strong} 2.1



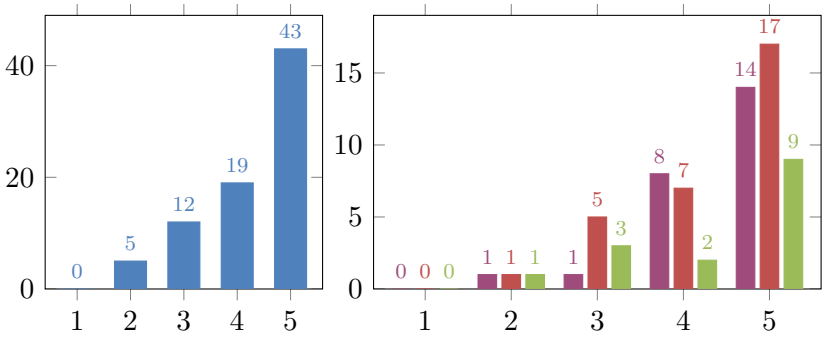
Q3: How much did you use visAST in this assignment? 1=as little as possible, 5=a lot.

μ_{all} 2.8
s_{all} 1.1
μ_{weak} 3.1
μ_{avg} 2.6
μ_{strong} 2.4



Q4: How useful was visAST for your understanding of ASTs in this assignment? 1=not useful, 5=very useful.

μ_{all} 4.3
s_{all} 0.9
μ_{weak} 4.5
μ_{avg} 4.3
μ_{strong} 4.3



Q5: How useful do you think visAST would have been for your understanding the first time you were introduced to ASTs? 1=not useful, 5=very useful.

Figure 2: Questions Q2–Q5. The leftmost charts show the distribution for all students and the rightmost for each group according to weak average strong. The x-axis is the number of answers, y-axis their ordinal scale. The columns on the left list the averages μ, and the standard deviation s for all students.

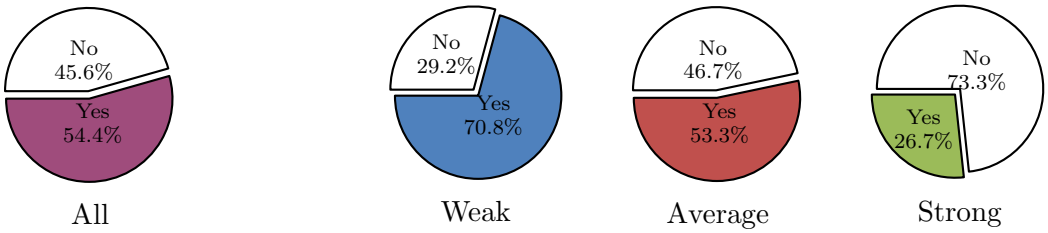


Figure 3: Q6: Did visAST help you understand parsing better?

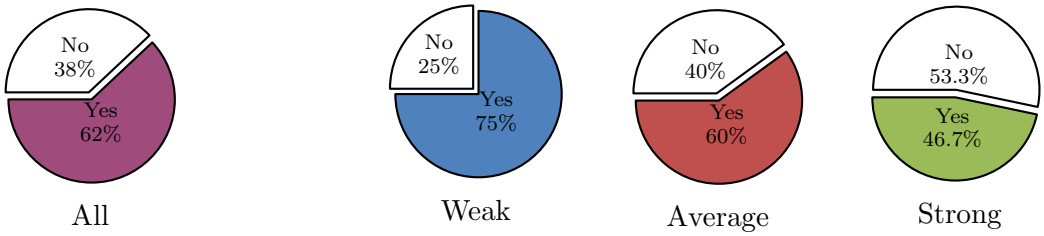


Figure 4: Q7: Did visAST help you understand evaluation of expressions better?

Figure 3 shows that a majority thought that visAST helped them understand parsing better, which answers part of our second research question above. However, quite a few students did not agree. The difference between groups was rather stark: only 27 % of the stronger students agreed, compared with 71% of the weaker ones.

Figure 4 shows that students found visAST even more helpful for understanding expression evaluation: a larger majority, 62 %, agreed with Q7, which answers the second half of our second research question. Again, of the three groups the weaker students found visAST most helpful, with 75 % agreeing, but almost half (47 %) of the stronger students agreed too.

Of the answers to the free-form question Q8, we identified four common themes. First, several students (28) commented on the timing of when the tool was introduced, reiterating that it would have been more useful earlier in the class. Second, students commented on the tool's usability positively, finding the design simple and vsualisations neat. Three students mentioned that visualising their own code helped them verify its correctness. Third, several students had good suggestions for future features or improvements for visAST, such as animating transitions between the trees to help focus on what changed in a reduction step. Fourth, there were a handful of negative responses, practically all about the installation of external Haskell libraries that visAST's Advanced mode requires.

