University of Bergen
Department of Informatics

VisAST: Generic AST Visualiser for Software Language Education

Author: Ragnhild Aalvik
Supervisor: Jaakko Järvi

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May, 2019

Abstract

Structural concepts, like *abstract syntax trees* (ASTs), are often best explained through visual representations. Students seem to have little trouble understanding what is presented to them visually, but they find it harder to translate their visual conception into source code when programming assignments on their own. Few resources are available to help students make this connection between visual and textual representations.

We developed a tool, visAST, for dynamically visualising ASTs of small languages written in Haskell, to help students connect the visual representations of ASTs to their own source code. The goal was to make the visualisations of visAST effortless to adopt for any new language that the students define. To assess the benefits and usability of visAST we conducted a user study, where visAST was used while implementing a simple interpreter. We asked students about their opinions on the tool and measured their performance with and without visAST. Our results show that students like visAST and find it useful. The results also suggest that visAST slightly improved students' performance in a programming class.

Acknowledgements

First of all, I would like to thank my supervisor, Jaakko Järvi, for all your insight and inspiration throughout this thesis. I truly appreciate how you always have time for my questions, and especially your interest in grammar rules and spelling. I would also like to thank Anya Helene Bagge, who inspired me to pursue this project in the first place.

I am grateful to all those who have encouraged me during this thesis. My fellow students have been a great support, especially the students at JAFU. A special thanks to Kristian and Kaja for always being there and motivating me during my whole time as a student. I also want to thank Ingrid Kyllingmark, from the department administration, for feeding us leftovers every week.

Lastly, I want to mention my family, who has always encouraged me to pursue my dreams.

`Ragnhild Aalvik`

13 May, 2019

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

This thesis is about a visualisation tool called visAST that we developed for use in computer science education. The tool is used to visualise *abstract syntax trees* (ASTs) generated from programs either typed directly into the browser by the user or generated programmatically in the tool's *Advanced mode*. The purpose of visAST is to help students gain an understanding of how ASTs and expression evaluation work, and how the visual representation relates to the source code. Students tend to struggle with the intuition of relating what was visually presented in class to the textual representation they work with, e.g., in assignments.

The thesis reports on our experiment that investigated the impact of visAST on students' learning, and how students perceive the use of visAST. The experiment was conducted on students in a functional programming class who were instructed to use visAST as part of an assignment. After completing the assignment, the students answered a questionnaire about visAST. Overall, the students reviewed visAST favourably, and the students' performance in the class before and after using the tool slightly improved.

1.1 Visual learning in computer science

The visual learning style is one of the three categories often used to divide the different ways of receiving information. The other two categories are the auditory and kinesthetic learning styles. Visual learners are the ones who best remember 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 if something is explained to them verbally rather than visually. Kinesthetic learners learn best through touch, taste, and smell. Most students are visual learners [22, 18], yet most teaching is verbal. Since the majority of students benefit the most from visual input, we look at how today's education meets this preference, before we dive further into the world of visualisation tools used in teaching.

Even though most students would benefit more from visual input, traditional lectures consist mostly of the instructor talking to the class, often also using the blackboard to write textual content. One might think that written text on the blackboard is perceived as visual input because we read it with our eyes, but research has shown that our brain processes the written text as if it were spoken words [22], making traditional teaching most beneficial for auditory learners. Of course, many instructors also show visualisations in class, either by drawings or graphs on the blackboard or through videos or animations.

Visualisation tools exist in all kinds of areas, and in Chapter 2 we discuss several such tools and the results of using them in teaching. Typically, visualisation tools are used to create a visual representation of some abstract concept, to give the learner something concrete to relate to. From personal experience as a visual learner, I tend to visualise in my own head, or draw on paper, concepts that are explained to me verbally. If I am not able to visualise new information in some way, there is a high chance I will not understand or remember the information. This is a strong motivation for me to study and further develop visualisations for teaching. Results from prior research summarised in Chapter 2 show that students tend to like visual tools, which agrees with most of them being visual learners.

In computer science education, visualisation tools are used in teaching to varying degrees. A bit curiously, visualisation tools for teaching sorting algorithms seem to be especially popular. There are numerous tools designed for visualising different sorting algorithms [20, 31, 24, 34], and as far as we can tell these are designed to be used as part of the teaching material in algorithms courses. This might be because sorting algorithms are easy to explain using visual representations, and also because sorting is a concept that is easy to visualise. Other concepts in computer science could also be easily explained visually, and a natural question is whether these areas could also benefit from more visualisation tools. Graphs in algorithms classes are usually presented as drawings of nodes and edges, but typically these visualisations take place on the blackboard. Tools for drawing graphs are available online [2,

6, 3], but are not widely used by instructors in class. Popular programming languages like Java and Python have some good tools for visualising what happens during a program execution. Tools like Online Python Tutor [19] visualises the execution of programs written in various different languages, not limited to Python. Online Python Tutor is essentially a debugger, but meant for teaching. The user can step forwards and backwards in the execution of a program, line by line, to see the state of the objects and stack frames etc. at each execution point. This and other tools are meant to be used when learning to program or when learning a specific programming language.

Visual interfaces, tools, and languages are in heavy use when teaching programming to children. Nowadays, many children are introduced to programming early, predominantly through visual interfaces, languages, or libraries. Being exposed to technology from a young age, children are used to manipulating objects on a screen from playing games or using different apps. Programming in block-based languages typically happens through graphical manipulation, e.g., by dragging and dropping if statements or loops, which is quite similar to playing a game. One example of such a library is Blockly [28], which facilitates making your own block language for this purpose. Another example is Scratch [5], a block-based programming language targeted primarily for children. Multiple projects for teaching coding to children have had great success using such visual tools and languages, like Code Club [1] and Kids & Code [4].

1.2 AST visualisations for software language education

In software language education, when learning about interpreters, type checking, and other language processors, students are often presented with ASTs and other 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. Based on our experience, most students have little trouble understanding even complicated language processing operations when working with visual representations. On the other hand, many seem to struggle to get the same understanding when they work only with the textual representation, as in their assignments. As a previous teaching assistant, I have myself explained concepts like ASTs or grammars, visually, by drawing the syntax trees or the productions of the grammar on the blackboard. A common experience of mine is that a student understands the drawn concepts, but then has trouble translating the drawings into code. Another

example of confusion between what is presented in class and how it is represented in code is grammars versus ASTs. Many students find it hard to understand how a concrete grammar is not something they can write directly into their code, and there is often confusion about how a grammar specification corresponds to a datatype representing an AST.

We conjecture that a visual, dynamic view on a program's internal representation would help the students see the connection between the program text and what it represents. Since ASTs are one of the building blocks for working with language processors, it is crucial that the students get comfortable with ASTs at an early point. Dynamic visualisations, where the students can step through a sequence of ASTs representing the different consecutive states of a program execution, would give the students hands-on experience and the intuition they need to get comfortable with ASTs.

It would be unrealistic to expect that students, when learning about ASTs, interpreters, and other language processing tasks, simultaneously implement graphical views of their programs. To be practical, such study aids must be at the student's disposal with minimal effort and complexity. We are not aware of visualisation tools today that meet this need with respect to ASTs, and we argue that this is a gap that needs to be filled.

1.3 Connecting visual representations to source code

The first version of visAST was built as part of a project in a language processors class. The tool was further developed in this thesis into what we call visAST, with the purpose of creating a teaching tool for more inexperienced students. This new version of visAST got good initial feedback from fellow students, who liked the visualisations and were positive to the idea of using the tool as part of the teaching material in relevant classes. This was the motivation for doing an experiment on students of a functional programming class, to assess the usefulness of the tool in a real setting. Our hypothesis was: *visAST helps students understand ASTs better.*

Providing students of software language education with a tool like visAST fills a void in teaching as it is today. The dynamic visualisations provided by visAST aid in helping students make connections between source code and a program's internal representation. Using visAST the students can experiment with expressions and ASTs themselves, to fully

gain the intuition they need. Requiring little to no installation, visAST is a tool that is easy to adopt in education.

The experiment reported in this thesis was conducted as part of a class in which I was a teaching assistant, and I was given permission to prepare a compulsory assignment with tasks involving the use of visAST. The assignment included tasks about parsing and evaluation of expressions, which naturally include working with ASTs. The students were instructed to use visAST to solve some of the tasks, and upon finishing the assignment they were asked to answer a questionnaire about visAST. The questionnaire asked about how the students liked the tool, and if they thought it helped with their understanding of ASTs, parsing, or expression evaluation. Results showed that the students overall liked the tool and found it useful, but that it probably would have been even more useful earlier in the course when they knew less about ASTs. A performance analysis indicatively showed that students did better with visAST than without.

Chapter 2

Related work

This chapter surveys several studies on the effect visualisation tools used in teaching have on learning. The contexts of the studies vary from introductory programming classes, AI classes, and programming languages classes. In summary, these studies show that students generally like using visualisation tools. Many studies also show that different types of students benefit differently from visualisations, but whether it is the stronger or weaker students that benefit the most varies. We also look at tools whose effects on learning have not been studied, but that are otherwise close to our work in that they visualise syntax trees.

Few works report statistically significant results on the impact of visualisation tools with regards to students' performance in classes where such tools are used. Most research shows that in general visual tools have some positive effect, but that the results are only indicative. We discuss some of these studies in this chapter. Further, we discuss a meta-analysis that collected results from various research done on visualisation tools used in teaching, to understand what type of tool could be most beneficial to the students. This study divided visualisation tools into two categories based on the level of user involvement and concluded that tools requiring active user engagement, e.g., to program the algorithm to be visualised, are far more effective than tools where the user only observes a visual representation.

Lastly, we look into research on students' learning styles, which shows that the majority of students learn best through visual input.

2.1 Visualisation tools in introductory programming courses

Multiple studies have shown that students in introductory programming courses overall like visualisation tools [33, 10, 14]. Three of the studies we looked at found that students with little prior programming experience have especially favourable views of visualisation tools [10, 14, 21].

VIP, a visual interpreter used in teaching introductory programming in C++, was overall rated positively by students [33]. Another tool used in an introductory programming course is VIPER, a tool for visualising Pascal code, created by Adamaszek et al. [10]. Student feedback on the usability of the tool was positive, especially from the beginners among the students. They also reported that it was especially useful to be able to see how their program worked, step by step.

Daly [14] described a controlled experiment conducted in an introductory programming course on two different, comparable, campuses using a 3D animations tool called Alice. Alice is a pedagogical tool that includes introductions to fundamental programming concepts such as objects, loops, and recursion. Daly found that the students who used the tool reported having liked it and that they on average achieved better results and higher self-efficacy than the control group. Results also showed that the students with less prior programming experience found Alice significantly more useful and intuitive than the students with more experience.

The last of the three studies mentioned above investigated the use of the tool ViLLE in an introductory programming course. ViLLE is a language-independent visualisation system with a built-in editor. Kaila et al. [21] concluded, based both on observing a larger increase in grades for novice students and student feedback, that the tool was most useful for novice programmers.

One study by Ben-Bassat Levy et al. [12] showed a significant improvement in grades of students who used Jeliot, a tool for visualising Java code. They reported that average students benefited the most from the tool, especially in the long term. Further, they found that better students did not need the tool, whereas the weakest students were overwhelmed

by it. This is an interesting contrast to our findings, reported in Section 4.2.4, that weaker students were the ones who used the tool the most, and also reported the tool the most useful. These differences might be dependent on the difficulty of the topic covered by the visualisations, and the ease of use of the visualisation tool.

2.2 Visualisation tools in programming languages courses

As in introductory programming, also in *programming languages* (PL) related classes, visualisations have been reported useful. We discuss two studies on visualisation tools used in compiler courses that both received good feedback from the students.

In the first study, Vegdahl [32] reported their experience of using visualisations in a compiler course. The students were subjected to the use of a pair of Java packages to visualise various steps of the compilation process. One of the tool's features was visualising the AST created by the compiler, a feature the students reported using a lot. Reviews from the students on the tool ranged from "very helpful" to "absolutely vital" for understanding and debugging their programs, and for gaining an intuition of how the executable code related to the source program. This last part was also a motivation for our project, to help students relate their source code to the visual representations often presented in class.

In the second study, Mernik and Zumer [25] investigated the impact of the interactive tool LISA. LISA provides animated visualisations of different parts of a compiler, such as finite state automata, various types of parsers, and evaluation strategies. The students generally liked the tool, and 76 % of them reported the tool as "important" in understanding compilers better. That so many students ranked the tool as important is a strong indication of the usefulness of the visualisations, especially since the answer choices included also "helpful".

Other visualisation tools aimed at teaching PL course material are LLparse/LRparse and VAST. Blythe et al. [13] developed LLparse and LRparse, two interactive tools for visualising examples of LL(1) and LR(1) parsing. The tools were used in a programming course with a focus on automata theory, and the students used the tools for solving and error checking their assignments. The tools' impact on learning was not studied, but it was observed that students had fewer errors in their assignments after using the tools.

VAST is an educational tool for visualising *syntax trees* (STs), to be used in compiler or language processing classes [11]. The tool can be used to generate and visualise STs and how they are constructed. It is designed to handle very big trees. In an educational evaluation conducted on students in a language processors class, the instructors observed that the students were enthusiastic about VAST. The students' opinions on the tool were positive, and they thought that the tool was easy to use and that it supported them in the learning process.

2.3 Visualisation tools in AI courses

Naser [27] conducted a statistical analysis of students' exam results in an AI searching algorithms course, where a portion of the students was taught using a visualisation tool as part of the teaching materials. The results indicated that there was an advantage in using visualisations over traditional teaching methods, as the students who had used the tool got better results on exams. The study also showed a strong correlation between students evaluating the visualisation tools as beneficial and their performance improvement in exams.

2.4 Reviewing the role of visualisation tools

Naps et al. [26] reviewed multiple experimental studies conducted on the effectiveness of visualisation tools. They divided the tools into two categories: passive observation and active engagement. Passive observation includes the user observing different representations of the learning material through animations. Active engagement includes a varied level of 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 visualisation tools in the active engagement category were consistently more effective in terms of learning than passive observation tools. Of the tools that required active engagement from the user, 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.

2.5 Students' learning styles

Gray et al. [18] studied students' learning styles through a survey asking the students what their learning style was. The students were asked to check all that applied of a visual, auditory, or kinaesthetic learning style, or answer "I don't know". They found that of the students who did classify their learning style, 70 % stated that they were visual learners or visual combined with another style. That a majority of students think of themselves as visual learners, is in accordance with the finding of many studies discussed above that students generally like visualisation tools.

2.6 Related AST viewing tools

There is a large number of tools for visualising program state (e.g., debuggers), but only some of them can draw ASTs or parse trees. One example is the Ohm online editor [15], where one can specify a grammar and insert an expression conforming to that grammar to see the parse tree visualised. The tree is drawn below the expression, to show how the expression corresponds to the tree. Another tool called ExtendJ Explorer [16] offers to explore the ASTs built by ExtendJ and shows the parsed AST next to the Java source code it originates from.

Chapter 3

visAST

This chapter describes what visAST is and how it works. We also discuss how the tool was implemented, what technologies and dependencies were used to build it, along with an example of how easy it is to visualise one's own code with visAST. The program collects some usage data, which is discussed at the end of this chapter.

3.1 Description

Figure 3.1: The visAST landing page.

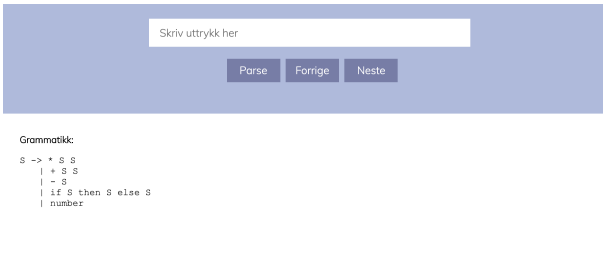
The purpose of visAST is to visualise abstract syntax trees. It is implemented as a web application, and we have made it available at `https://vis-ast.netlify.com`. The tool is used to visualise expressions of any given grammar, represented as ASTs. The tool can also

visualise the evaluation steps produced by a small-step evaluator. A small-step evaluator follows the rules of small-step operational semantics, which defines transitions from one state, or expression, to another. A transition, or evaluation step, takes an expression one step towards the final result [29]. The user can step forwards and backwards between these evaluation steps using the buttons in the GUI.

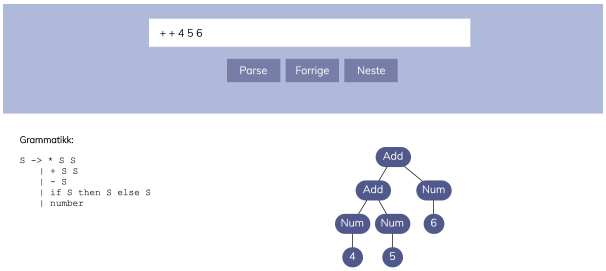
When visiting the visAST webpage the user must choose between two modes, as shown in Figure 3.1, namely *Get familiar* and *Advanced*:

3.1.1 The *Get familiar* mode

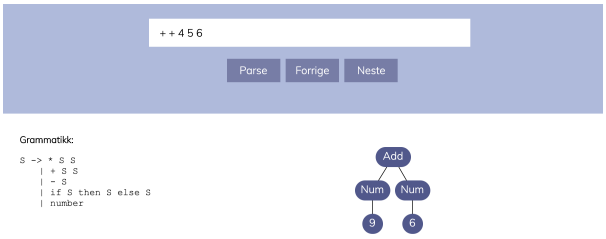
Figure 3.2: The *Get familiar* mode on the visAST webpage.



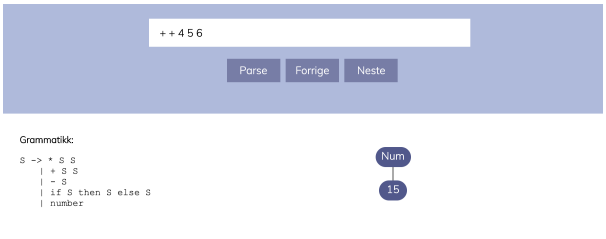
(a) View after choosing the *Get familiar* mode.



(b) View of intitital AST after parsing an expression.



(c) View after taking one evaluation step on the parsed expression.



(d) View after taking two evaluation steps on the parsed expression and a value is reached.

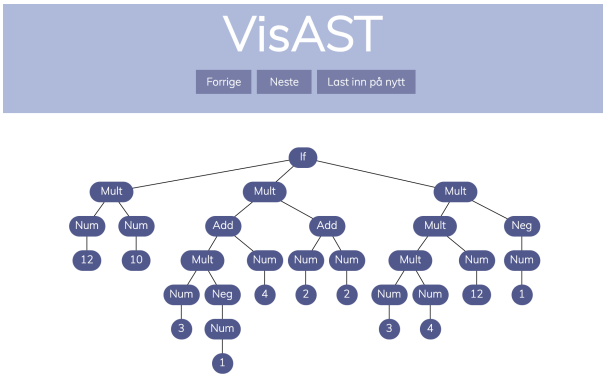
The *Get familiar* mode lets the user play around and get familiar with the tool without having to write any code. Figure 3.2a shows the initial view after choosing the *Get familiar* mode, where the user is presented with a grammar and an input field. The grammar describes what expressions can be typed into the input field and then be visualised. When the user

types in an expression and hits *Enter* or clicks the *Parse* button, the program tries to parse the expression using a built-in parser conforming to the grammar. If parsing succeeds, the program produces all evaluation steps using a built-in evaluator that takes the initial AST and returns a list of all evaluation steps. Then the initial AST is drawn on the screen, as in Figure 3.2b, and the user can step forwards and backwards through the evaluation steps using the buttons. If parsing fails, an error message is presented on the screen.

3.1.2 The *Advanced* mode

Figure 3.3: The *Advanced* mode on the visAST webpage.

(a) View after choosing the *Advanced* mode.



(b) View of intitital AST saved for a given lookup key.

The *Advanced* mode lets the user visualise her own code. The user implements her own small language, with a parser and evaluator function conforming to the rules of her language. This part of the tool works by first sending a list of ASTs, representing the evaluation steps produced by the evaluator, and a string, a lookup key to identify the user, to visAST through a function call. Then, on the webpage in the *Advanced* mode, the user is asked to type in the lookup key she sent to visAST, as shown in Figure 3.3a. After entering the lookup key the tool retrieves the ASTs saved for that key and draws the initial AST on the screen, similarly to the *Get familiar* mode. An example of a rather large expression stored and retrieved in the *Advanced* mode is shown in Figure 3.3b. In the *Advanced* mode, no grammar is displayed on the webpage because the user makes her own grammar rules when programming the parser function.

If the user did not write an evaluator function, the list of ASTs will contain only one tree. Then, one naturally cannot step through the evaluation steps as there is only one step. Otherwise, the program works in the same way as in the *Get familiar* mode. In the experiment, we required the lookup key sent to visAST to be the students' unique six-letter username from the University of Bergen. This way, we could collect data on the students' usage of the tool.

3.2 Architecture and implementation

As mentioned, visAST is a web application. The frontend speaks with the backend, a Haskell server running on Heroku, through a RESTful API[1]. The whole project can be found in three separate GitHub repositories: the visAST frontend [9], the visAST backend [7], and the visAST client [8].

In the *Get familiar* mode, the expressions typed in by the user on the webpage are sent to the backend as a POST request, as shown in Figure 3.4. On the server, the expression is parsed using a parser function stored on the server. If parsing succeeds, the parsed AST is given as input to an evaluator function that is also stored on the server. The evaluator function returns a list of ASTs, which is sent back to the frontend as a response to the POST request. Figure 3.5 shows the case where parsing fails: the response from the server is an empty list, which is handled in the frontend by displaying an error message on the screen.

In the *Advanced* mode, the user defines her own AST data type and programs a parser and an evaluator function. Next, the user calls the function `visualise` from the visAST client [8]. The `visualise` function takes as arguments a lookup key and a list of ASTs, then sends a POST request to the server, as Figure 3.6 shows. On the server, the lookup key and the list of ASTs are stored in a database. The user can then request the stored ASTs by typing in her lookup key on the webpage and hitting enter. From Figure 3.6 we can see that this triggers a GET request with the lookup key as a parameter, and the response is the associated list of ASTs retrieved from the database. If no entry with the given lookup key is found, an empty list is sent as the response and an error message is displayed on the screen, as shown in Figure 3.7.

[1] A RESTful API is an API that uses HTTP requests to GET, POST, PUT and DELETE data from the server, and all calls are stateless.

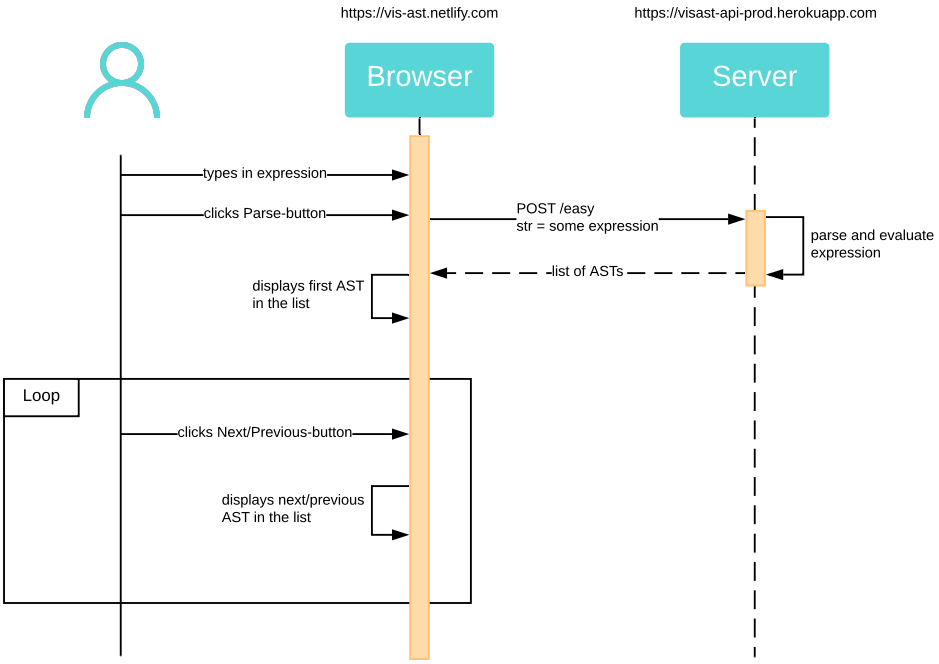


Figure 3.4: Parsing and visualising valid expressions in the *Get familiar* mode.

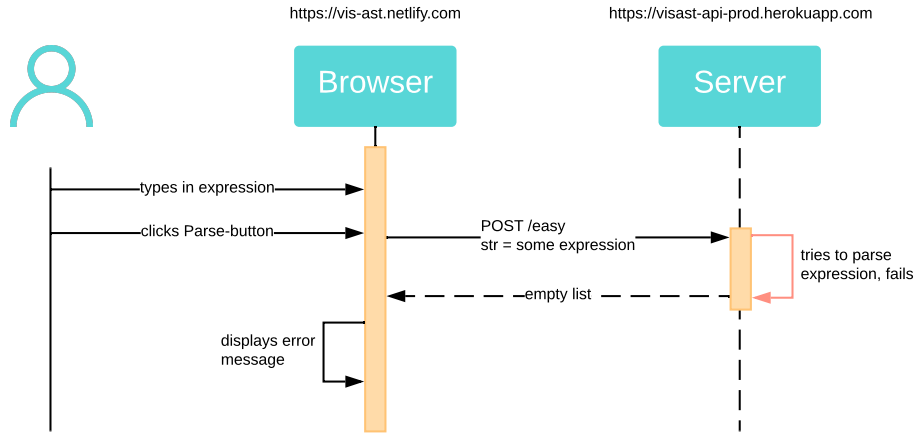


Figure 3.5: Parsing illegal expressions results in an error message in the *Get familiar* mode.