In addition to gathering students' thoughts on visAST, we wanted to measure whether the use of visAST had an impact on students' learning. Concretely, we formulated an additional hypothesis H_a: *the use of visAST improved students' performance in assignment A2.*

As discussed in Section 4, we had no control group—all students were instructed to use visAST. We therefore used the students' performance in other assignments/exams as a reference point, as the expected performance in assignment A2. Prior to any comparison, we normalised the scores to standard normal distribution. We compared the score in A2, in which visAST was used, to this reference point and correlated the difference in scores to how much the students used visAST in A2. A positive correlation of the difference with how much visAST

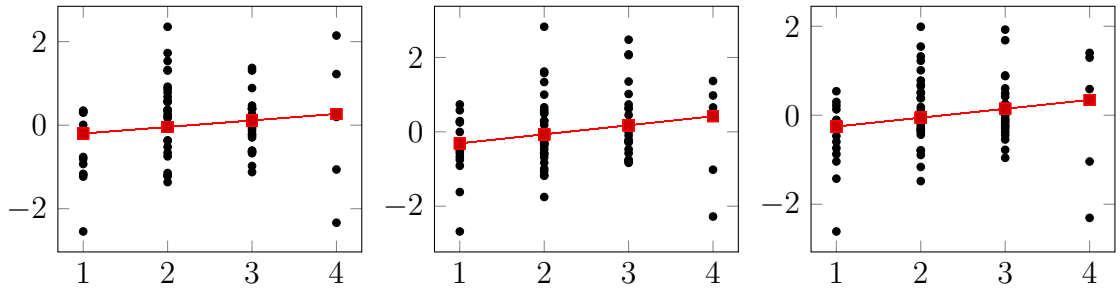


Figure 5: The difference of A2 normalized score and the reference score, A2-A1 on the left, A2-Exam in the middle, and A2-Rest on the right. The x-axis in all charts is from 1–4, since no student answered 5. The red lines show the linear regressions.

	Estimate	Std. error	t-value	$\Pr(>\|t\|)$
Δ_{A2-A1} (intercept)	-0.3566	0.3262	-1.093	0.278
visAST-use (slope)	0.1567	0.1343	1.167	0.247
$\Delta_{A2-Exam}$ (intercept)	-0.5520	0.3530	-1.564	0.1226
visAST-use (slope)	0.2426	0.1454	1.669	0.0998
$\Delta_{A2-Rest}$ (intercept)	-0.4543	0.3018	-1.505	0.1370
visAST-use (slope)	0.1997	0.1243	1.607	0.1130

Figure 6: Regressions between the difference in score and the use of visAST.

was used supports our hypothesis. We performed this analysis three times: A2 against the assignment A1, A2 against the exam, and A2 against the combined score (using equal weights) of A1 and the exam; we call these cases *A2-A1*, *A2-Exam*, and *A2-Rest*. Assignment A1 preceded A2, so visAST use did not effect A1 scores. The exam came after A2 and thus its score could in theory have been impacted by visAST use. No exam question, however, was directly related to ASTs, parsing, or evaluation and we thus expect any effect to be negligible.

For the A2-A1 comparison we discarded three students, A2-Exam four, and A2-Rest seven students who did not submit A1 or take the Exam. Figure 5 shows the difference in the normalised score between A2 and, respectively, A1, Exam, and the combined A1/Exam score, compared to how much the student used visAST in A2.

A summary of the regression analyses is shown in Figure 6. In the A2-A1 case, correlation with the score difference and visAST use was weak, 0.1411, and the p-values for the coefficients high, 0.278 and 0.247: we cannot reject the null hypothesis of no correlation. In the A2-Exam case, correlation was 0.1998; generally 0.2 is considered a threshold between weak and moderate positive correlation. The p-values for the coefficients (0.1226 and 0.0998) were smaller but do not indicate statistical significance. In the A2-Rest case, correlation was 0.1926 which is slightly lower than the A2-Exam case. The p-values were 0.1370 and 0.1130, which do not indicate statistical significance either. We can at best interpret the results as hinting towards that more use of visAST correlates with a higher A2 score.

5 Discussion and conclusion

Results from the questionnaire showed that students liked to use visAST as a learning tool. Part of the positive attitude may stem from the good usability of

visAST; we put a lot of effort to node layout for a visually pleasing outcome, and for making it possible to create visualisations with ease, directly from the students' program code. As discussed in Section 2, several other studies have reported students to have positive views toward using visualisation tools.

A notable characteristic of VisAST is that it can visualise ASTs of any language that a student implements, without requiring the student to write (practically) any code for the visualisation. Though we omitted many details of the implementation, it bears mentioning that Haskell's data-type generic features were key in our solution. We suspect that with some effort similar ease of use could be achieved in other languages, e.g., in Java with the help of the reflection features or in JavaScript by inspecting objects' properties dynamically.

Besides usability, students evaluated visAST's usefulness for learning. The majority considered visAST to be useful for understanding ASTs, parsing, and expression evaluation. These views were most dominant in the weaker student group; it was also this group that reported the highest use of visAST.

The regression analyses we performed hint towards that visAST can improve learning results, but we say this with reservations. More accurate results could be obtained with a better experiment design. As we had no control group, we used students' performance in other tasks as controls, which we expected to be noisy. Ideally, we would also control for the time invested on the assignment as assignment grades can correlate with effort, not only with knowledge, and more use of a tool may also mean more time invested on an assignment.