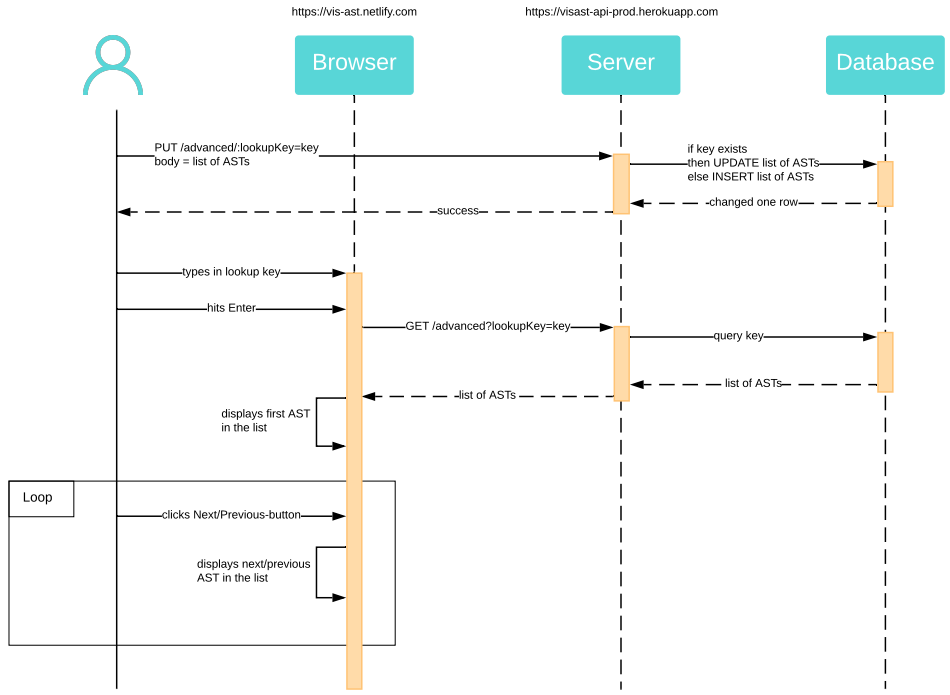


Figure 3.6: Visualising expressions in the *Advanced* mode.

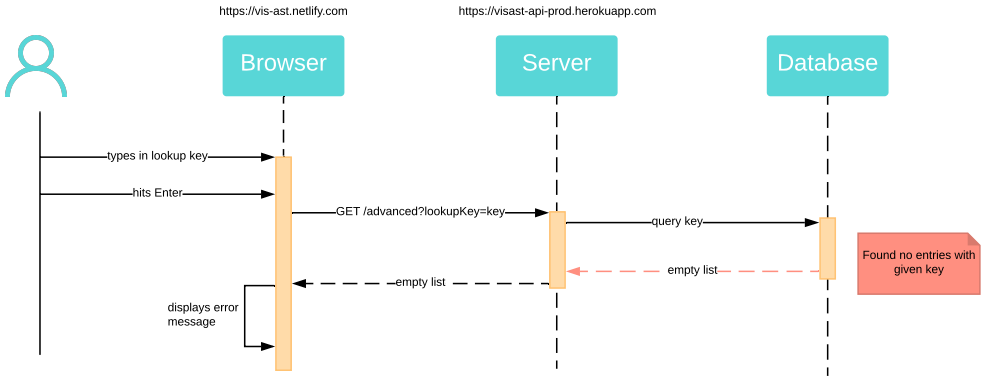


Figure 3.7: Unknown lookup key results in and error message in the *Advanced* mode.

3.2.1 Example implementation of a client for the *Advanced* mode.

Listing 3.1 shows how a programmer can use the client to send her own ASTs to visAST. Listing 3.2 shows the outline of the implementation of the client code of visAST. The client takes care of storing ASTs on the visAST server, through the function visualise . The complete source code can be found on GitHub [8].

Listing 3.1 shows how a user could use visAST to visualise her own ASTs. Expr is the data type for the AST of the user's language. The parse function is implemented following a grammar of choice; the implementation is not shown here. The evaluate function collects all evaluation steps generated by the function takeOneStep. As the function name implies, takeOneStep takes one evaluation step on the tree given as an argument. Lastly, the main function in Listing 3.1 shows an example of parsing an expression, collecting the evaluation steps, and sending them to visAST by calling the function visualise .

The program does not need to conform exactly to this structure (parse, evaluate, takeOneStep); the visualise function, in addition to a lookup key, merely needs a list of ASTs of any data type that derives Generic and is an instance of Generalise (See Section 3.2.2 for further explanation). It is up to the programmer to decide how to produce such a list of ASTs.

Listing 3.2 shows the implementation of the visualise function. We use Haskell's Servant library for communication with the visAST server. Servant allows for automatic derivation of Haskell functions that can make calls to the API endpoints. First, we describe the API as defined on the server, by defining the type API on lines 4 through 9. Using the API description as input, the Servant function client derives functions for each of the server endpoints, as shown on line 12. The endpoint used by the client is advancedPut, which is called from the run-function, on lines 26–27.

The visualise function is essentially a wrapper around the function run defined below. The visualise function converts the user-defined ASTs into a generic tree representation that visAST knows how to draw, then passes this list of generic ASTs to run. The details of this conversion are explained in Section 3.2.2. Finally, the run function sends an HTTP request with the generic ASTs and the lookup key to the server, then reports the result of the request. Again, the code in Listing 3.2 is part of visAST, not something that the user has to write.

Listing 3.1: Using the visualise function to send ASTs to visAST.

```
1   -- Type of the ASTs to be visualised
2   data Expr
3       = Num Int
4       | Add Expr Expr
5       | Mult Expr Expr
6       | Neg Expr
7       | If Expr Expr Expr deriving (Eq, Show, Read, Generic)
8
9   -- This is needed to be able to send Exprs to visualise
10  instance Generalise Expr
11
12  -- Parser conforming to some grammar rules defined by the user
13  parse :: String -> Expr
14  parse = ...
15
16  -- Collects all the evaluation steps in a list
17  evaluate :: Expr -> [Expr]
18  evaluate (Num num) = Num num]
19  evaluate expr = expr : evaluate (takeOneStep expr)
20
21  -- Small step evaluator, taking one evaluation step on the given Expr
22  takeOneStep :: Expr -> Expr
23  takeOneStep (Num num) = Num num
24  takeOneStep (Add (Num n1) (Num n2)) = Num $ n1 + n2
25  -- ... other reductions of the interpreter
26
27  main = do
28      let initialExpr = parse "+ * 1 2 15" -- example expression
29          asts = evaluate initialExpr -- get list of ASTs
30      visualise "someLookupKey" asts
```

Listing 3.2: Implementation of the visualise function for a visAST client.

```
1   -- Some imports...
2
3   -- Client API corresponding to the server's endpoints
4   type API = "easy" :> ReqBody '[JSON] InputString
5            :> Post '[JSON] [GenericAST]
6       :<|> "advanced" :> Capture "lookupKey" String
7            :> ReqBody '[JSON] Steps :> Put '[JSON] ResponseMsg
8       :<|> "advanced" :> QueryParam "lookupKey" String
9            :> Get '[JSON] [GenericAST]
10
11  -- Handlers (Haskell functions) to call the API endpoints
12  easy :<|> advancedPut :<|> advancedGet = client (Proxy :: Proxy API)
13
14  -- Function for sending ASTs to visAST, then open the browser on the
        ↪ visAST webpage
15  visualise :: Generalise a => String -> [a] -> IO ()
16  visualise key asts = do
17      putStrLn "Sending ASTs to the visAST server"
18      run key (map toGeneric asts)
19      openBrowser "https://vis-ast.netlify.com" >>= print
20
21  -- Function for running client request
22  run :: String -> [GenericAST] -> IO ()
23  run key asts = do
24      manager' <- newManager defaultManagerSettings
25      let url = BaseUrl Http "visast-api-prod.herokuapp.com" 80 ""
26          query = advancedPut key (Steps { evalSteps = asts })
27      res <- runClientM query (mkClientEnv manager' url)
28      case res of
29          Left err ->
30              putStrLn "Something went wrong."
31          Right response ->
32              putStrLn $ "Success: ASTs stored on the visAST server."
```

3.2.2 Datatype-Generic AST visualiser

The user can invoke visualise with (practically, see below) any AST data type. This is important because it allows a student to use visAST with any language she is creating. To

attain such generality, we resort to somewhat advanced Haskell features, namely Haskell Generics, defined in the GHC.Generics module.

Generics in Haskell, often called datatype-generic programming, *is about making programming languages more flexible without compromising safety* [17]. It allows one to manipulate data types generically. A common use of Generics is the automatic generation of functions required by common type classes such as Show and Eq. We use the Generics features to automatically map user-defined ASTs into our own generic AST data type, GenericAST. For simplicity, we denote user-defined ASTs as UserAST in this subsection. We defined the class Generalise, which contains a function toGeneric that takes a UserAST and converts it into a GenericAST. A GenericAST contains, for each node, the name of the node and a list of its children, which is all visAST needs for drawing trees. Listing 3.3 shows how the function toGeneric maps a UserAST, here type-variable b, into Haskell Generics' own generic representation type, Rep, before calling the function gtoGeneric. The mapping to Rep is provided by the GHC.Generics module through the function from.

Listing 3.3: Implementation of the Generalise class, which defines the mapping from a user-defined AST to a GenericAST.

```
1 class Generalise b where
2     toGeneric :: b -> GenericAST
3
4     default toGeneric :: (Generic b, GGeneralise (Rep b)) => b -> GenericAST
5     toGeneric = head . gtoGeneric . from
```

The function gtoGeneric comes from the GGeneralise class (see Listing 3.4) and defines the mapping from the Rep type to a GenericAST. The Rep type describes any algebraic data type as sums of products. Concretely, it views a data type's collection of constructors as a sum type and the parameters of each constructor as a product type. Rep also stores meta-information like the constructor names of the data type it represents. This view allows us to write algorithms that work on any data types. The general structure of these algorithms is a traversal of the operands of the sums, and further of the arguments of the products. They then extract information of or manipulate particular constructors. In particular, here we define an algorithm that transforms the generic Rep type to a common tree representation, GenericAST, that visAST knows how to visualise. In Listing 3.4 we define instances of GGeneralise for every branch of the Rep type. We do not special-case on any particular

Listing 3.4: Implementation of the GGeneralise class, which defines the mapping from a Rep to a GenericAST.

```
1  class GGeneralise f where
2      gtoGeneric :: f a -> [GenericAST]
3
4  -- | Unit: Constructors with no arguments
5  instance GGeneralise U1 where
6      gtoGeneric U1 = []
7
8  -- | Product
9  instance (GGeneralise a, GGeneralise b) => GGeneralise (a :*: b) where
10     gtoGeneric (a :*: b) = gtoGeneric a ++ gtoGeneric b
11
12 -- | Sum
13 instance (GGeneralise a, GGeneralise b) => GGeneralise (a :+: b) where
14     gtoGeneric (L1 x) = gtoGeneric x
15     gtoGeneric (R1 x) = gtoGeneric x
16
17 -- | Data type information, not interesing here
18 instance (GGeneralise f) => GGeneralise (M1 D c f) where
19     gtoGeneric (M1 x) = gtoGeneric x
20
21 -- Constructor meta-data: Here we extract the constructor name
22 instance (GGeneralise f, Constructor c) => GGeneralise (M1 C c f) where
23     gtoGeneric (M1 x) = [GenericAST nodeName children]
24         where
25             nodeName = conName (undefined :: t c f a)
26             children = gtoGeneric x
27
28 -- Selector meta-data, discarded
29 instance (GGeneralise f, Selector c) => GGeneralise (M1 S c f) where
30     gtoGeneric (M1 x) = gtoGeneric x
31
32 -- Constructor parameter: The data contained in the node
33 instance (Generalise f) => GGeneralise (K1 i f) where
34     gtoGeneric (K1 x) = [toGeneric x]
```

constructors, we just call the function conName to extract the constructor names of the original data type recursively, as shown on line 22–26 in Listing 3.4.

Lastly, we must implement instances of Generalise for primitive types, i.e., define how to convert an Int to a GenericAST. The instances for Int and String are shown in Listing 3.5.

Listing 3.5: Instances of Generalise for primitive types.

```
1 instance Generalise Int where
2     toGeneric i = GenericAST (show i) []
3
4 instance Generalise String where
5     toGeneric s = GenericAST s []
```

With this machinery in place, for the users to visualise their own ASTs they only need to meet the following two requirements, which amounts to a minimal amount of boilerplate:

- Make their UserAST data type derive Generic (deriving Generic).
- Make their UserAST data type an instance of Generalise (instance Generalise UserAST).

3.2.3 Drawing trees using SVG

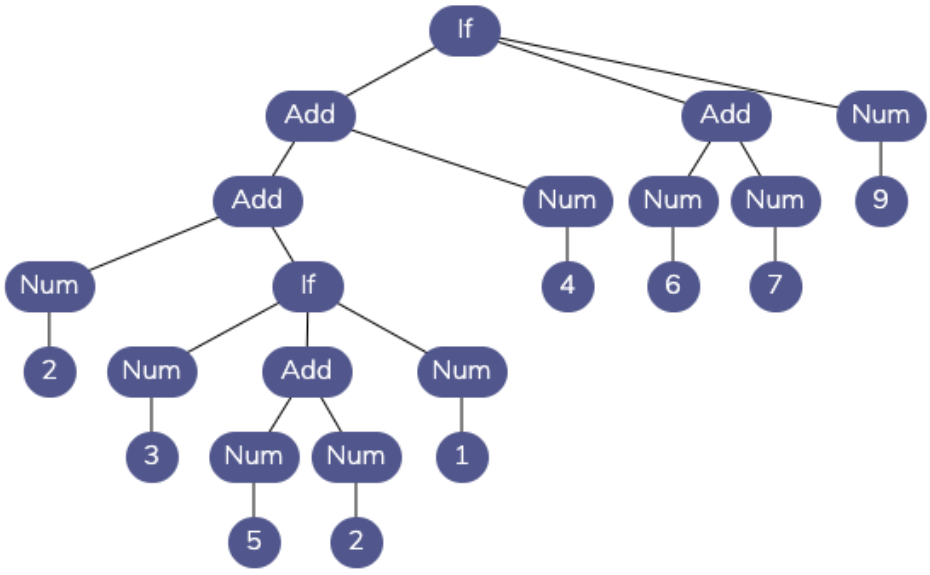


Figure 3.8: Example tree drawn with SVG.

In visAST, the ASTs are drawn as SVG[2]. Trees are drawn top-down, starting from the root node. The conversion from an AST data type to a GenericAST returns a name for each node and a list of children. The draw-function thus simply recurses through the tree, drawing all the nodes along the way. Each node is drawn as an SVG circle containing the node name, and the children are distributed below their parent. Edges between every parent and child are drawn as SVG lines.

For placing the nodes, visAST tries to be clever so that the outcome is visually pleasing. The strategy is roughly as follows:

The nodes of a tree are recursively drawn on the screen, starting at the root node, placing it in the middle (of the x-axis) of the screen. Then, the total width of the tree is computed by recursing through all of its children and adding the widths together. The children of the root node are placed one level below the root, where they are distributed along the x-axis based on how much space they take up, i.e., based on their own widths. The way the children are distributed is best explained using an example: Let's say a tree has a total width of 6 (of some unit), as shown in Figure 3.9. The root has three children, or subtrees, of widths 2, 1, and 3, respectively. Then, the first child is given a space of $\frac{2}{6}$, or $\frac{1}{3}$, of the total width of the tree, the second child gets $\frac{1}{6}$, and the third child gets the remaining $\frac{3}{6}$, or $\frac{1}{2}$. We then draw the children recursively, in the same way as just desrcibed.

[2]SVG, Scalable Vector Graphics, is an XML-based vector image format for two-dimentional graphics.

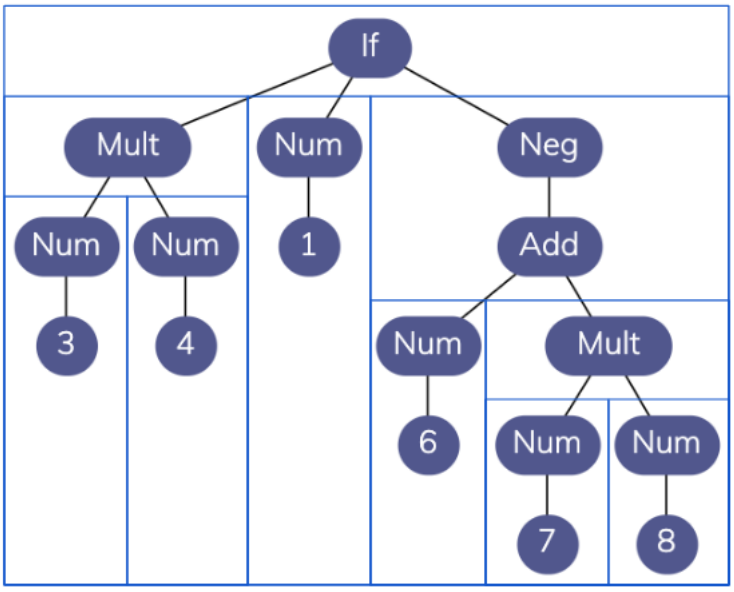


Figure 3.9: Example of how children are distributed below their parent, based on their own widths. Total width of the tree is 6 (of some unit).

3.2.4 Tool dependencies

The project builds on several technologies, languages, and libraries:

Elm The frontend is written in Elm. Elm is a relatively new, purely functional programming language for the web that compiles to JavaScript. Url: `https://elm-lang.org/` (Version 0.19).

Haskell The API server is written in Haskell. Haskell is a statically typed, purely functional programming language. Url: `https://www.haskell.org/` (Version 8.4.4).

Servant The Servant library was used to define and serve the API for the server. Servant is a set of Haskell libraries for writing type-safe web applications. Url: `https://docs.servant.dev/en/stable/index.html` (Version 0.16).

Netlify The visAST webpage is hosted on the cloud computing service Netlify, which offers hosting of static websites. Url: `https://www.netlify.com/`.

Heroku The API server is hosted on Heroku, a cloud platform which makes it easy to deploy, run, and manage applications. Url: `https://www.heroku.com/`.

PostgreSQL The API server uses a PostgreSQL database to store the data. PostgreSQL is an object-relational database. Url: `https://www.postgresql.org/`.

3.3 Data collection

The primary reason for choosing to implement parts of the visualisations on the server, rather than fully in Elm on the web client, was to be able to collect usage data. That is, data on how the students interact with the visualisation tool. Another reason was to make it easier for letting the students visualise their own Haskell implementation, by providing them with one single Haskell function to interact with the tool. This function, when called, makes a request to the server to store the student's list of ASTs, relieving the students from implementing their own client to interact with the server.

We recorded every request to the server. Concretely, the server logged three types of events:

- When a user parsed an expression in the *Get familiar* mode.
- When a user requested ASTs for a specific lookup key in the *Advanced* mode.
- When a user called `visualise` to store ASTs on the server.

We recognise that logging could be more detailed. We could, e.g., log backwards/forwards clicks on the webpage and the number of failed attempts to parse expressions in the *Get familiar* mode. This would give more information on how the program is actually used, which might lead to insights into students' learning.

Chapter 4

Experiment and results

This chapter describes the experiment with visAST conducted on computer science students at the University of Bergen (UiB).

4.1 Research method

4.1.1 Design

The experiment consisted of an assignment in an undergraduate functional programming class with an accompanying questionnaire to be answered after completing the assignment. The assignment contained a set of programming tasks, some of them involving the use of visAST.

We were not able to conduct a controlled experiment. This was because the course is compulsory for most of the students taking it and we wanted equal treatment of the students. The experiment was therefore conducted on all students in the course as part of a compulsory assignment. If one were to have a control group, so that only a portion of the students were instructed to use visAST and the tool proved to have either a positive or negative effect on their learning then the two groups, the students who used the tool and the students who did not, would not have been treated equally.

There are experiment settings to minimise such effects, e.g. we could have used a control group, introducing visAST to only one portion of the students. To even out the potential differences between the groups we could have taught the control group about visAST after the experiment was done. Due to the limited time frame of the thesis, we were not able to do this, and we only had one chance of conducting the experiment on suitable participants.

Even though we did not use a control group, by comparing the students' answers to the questionnaire and their performance in the assignment, we can nevertheless see some connections.

4.1.2 The assignment

The first part of the experiment was a compulsory assignment in the course "Functional Programming" at the Department of Informatics at UiB. The main learning outcome for this course is knowledge about and experience with the functional programming paradigm. Programming exercises are in Haskell. The course also teaches how to use Haskell to parse and evaluate expressions, which was the main focus of this assignment. The original assignment can be found in Appendix A.

The participants were given a concrete grammar describing a simple expression language consisting of arithmetic expressions along with if-then-else expressions:

$$
\begin{aligned}
\langle S \rangle \;::=\; & \text{`}\texttt{*}\text{'}\ \langle S \rangle\ \langle S \rangle \\
\mid\; & \text{`}\texttt{+}\text{'}\ \langle S \rangle\ \langle S \rangle \\
\mid\; & \text{`}\texttt{-}\text{'}\ \langle S \rangle \\
\mid\; & \text{`}\texttt{if}\text{'}\ \langle S \rangle\ \text{`}\texttt{then}\text{'}\ \langle S \rangle\ \text{`}\texttt{else}\text{'}\ \langle S \rangle \\
\mid\; & \langle \mathit{digit} \rangle^{+}
\end{aligned}
$$

Note: For simplicity, '-' is a unary minus. There is no binary subtraction operator.

Based on this language description the students were asked to complete the following tasks:

1.1 Implement a parser for the described language.

1.2 Implement a pretty-printer for ASTs of expressions in the language.

1.3 Implement a small-step evaluator for the language.

1.4 Write a function that can read commands from the user and perform the requested action. An example of an action is a user asking to have an AST pretty printed on the screen.

2.1 Add a command to the function from task 1.4. The command should visualise ASTs using visAST.

2.2 Fill in a questionnaire about visAST.

3.1 As a task unrelated to visAST, find the type of a given function using the rules for type inference and unification.

Before starting on the tasks, the participants were instructed to download the GitHub-repository for the visAST client to check that the client worked on their machine. All that is needed to connect to and run visAST is one Haskell function, `visualise`, but the whole repository is required because the function relies on several external libraries.

Task 1.1

The first task was to write a parser for the described language, which should take a string as an argument and return an AST. As a voluntary task, giving no extra credit, the students were asked to test their parser using visAST. Our explicit goal was to make the use of visAST as convenient as possible. Therefore, all that the student needed to do was to call the function `visualise` to trigger visAST. The `visualise` function takes as a parameter the student's username and a list of ASTs, and stores the ASTs on the server. At this point, the students were working with only one AST, so they needed to call `visualise` with a singleton list containing this AST.

Task 1.2

This task asked to implement a function `prettyPrint` that takes as a parameter an AST and prints it to standard output in a specified format. This task mainly focused on the use of IO-functions and was not directly tied to the experiment.

Task 1.3

In task 1.3 the students were given a small-step operational semantics for the expression language before being asked to implement an evaluator according to the semantics. With this evaluator a program is gradually, step-by-step, reduced to a normal form/a value. visAST uses all of the intermediate ASTs as they are potentially useful for a student for gaining understanding of how evaluation works. Concretely, the task asked the students to implement the function takeOneStep, which takes an AST and returns a new AST, after taking one evaluation step on the input AST.

Assuming takeOneStep is implemented, the operational semantics of the expression language was described to the students as follows:

1. Num x is a value and evaluates to itself. Example: takeOneStep (Num 5)== Num 5.
2. Add x y should first evaluate x to a value, step-by-step. If x is a value then y should be evaluated to a value, step-by-step. If both x and y are values then they are added together and returned wrapped in a Num.
3. Mult x y should, like with Add, first evaluate x to a value, then y, then multiply x and y.
4. If eCond eThen eElse should, as above, first evaluate eCond (the condition of the if-sentence) to a value, step-by-step. If eCond is already a value then eCond's value is checked. If eCond is not equal to 0 then eThen is returned, otherwise eElse is returned.

Task 1.4

Task 1.4 was to write a function mainStep that reads from standard input and performs actions based on that input. For example, the function should, when called, read an expression from standard input, parse it, and take evaluation steps on it. This task was not closely tied to visAST and the experiment.