In sum, our work adds to the evidence that active engagement visualisation tools, which visAST is, are beneficial for learning programming concepts.

References

[1] Walter Burke Barbe, Michael N Milone, and Raymond H Swassing. *Teaching through modality strengths: Concepts and practices*. Zaner-Bloser, 1988.

[2] R. J. Kapadia. Teaching and learning styles in engineering education. In *38th Annual Frontiers in Education Conference*, pages T4B–1–T4B–4, Oct 2008.

[3] G. Gray and D. Duffin. SIF: Learning styles theme final report: Project report 2007-2009. Unpublished report, obtained by request from author, 2011.

[4] Steven Halim. Visualgo–visualising data structures and algorithms through animation. *Olympiads in Informatics*, page 243, 2015.

[5] Kanasz Robert. Visualization and comparison of sorting algorithms in C#. www.codeproject.com/Articles/132757/Visualization-and-Comparison-of-sorting-algorithms, December 2010. Accessed: 2019-05-02.

[6] Callum Macrae. Sorting algorithms visualised. http://macr.ae/article/sorting-algorithms.html, December 2016. Accessed: 2019-05-02.

[7] Carlo Zapponi. Sorting. http://sorting.at/, 2014. Accessed: 2019-05-02.

[8] Graph editor. https://csacademy.com/app/graph_editor/. Accessed: 2019-05-02.

[9] Graph online. https://graphonline.ru/en/, 2015. Accessed: 2019-05-02.

[10] Graphtea. http://www.graphtheorysoftware.com/. Accessed: 2019-05-02.

[11] Philip J. Guo. Online Python Tutor: Embeddable Web-based program visualization for CS education. In *Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. ACM.

[12] Patrick Dubroy, Saketh Kasibatla, Meixian Li, Marko Röder, and Alex Warth. Language hacking in a live programming environment. In *LIVE 2016 at ECOOP 2016*, USA, 2016. https://ohmlang.github.io/pubs/live2016/.

[13] Torbjörn Ekman. ExtendJ Exploring Tool. https://extendj.org/visualizer.html. Accessed: 2019-03-12.

[14] A.T. Virtanen, E. Lahtinen, and H.-M. Järvinen. VIP, a visual interpreter for learning introductory programming with C++. In *Kolin Kolistelut—Koli Calling 2005 Conference on Computer Science Education*, pages 125–130, 2005.

[15] Michał Adamaszek, Piotr Chrzastowski-Wachtel, and Anna Niewiarowska. VIPER, a student-friendly visual interpreter of pascal. In Roland T. Mittermeir and Maciej M. Sysło, editors, *Informatics Education—Supporting Computational Thinking*, pages 192–203. Springer Berlin Heidelberg, 2008.

[16] Tebring Daly. *Influence of Alice 3: Reducing the Hurdles to Success in a Cs1 Programming Course*. CreateSpace Independent Publishing Platform, 2013.

[17] E. Kaila, T. Rajala, M.-J. Laakso, and T. Salakoski. Effects, experiences and feedback from studies of a program visualization tool. *Informatics in Education*, 8:17–34, 2009.

[18] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka Uronen. The Jeliot 2000 program animation system. *Computers & Education*, 40:1–15, 01 2003.

[19] Steven R. Vegdahl. Using visualization tools to teach compiler design. In *Proceedings of the Fourteenth Annual Consortium on Small Colleges Southeastern Conference*, CCSC '00, pages 72–83, USA, 2000. Consortium for Computing Sciences in Colleges.

[20] M. Mernik and V. Zumer. An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68, Feb 2003.

[21] Stephen A. Blythe, Michael C. James, and Susan H. Rodger. LLparse and LRparse: Visual and interactive tools for parsing. *SIGCSE Bull.*, 26(1):208–212, March 1994.

[22] F. J. Almeida-Martinez and J. Urquiza-Fuentes. Syntax trees visualization in language processing courses. In *2009 Ninth IEEE International Conference on Advanced Learning Technologies*, pages 597–601, July 2009.

[23] Samy S. Abu Naser. Developing visualization tool for teaching AI searching algorithms. *Information Technology Journal*, 7(2):350–355, 2008.

[24] Thomas L. Naps, Guido Rößling, Vicki Almstrum, Wanda Dann, Rudolf Fleischer, Chris Hundhausen, Ari Korhonen, Lauri Malmi, Myles McNally, Susan Rodger, and J. Ángel Velázquez-Iturbide. Exploring the role of visualization and engagement in computer science education. *SIGCSE Bull.*, 35(2):131–152, jun 2002.

[25] Gordon D. Plotkin. A structural approach to operational semantics. *J. Log. Algebr. Program.*, 60-61:17–139, 2004.

[26] Ragnhild Aalvik. VisAST: Generic AST visualiser for software language education. Master's thesis, The University of Bergen, 2019. http://hdl.handle.net/1956/19781.

[27] Barbara A. Kitchenham and Shari L. Pfleeger. Personal opinion surveys. In *Guide to Advanced Empirical Software Engineering*, chapter 3, page 90. Springer-Verlag London, London, 2008.