Task 2.1

This task asked the students to extend the mainStep function with one command: Visualising all the evaluation steps of an expression until it reaches a value. The students were supposed to use the visualise function, calling it with their username and a list of the ASTs representing each evaluation step. To retrieve this list they could call takeOneStep from task 1.3 with the expression until it reaches a normal form, collecting all intermediate ASTs in a list. Every time visualise is called, the evaluation steps are sent to and made available on the server, and the visAST webpage is opened in a browser. Then, the student can type in her username in the browser to look at the visualisations of the evaluation steps.

Task 2.2

This task included the questionnaire as a Google Form. The questions can be found in Table 4.1. In the questionnaire, the students were first asked for their username, then about how they experienced the use of visAST.

Task 3.1

The last task was unrelated to the experiment and is not described.

4.1.3 Participants

The experiment was conducted on all students taking the course "Functional Programming" at the Department of Informatics at UiB. There were originally 214 students signed up for the course. A total of 93 students turned in the assignment, whereas 79 of them had answered the questionnaire. The low number of students participating is explained by the assignment being published late in the semester. This is a tough course, and it is common for many students to drop out during the semester. The questionnaire was given as part of the assignment, and it was stated clearly that answering it was mandatory but the answers to the questions would not be graded.

Most of the participants were computer science students in their third semester of their bachelor degree at the Department of Informatics at UiB. A small number of students came from *Høgskolen på Vestlandet (HVL)*, or from other bachelor or master programs at UiB.

Standard curricula at study programs at UiB and HVL do not include learning about parsing and interpreters. Therefore most students knew little to nothing about these topics before this course. At the time the assignment was published, the course curriculum had already given some exposure to these topics. The results from the questionnaire show how the students characterised their own knowledge of ASTs, parsing, and evaluation. None of the participants were introduced to visAST before this experiment.

4.1.4 Procedures

The experiment was conducted at the University of Bergen in November 2018. The compulsory assignment was released on November 13th, and the participants had two weeks to finish the assignment. Answering the questionnaire was a part of the assignment and had the same deadline as the rest of the assignment but was not graded. The assignment was to be handed in individually, yet some collaboration was allowed, except on the questionnaire.

The assignment was given in writing through the course pages. The instructions included the programming tasks, instructions for visAST, and a link to the questionnaire. All information about the questionnaire was given in the Google Form linked to from the instructions writeup.

4.2 The questions

4.2.1 The questionnaire

The questionnaire was created as a Google Form. The questions and description were in Norwegian, a translated version is given below:

Table 4.1: English version of the questionnaire used in the experiment

Questions about visAST – compulsory 2, INF122
Filling out this form is your answer to task 2.3. Your answers do not count on your total score on the assignment, we will only check if you answered or not. Questions marked with * are mandatory to answer, this includes all questions except the last one, which asks for your opinions/suggestions/etc. if you have any. visAST is a program under development, and is supposed to be a teaching tool for better understanding how *abstract syntax trees (ASTs)* work and look. We therefore need feedback on how it has been to use this, if it helped with understanding, etc. Thank you for your feedback! This helps both our research and potentially future students.
Q1 – Your Mitt UiB username (on the form *abc123*)* **Your answer here.**
Q2 – How well did you understand ASTs before this assignment?* **Understood little o o o o o Good understanding**
Q3 – How much did you use visAST in this assignment?* **As little as possible o o o o o A lot**
Q4 – How useful was visAST for your understanding of ASTs in this assignment?* **Not useful o o o o o Very useful**
Q5 – How useful do you think visAST would have been for your understanding the first time you were introduced to ASTs?* **Not useful o o o o o Very useful**

Continued on next page

Table 4.1 – *Continued from previous page*

Q6 – Did visAST help you understand parsing better?* **Yes o o No**
Q7 – Did visAST help you understand evaluation of expressions better?* **Yes o o No**
Q8 – We would like your thoughts about visAST, in your own words. What was useful, what was not? What was easy, what was not? Do you have any suggestions for improvements? Other thoughts? **Your answer here.**

4.2.2 Research questions

To assess the effect visualisation tools can have in learning about interpreters and other language processors, we were interested in seeing how students evaluate both their knowledge of ASTs and related topics, and their experiences with visAST. From our hypothesis, *visAST helps students understand ASTs better*, 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?

Questions 2, 3, and 4 deal with the students' previous understanding and how much they used the tool in the experiment. These are all relevant for answering the research question 1.

To answer the second research question, questions 2, 3, 6, and 7 are relevant. Questions 6 and 7 directly address this research question, and question 3 is also relevant because it has to do with how much the students used the tool. The relevance of question 2 may not be obvious, but since an understanding of ASTs is needed to work with both parsing and evaluation, answers to this question are also useful in this context.

The rationale for including question 5, which asks how useful the students think visAST would have been the first time they were introduced to ASTs, is that we suspected that the students might have already learned too much about ASTs for visAST to be very useful. Answers to a hypothetical question like this should be interpreted with caution, but they seem to indicate that visAST would have been more helpful if introduced earlier.

Question 8 asks for any personal opinions or suggestions related to the experiment or the tool. An answer was not obligatory. The purpose of this question was to get feedback, on the students' experience with visAST and its design and user experience, to help in further development of the tool. It was expected that not many students answer this question; in the current study culture, students tend not to do more work than required to finish compulsory assignments. We expected both positive and negative feedback on the experience of using visAST.

4.2.3 Response rates

In total 79 students answered the questionnaire. There were 214 students in the course and 93 of them turned in the assignment. Relative to the number of students in the course the response rate was 37 %, but relative to the number of students turning in the assignment the response rate was 85 %. The response rate to the final, voluntary, question was 61 %, with 48 out of 79 participants answering this question. No student who did not turn in the assignment answered the questionnaire.

4.2.4 Results

Results from each question are presented and described below. We first look at results from all students combined, then we look at the differences in answers from different types of students. We divide the students into three groups based on their final grade in the course, i.e., their exam score, which ranged from 0 to 105 points:

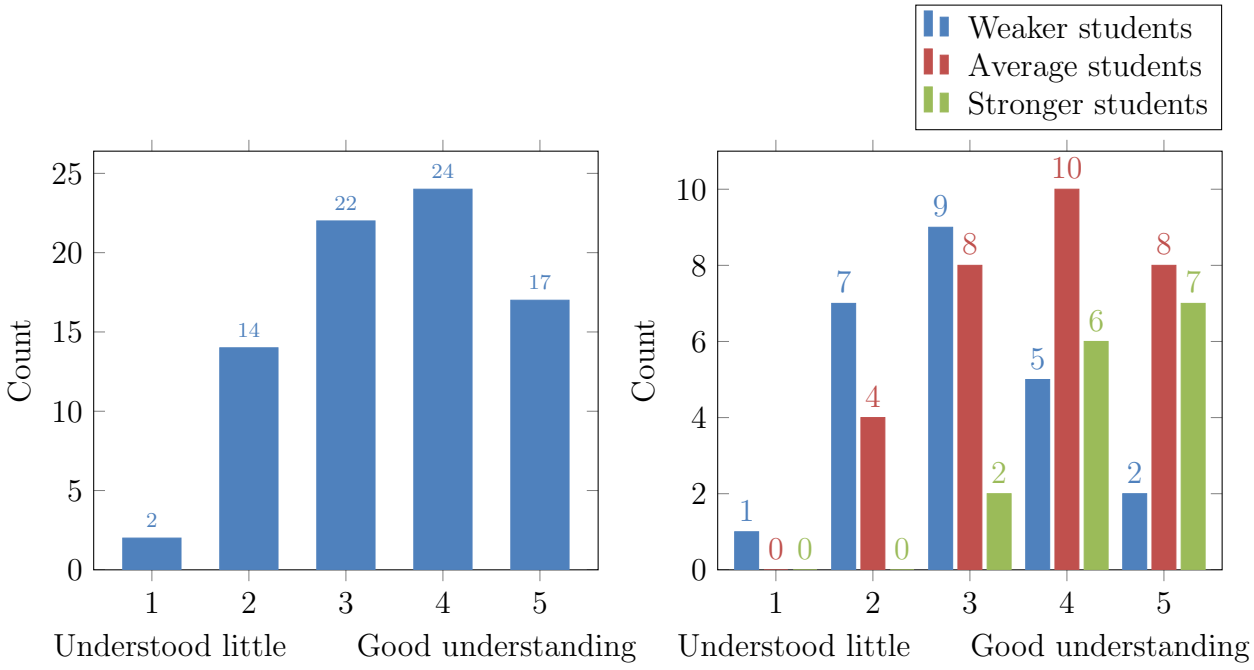
- Stronger students (80 points or more)
- Average students (50–80 points)
- Weaker students (less than 50 points)

When dividing the students into groups we excluded those who answered the questionnaire but did not take the final exam. It is not obvious why these students did not finish the course, and therefore we cannot place them in a group. In total ten students were excluded, resulting in 69 students making up the three groups. The reason for looking at each group individually was to investigate how the different types of students experienced visAST, to potentially discover who benefited the most from using the tool. As discussed in Chapter 2, many studies show that visualisation tools benefit different types of students differently.

Questions 2 through 5 asked the participants to respond on an ordinal scale with labels on the lower (1) and upper (5) bound of the scale. To compute the averages from responses on an ordinal scale we converted the scale to its numerical equivalent. In general, one should be careful when converting from a numerical scale to an ordinal scale, as this might give misleading results [23]. 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 when converting to numerical values is low [23].

Figure 4.1: Q2 – How well did you understand ASTs before this assignment?



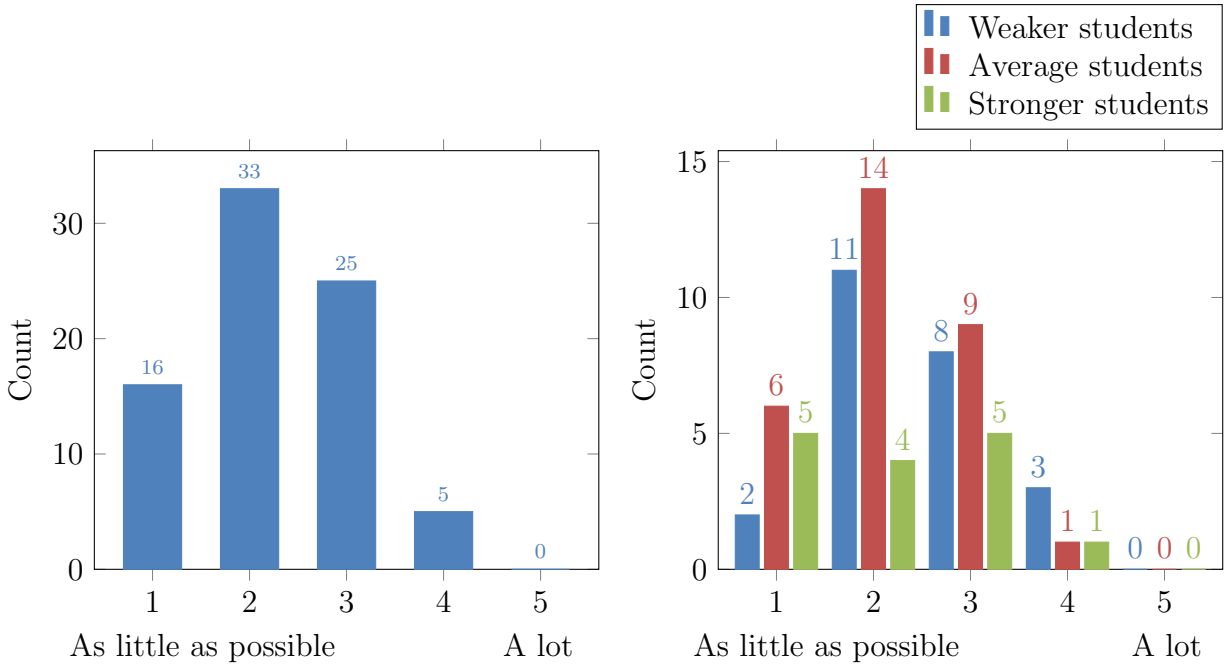
(a) All answers.

(b) Grouped answers.

Answers from question 2 gave a mean of 3.5, suggesting that the students already had a moderately good understanding of ASTs, and a standard deviation of 1.1, indicating that the student population had a varying level of prior knowledge. We can see in Figure 4.1a that there were few students who "understood little" about ASTs before the assignment.

Figure 4.1b shows that among the stronger students the majority had a good understanding of ASTs already, with all answers ranging from 3 to 5 on the scale and a mean of 4.3. The average students, with a mean of 3.7, had a fairly good understanding of ASTs and we can, unsurprisingly, see that the only ones who "understood little" about ASTs were among the weaker students. The weaker students had a mean of 3.0, indicating that even this group had a moderately good understanding, with a few exceptions.

Figure 4.2: Q3 – How much did you use visAST in this assignment?

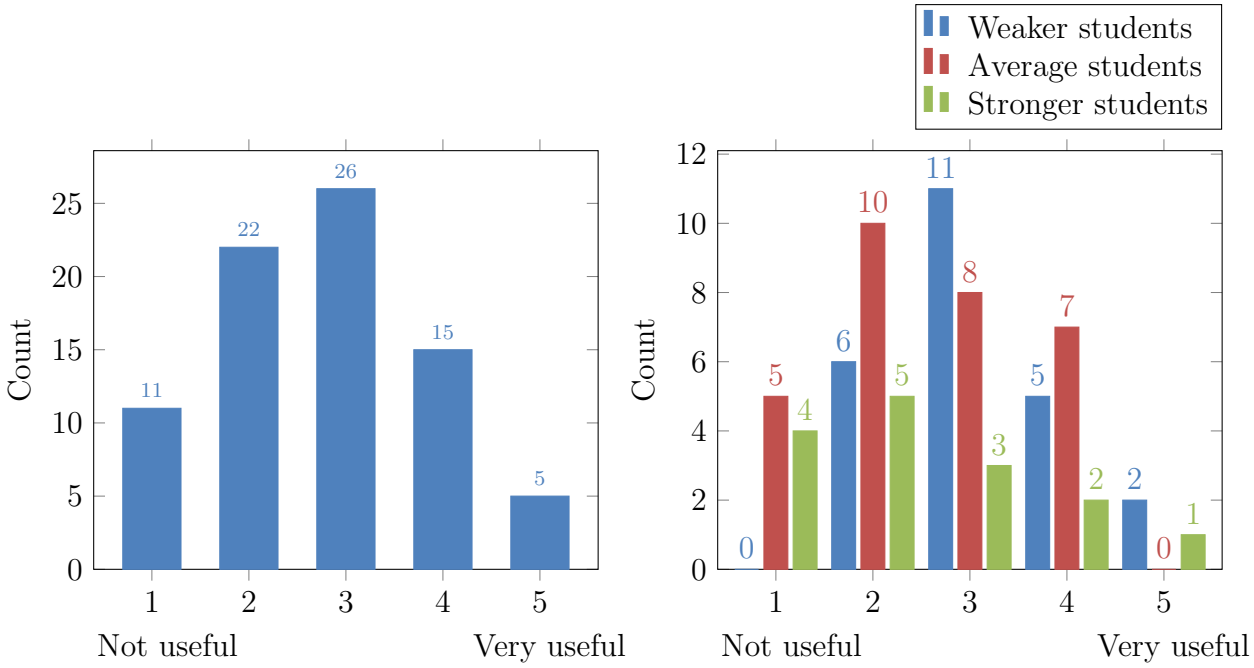


(a) All answers.

(b) Grouped answers.

Figure 4.2a shows that no students reported using visAST ”a lot” in the assignment, but that a substantial number of students reported moderate use. The mean was 2.2 and the standard deviation was 0.9. From the grouped answers in Figure 4.2b, we can see that the weaker students were the ones who used the tool the most, with a mean of 2.5. The stronger and average students’ answers had means of 2.1 and 2.2, respectively.

Figure 4.3: Q4 – How useful was visAST for your understanding of ASTs in this assignment?



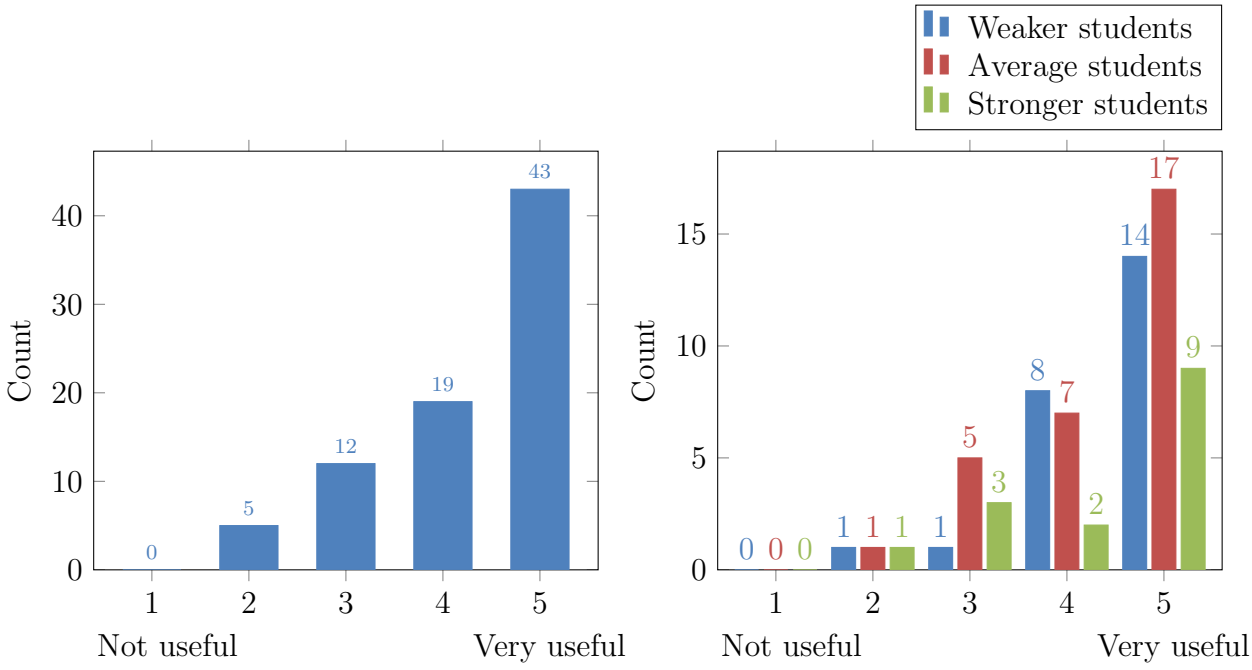
(a) All answers.

(b) Grouped answers.

Figure 4.3a shows that the average of how useful the students found visAST for understanding ASTs is approximately in the middle of the scale. This answers our first research question from Section 4.2.2, confirming that a large number of students find visAST moderately or more helpful in understanding ASTs better. The mean for this question was 2.8 with a standard deviation of 1.1.

In Figure 4.3b we can see that of the three groups, the weaker students found visAST the most helpful for understanding ASTs. Not a single student in this group answered 1, "not useful"! The mean of the weaker students was also the highest of the groups, with 3.1 versus 2.6 (average group) and 2.4 (stronger group). The stronger students found the tool the least useful, yet some of them reported that they, too, found visAST either useful or very useful.

Figure 4.4: Q5 – How useful do you think visAST would have been for your understanding the first time you were introduced to ASTs?



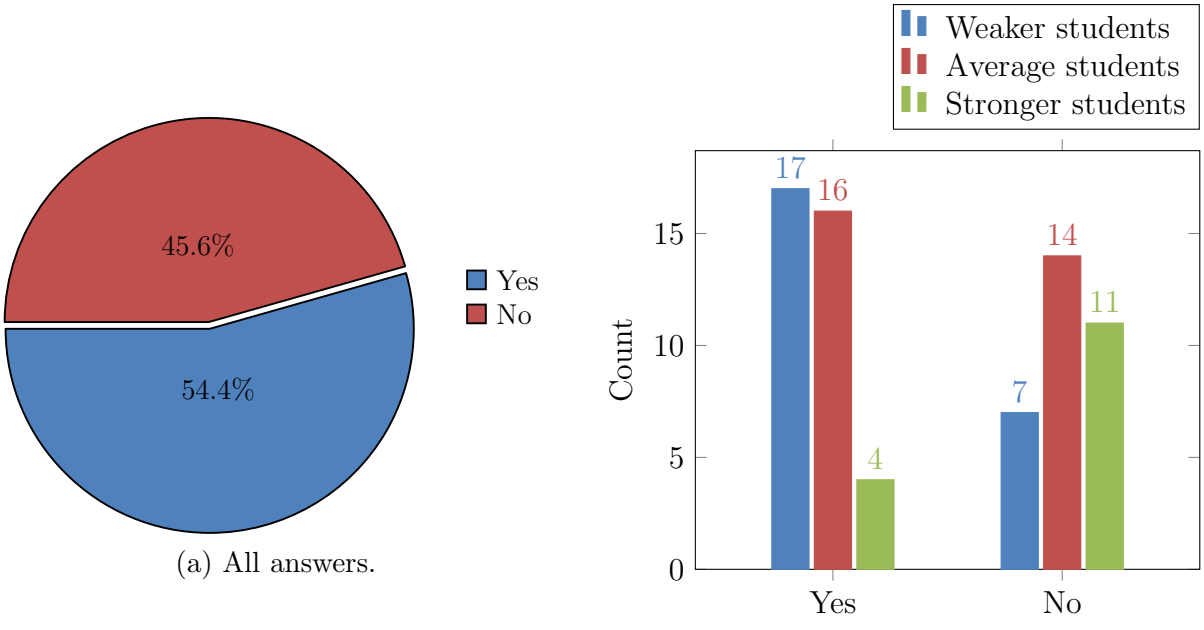
(a) All answers.

(b) Grouped answers.

Figure 4.4 represents the students' thoughts on how useful visAST could have been if introduced the first time they learned about ASTs. In Figure 4.4a, showing all answers combined, the mean is 4.3, indicating that the students believed that visAST would have been fairly to very useful if introduced earlier, with a standard deviation of 0.9. No one answered that the tool would have been "not useful" when ASTs were first introduced.

Figure 4.4b shows the answers from the different types of students, and we can see no significant difference between the three groups. The means are 4.3 for both the stronger and the average students, while the weaker students have a mean of 4.5. All three groups believe that the tool would have been useful at an earlier point, with the weaker students slightly more convinced than the other groups.

Figure 4.5: Q6 – Did visAST help you understand parsing better?

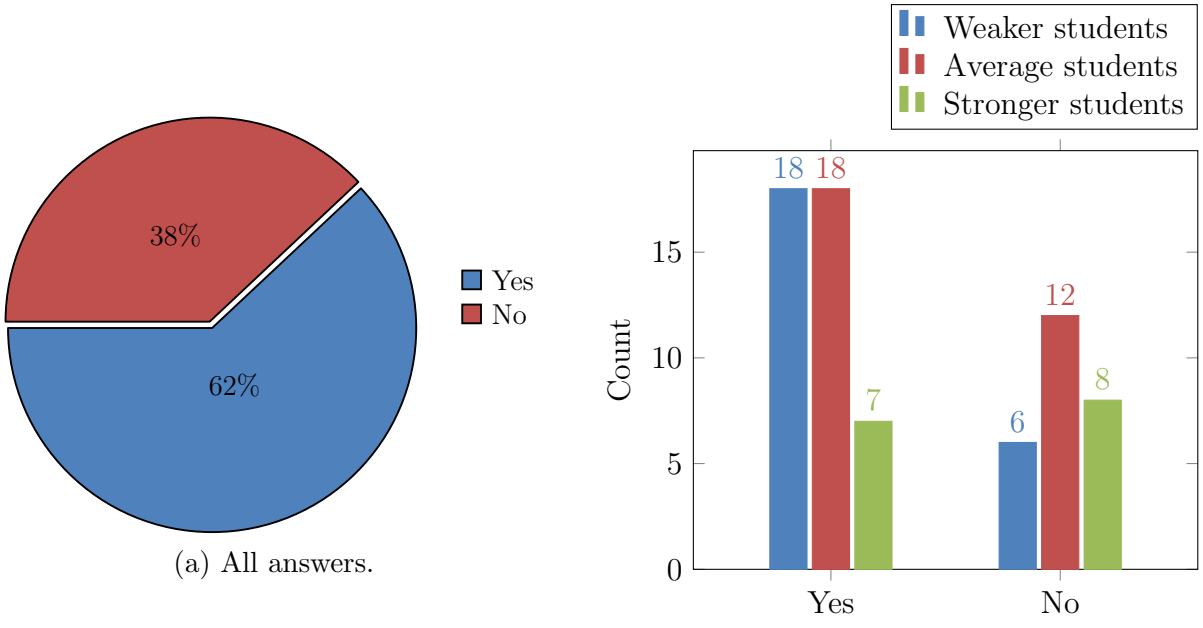


(a) All answers.

(b) Grouped answers.

Figure 4.5a shows that 54.4 % of the students thought that visAST helped in understanding parsing better, answering part of the research question 2 in Section 4.2.2; a large number of students think visAST is helpful for understanding parsing. In total 43 students answered yes to the question, and the remaining 36 students answered no. Figure 4.5b shows a clear difference in answers from the different types of students; only 27 % of the stronger students answered yes to this question. The average students were more divided, with almost a 50–50 division of answers, whereas as much as 71 % of the weaker students agreed that visAST helped with parsing.

Figure 4.6: Q7 – Did visAST help you understand evaluation of expressions better?



(a) All answers.

(b) Grouped answers.

Figure 4.6a shows that 62 % of the students thought that visAST helped in understanding the evaluation of expressions better, which is an answer to the second half of the research question 2 in Section 4.2.2. In total, 49 students answered yes to the question and the remaining 30 students answered no. From Figure 4.6b we can see that also here the weaker students found visAST the most helpful of the three groups, with 75 % answering yes to the question. The average students were a bit more divided, with 60 % answering yes. The stronger students found visAST more helpful for understanding expression evaluation than parsing, but still under half of them, 47 %, reported that visAST was useful for understanding evaluation better.

Summary of grouped results

Table 4.2 summarises the means from each group of students (weaker, average, and stronger) for every question in the questionnaire.

Seeing the grouped results in context of each other, we observe that the weaker students had less prior knowledge on ASTs than the other two groups and were also the group that

Table 4.2: Means of the answers to the questionnaire, grouped after type of student. The last two questions show the percentage that answered "yes".

	Weaker students	Average students	Stronger students
Q2: How well did you understand ASTs before this assignment?	3.0	3.7	4.3
Q3: How much did you use visAST in this assignment?	2.5	2.2	2.1
Q4: How useful was visAST for your understanding of ASTs in this assignment?	3.1	2.6	2.4
Q5: How useful do you think visAST would have been for your understanding the first time you were introduced to ASTs?	4.5	4.3	4.3
Q6: Did visAST help you understand parsing better?	71 %	53 %	27 %
Q7: Did visAST help you understand evaluation of expressions better?	75 %	60 %	47 %

used visAST the most in the assignment. In each question about how useful the students found visAST, for understanding ASTs, parsing, and evaluation, the weaker students had the highest mean and highest "yes" percentage. In contrast, the stronger students consistently had the lowest means and percentages, and were also the group that reported the highest level of prior knowledge on ASTs.

Results from question 8

We analysed the results from question 8, the free form question, and identified four common themes in the students' answers. Below we discuss these themes and quote some representative comments from students.

Several students commented on the timing of when the tool was introduced. The students pointed out that since visAST was made available late in the course, they already had a good understanding of ASTs and therefore the tool was not that useful. This confirms our findings

from the earlier questions in the questionnaire. Ten students mentioned that they believe the tool would have been very useful the first time they were introduced to ASTs, and some explicitly suggested introducing it in the first assignment instead. Below we paraphrase and group typical comments from the students on the timing:

- *I already had a good understanding of ASTs before this assignment.* (7 students)
- *It is a bit too late for visAST to be useful now that I have the understanding.* (11 students)
- *Probably very useful when learning this for the first time.* (10 students)

Next, we saw that the students generally evaluated the tool's usability positively. They liked the simple design and found the visualisations neat. Also, three students mentioned how useful it was to have their own code visualised and in that way be able to verify the correctness of their code. Below are some representative comments, paraphrased, related to the usability of the tool:

- *The tool was very intuitive, with a simple design.* (6 students)
- *Neat and tidy way of visualising trees.* (4 students)
- *It was useful to be able to verify that my own code produced the correct answer.* (3 students)

Some students also had suggestions for future features or improvements for visAST. Two students mentioned how it would be helpful to highlight the subtree to be evaluated next, to make the transition between evaluation steps clearer. Others suggested adding animations when a tree changes, for the same reason. Some suggested having unique URLs for each student, others requested easier navigation between the two modes of the tool. Both of these features have since been added to the tool. Below are some of the suggestions, paraphrased, that came from more than one student:

- *It would be nice if the subtree to be evaluated next was highlighted.* (2 students)
- *Animating the transition when a tree changes would be nice.* (2 students)
- *I missed being able to navigate between the two modes "Advanced" and "Get familiar" without having to refresh the page.* (5 students)

- *I suggest adding the username to the URL so that I don't have to re-enter my username each time I test with a new tree.* (3 students)

There were few negative responses, but one recurring pain point for the students was the installation process of the visAST client. They were instructed to use Stack, Haskell's package manager, because visAST relies on some external libraries. This process was somewhat time-consuming and some students had trouble getting it to work on their machine/operating system. However, any assignment depending on libraries outside of the standard library would generate the same kind of issues, so the problem is only indirectly related to visAST. Below are comments on the most common problems the students had:

- *Setup with Stack was a bit challenging.* (4 students)
- *Installation with Stack did not work as it should on my machine.* (4 students)
- *Installing with Stack took a long time.* (2 students)

4.3 Performance in assignment A2 compared to other assignments

In addition to the two research questions formulated in Section 4.2.2 regarding the students' thoughts on using visAST, we wanted to measure the students' performance with and without visAST to see if they did better with the tool than without. This way we approach our main hypothesis, *visAST helps students understand ASTs better*, more thoroughly. Concretely, we formulated an additional hypothesis this way:

- H_0: There is no correlation between the use of visAST and the students' performance in assignment A2.
- H_a: The use of visAST improved students' performance in assignment A2.

As discussed in Section 4.1, we did not have a control group of students that was not subjected to the use of visAST in assignment A2. We, therefore, used the students' performance in other assignments/exams as a reference point.

To assess how the use of visAST impacted the students' performance we studied the scores in the assignment with visAST compared to the scores in another compulsory assignment. We refer to the first assignment as A1 and the second one, the one from our experiment, as A2. We mapped the difference in scores in assignments A2 and A1 to how much the students used visAST in A2. Assignment A1 preceded A2 and the results of A1 are therefore not affected by the use of visAST. We did a similar analysis comparing the results of assignment A2 with the exam results, and a third one comparing A2 results to the combination of A1 and exam results. The exam, of course, came after A2 and the students' performance could therefore, in principle, have been affected by visAST use in A2. We confirmed, however, that no exam question was directly related to ASTs, parsing, or evaluation. We thus expect any effect to be negligible.

4.3.1 Results

To compare the results from the assignments and from the exam we first had to discard results from students who did not turn in both assignments. Of the students who turned in

A2 and answered the questionnaire about visAST, three students had not turned in A1 and had to be removed from this analysis. To compare A2 results with exam results we had to remove four more students who did not take the exam.

Figure 4.7: Comparing difference in score in A2 and A1 with how much visAST was used in A2. The red line shows the linear regression.

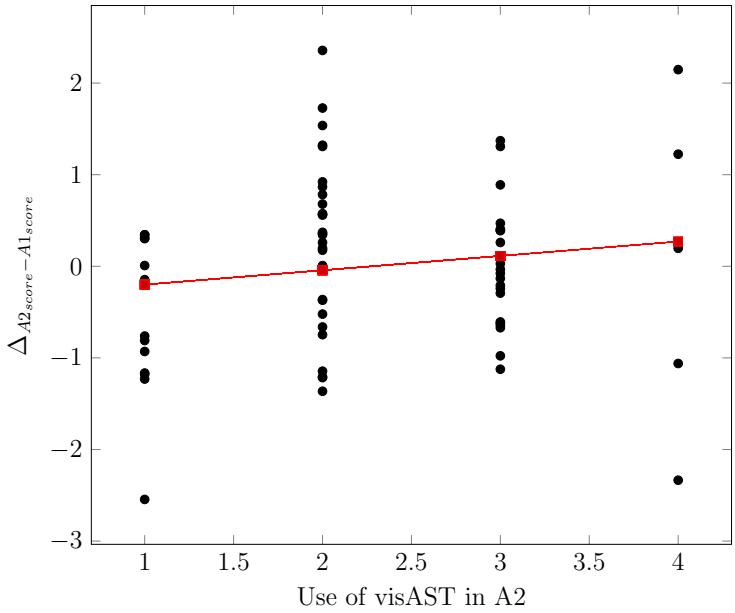


Figure 4.8: Comparing difference in score in A2 and the exam with how much visAST was used in A2. The red line shows the linear regression.

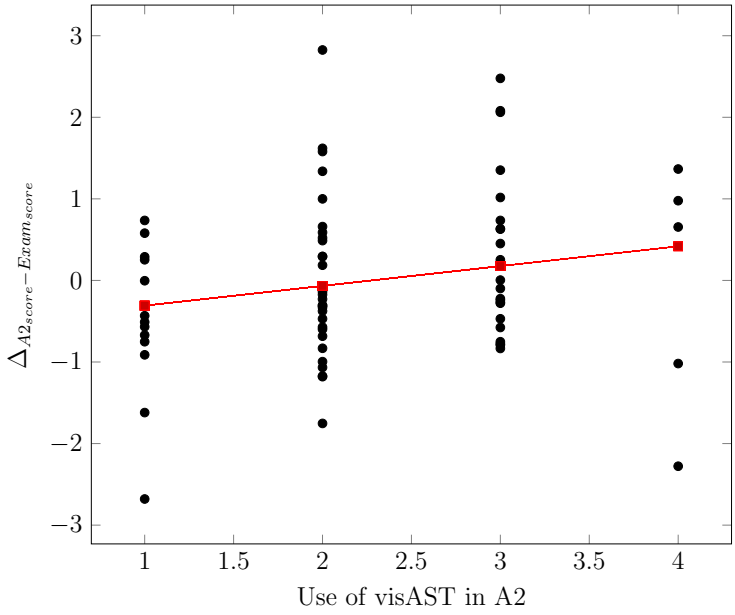
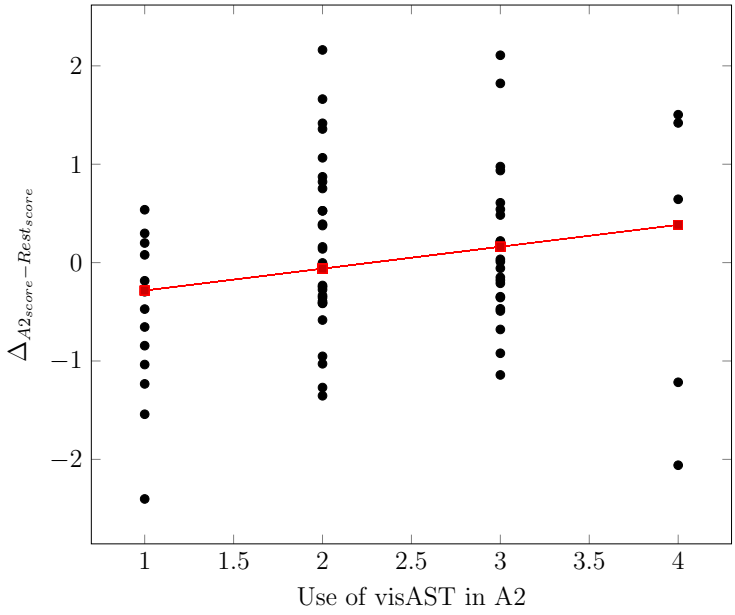


Figure 4.9: Comparing difference in score in A2 and the rest (A1 and exam scores combined) with how much visAST was used in A2. The red line shows the linear regression.



Before doing the comparison, the scores on A1, A2, and the exam were normalised. Figure 4.7 shows the difference in the normalised score between A2 and A1 compared to how much the student used visAST in A2. Figure 4.9 shows the difference in the normalised score between A2 and the combined normalised score in A1 and the exam, also compared to how much visAST was used in A2. Figure 4.8 shows the difference in the score between A2 and on the exam, also compared to visAST use in A2.

Analysis of A2 vs A1

We did a statistical analysis in R to find the significance of the observed results of the assignments A2 and A1. A summary of the analysis is shown below:

	Estimate	Std. error	t-value	Pr($>\|t\|$)
Δ_{A2-A1}	-0.3566	0.3262	-1.093	0.278
use_of_visAST	0.1567	0.1343	1.167	0.247

The p-values are 0.278 and 0.247, both much greater than the commonly used significance level of 0.05, which means we cannot reject the null hypothesis. This means that there is

not sufficient evidence against H_0 in favour of H_a, and the linear model is therefore not statistically significant [30]. The regression line has a slope of 0.1567, but since the model is not statistically significant, we can not use this with certainty. The correlation between the variables Δ_{A2-A1} and use_of_visAST is 0.14, which is smaller than 0.2. This suggests that much of the variation of Δ_{A2-A1} can not necessarily be explained by the use_of_visAST [30].

Analysis of A2 vs Exam

We did the same analysis as above for the comparison of A2 and Exam scores. The results are shown below:

	Estimate	Std. error	t-value	$\Pr(>\|t\|)$
$\Delta_{A2-Exam}$	-0.5520	0.3530	-1.564	0.1226
use_of_visAST	0.2426	0.1454	1.669	0.0998

The p-values are 0.1226 and 0.0998. Both values are not smaller than the statistical significance level, even if we choose a larger threshold of 0.1 that is commonly used. We, therefore, determine that the linear model is not statistically significant [30]. The correlation between the two variables, $\Delta_{A2-Exam}$ and use_of_visAST, is 0.1998, which is less than 0.2, suggesting that there is a relatively weak correlation between the two variables [30].

Analysis of A2 vs Rest (A1 and Exam combined)

The last statistical analysis we made included the comparison of A2 scores, and A1 and Exam scores combined. Below is the summary of the analysis computed in R:

	Estimate	Std. error	t-value	$\Pr(>\|t\|)$
$\Delta_{A2-Rest}$	-0.5066	0.3164	-1.601	0.1141
use_of_visAST	0.2226	0.1303	1.709	0.0921

Looking at the p-values, 0.1141 and 0.0921, we see that both values are not smaller than a significance level of 0.1, suggesting that the linear model is not statistically significant [30]. This is, however, closer to being significant than the other analyses. There is less than a

12 % chance that these results would be observed under the null hypothesis. The correlation between the two variables is $0.2044 > 0.2$, suggesting that there indeed is a correlation between $\Delta_{A2-Rest}$ and use_of_visAST [30]. With a slope of 0.2226, this indicates that more use of visAST helped improve the grade in assignment A2 compared to grades of the other assignment and the exam.

Chapter 5

Discussion and conclusion

In this chapter, the findings from Chapter 4 are discussed. Answers to the questionnaire are compared to the results from the performance analysis in Section 4.3 to answer the research questions and hypotheses. Our findings are compared to related research.

5.1 Discussion

Results from the questionnaire showed that the students generally liked visAST as a learning tool and that they found it intuitive and easy to use. This matches what other studies of visualisation tools used in teaching have found, as discussed in Chapter 2. The majority of the students thought visAST was moderately or more useful with regards to understanding ASTs. This view was held by the weaker students. In that respect we can confirm our first research question: *Do students find visAST helpful in understanding ASTs?*. We must, however, note that many of the students considered themselves to have a moderately good understanding of ASTs at the time of the experiment, and therefore found visAST less helpful. Nevertheless, they were in the opinion that visAST would be much more helpful for someone who is just starting to learn about ASTs and related topics. Several, ten to be exact, of the students even mentioned this explicitly in the free form question, some suggesting to adopt the tool for the first assignment. This confirms our suspicions that the tool was introduced too late.

Our results also showed that the students thought visAST helped with understanding parsing better, with over 50 % answering yes to that question. An even bigger portion of the students thought the tool was helpful for understanding expression evaluation better. This confirms our second research question: *Do students find visAST helpful in understanding parsing and evaluation better?*. A number of students commented that it was nice to have their own code visualised and to be able to verify that their parser or evaluator function worked correctly. It seems that they found it easier to verify correctness with the help of the visualisations rather than running examples on a text-based terminal. We did not ask explicitly about the students' prior knowledge of parsing and expression evaluation, but both topics rely on the understanding of ASTs. Therefore we have reason to believe that visAST would have also been more helpful with parsing and expression evaluation to someone with less prior knowledge of ASTs.

In addition to the questionnaire, we analysed in Section 4.3 the students' performance with and without visAST, to measure the real impact visAST had on the students' learning and not just what their own opinions were. None of the three analyses proved to have statistical significance, but the results are still interesting. The first analysis, comparing the difference in A2 and A1 results against how much the students used visAST, produced p-values far higher than the typical significance level of 0.05. We thus have no evidence that the use of visAST in A2 improved the students' performance compared to their performance in A1. The second analysis, comparing A2 and exam results, gave p-values close to a significance level of 0.1, but with only one of the two p-values below 0.1 and a relatively weak correlation between the variables. This suggests, as in the first analysis, that visAST use had no statistically significant impact on the performance in A2 compared to the performance on the exam. The last analysis looked at the scores in A2 compared to the combined scores in A1 and the exam. This analysis got very close to a significance level of 0.1 and also indicated that the two variables are correlated. This might mean that there was an improvement by using visAST in A2 when compared to the rest of the course combined, with a slightly positive slope. However, we cannot reject our null hypothesis in any of the models, which means we cannot be certain that the use of visAST improved students' performance in assignment A2. One interesting point to remember here is what was discussed in Chapter 2; a study [27] reporting a strong correlation between student evaluation of the visualisation tool and how they performed on the exam, i.e., the students who reported the tool beneficial also performed better on the exam. This might indicate that our students benefited more from the tool than what is shown in the performance analysis results.

Seeing the results from the performance analysis in light of what we found from the questionnaire, one may be able to explain why there was no significant improvement in performance from the use of visAST. The students claimed that the tool came too late to be very useful, and therefore we should expect no clear improvement in students' performance from using it.

The test setup was not optimal, as we had no control group. As a substitute, we used the students' performance in other tasks as controls and expected the results to be noisy. In an ideal experiment setting, we would have an assignment on ASTs, parsing, and evaluation, where one group of students would have visAST at their disposal and another, control, group would not be subjected to the use of visAST. Later, we would compare the two groups based on the results of an exam that measures knowledge of ASTs and parsing/evaluation, to find out if there were differences between the two groups. Finally, we would also measure the time invested on the assignment. The reason is as follows: A possible bias in our experiment stems from that assignment grades tend to correlate with effort instead of knowledge. Students choose themselves how much time they invest on the assignment, and therefore also how much they use the tool. Students who used the tool a lot, because it was fun or because they were asked to use it, may have gotten better results because they put more effort into the assignment. The primary reason for doing well in an assignment may have been investing more time on it; if using visAST correlates with more time spent on the assignment, then it may correlate with a better assignment grade too. A student who did not use visAST, but invested equally much time on the assignment as one who did use the tool, may have attained a similar performance boost.

5.2 Conclusion

In this thesis, the goal was to test our hypothesis that visualising ASTs (using visAST) helps students understand ASTs better. We tackled this goal by conducting an experiment on computer science students learning about ASTs. The students were subjected to the use of our visualisation tool, visAST, and we measured the usefulness of the tool, both by asking the students about their experience of using it and by analysing the students' performance with and without visAST.

The main results to remember from this thesis are that students liked to use visAST and that they found the tool moderately useful for understanding ASTs, parsing, and evaluation better. This confirms our research questions asking if students find visAST helpful for understanding these concepts better. The performance analysis indicated that visAST improved student results, but we cannot confidently confirm our main hypothesis that visAST helps students understand ASTs better. Yet, the results reported in this thesis, along with related studies, all point in the direction that dynamic visual tools benefit students. Visualisation tools used in education is an area worth pursuing more in future research, whether it be focused around ASTs or other complex topics where it may be difficult to obtain intuition based on program code only.

5.3 Future work

This thesis gives an indication that visualisation tools like visAST are good for learning structural concepts like ASTs. The thesis also gives guidance on what aspects and features are useful in a visualisation tool for teaching programming language material. The experimental results, however, do not allow us to make conclusive determinations about the effectiveness of visAST for learning. Therefore, future research should do further experiments on students to seek statistically significant results to that tools like visAST improve students' understanding of ASTs. Such experiments should include a control group, as described in Section 5.1. The target group should be students who are just learning about ASTs, as the results of this thesis strongly indicated that students with less experience with ASTs would benefit more from the tool.

Further, feature requests from feedback received in the experiment could be implemented and tested on users. The most interesting features to examine are animating the transition between two program states, i.e., between two trees, or highlighting the part of the tree which is currently being processed.

Bibliography

[1] Code club. https://codeclub.org/en. Accessed: 2019-03-27.

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

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

[4] Kids & code. http://www.kidsandcode.org. Accessed: 2019-03-27.

[5] Scratch programming language. https://scratch.mit.edu. Accessed: 2019-03-27.

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

[7] Ragnhild Aalvik. aaalvik/visast-backend: Second release of visAST backend, May 2019.
URL: https://doi.org/10.5281/zenodo.2765087.

[8] Ragnhild Aalvik. aaalvik/visast-client-example: Second release of visAST client, May 2019.
URL: https://doi.org/10.5281/zenodo.2765130.

[9] Ragnhild Aalvik. aaalvik/visast-frontend: Second release of visAST frontend, May 2019.
URL: https://doi.org/10.5281/zenodo.2765193.

[10] 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, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-69924-8.

[11] 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. doi: 10.1109/ICALT.2009.158.

[12] Ronit Ben-Bassat Levy, Mordechai Ben-Ari, and Pekka Uronen. The jeliot 2000 program animation system. *Computers & Education*, 40:1–15, 01 2003. doi: 10.1016/S0360-1315(02)00076-3.

[13] 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. ISSN 0097-8418. doi: 10.1145/191033.191121.
URL: `http://doi.acm.org/10.1145/191033.191121`.

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

[15] Patrick Dubroy, Saketh Kasibatla, Meixian Li, Marko Röder, and Alex Warth. Language hacking in a live programming environment. In *Presented at LIVE 2016*, USA, 2016.
URL: `https://ohmlang.github.io/pubs/live2016/`. HARC / Y Combinator Research.

[16] Torbjörn Ekman. Extendj exploring tool. `https://extendj.org/visualizer.html`. Accessed: 2019-03-12.

[17] Jeremy Gibbons. Datatype-generic programming. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. ISBN 978-3-540-76786-2.

[18] G. Gray and D. Duffin. SIF: Learning styles theme final report: Project report 2007-2009. Unpublished report, 2011.

[19] 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. ISBN 978-1-4503-1868-6. doi: 10.1145/2445196.2445368.
URL: `http://doi.acm.org/10.1145/2445196.2445368`.

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

[21] 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.

[22] R. J. Kapadia. Teaching and learning styles in engineering education. In *2008 38th Annual Frontiers in Education Conference*, pages T4B–1–T4B–4, Oct 2008. doi: 10.1109/FIE.2008.4720326.

[23] Barbara A. Kitchenham and Shari L. Pfleeger. Personal opinion surveys. In Forrest Shull, Janice Singer, and Dag I.K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, chapter 3, page 90. Springer-Verlag London, London, 2008.

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

[25] M. Mernik and V. Zumer. An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68, Feb 2003. ISSN 0018-9359. doi: 10.1109/TE.2002.808277.

[26] 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. ISSN 0097-8418. doi: 10.1145/782941.782998.
URL: `http://doi.acm.org/10.1145/782941.782998`.

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

[28] E. Pasternak, R. Fenichel, and A. N. Marshall. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B B)*, pages 21–24, Oct 2017. doi: 10.1109/BLOCKS.2017.8120404.

[29] Gordon D Plotkin. A structural approach to operational semantics. 1981.

[30] Selva Prabhakaran. Linear regression. `http://r-statistics.co/Linear-Regression.html`. Accessed: 2019-02-18.

[31] Kanasz Robert. Visualization and comparison of sorting algorithms in c#. `https://www.codeproject.com/Articles/132757/Visualization-and-Comparison-of-sorting-algorithms`, dec 2010. Accessed: 2019-05-02.

[32] 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.
URL: `http://dl.acm.org/citation.cfm?id=369340.369325`.

[33] 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. Contribution: organisation=ohj,FACT1=1.

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

Appendix A

Original assignment from the experiment

Oblig 2 – INF122, H-2018

Hvordan komme igang / hvordan levere

- Oppgaven skal leveres på Mitt UiB under Oppgaver, samme sted som du fant oppgaveteksten. Løsningen skal bestå av én zip-fil med navn på formen FORNAVN.ETTERNAVN.zip.
- zip-filen skal inneholde tre filer: Oblig2.hs, Main.hs (fra VisAST/app/) og Del3.pdf.
  Alle tre filene skal ha ditt Mitt UiB-brukernavn i en kommentar i toppen.
- Du skal klone (eller laste ned) dette git-repoet: github.com/aaalvik/Oblig2-INF122
- Innleveringsfristen er **Tirsdag 27. november, kl. 23.59**
- Når oppgaven ber deg om å lage en funksjon med gitt navn og type, så er det **viktig** at du ikke endrer på navnet/typen. Du kan selvfølgelig lage hjelpefunksjoner med andre navn og typer

Oversikt

OBS: Før du begynner skal du klone git-repoet nevnt over, følg instruksene i README-en på github.
All kode skal skrives inn i de gitte filene og mappene i din kopi av dette repoet.

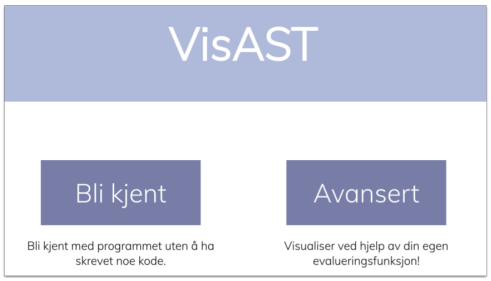
Vi skal jobbe med et lite språk som består av mattestykker i prefiksnotasjon (dvs operatoren først) og if-then-else uttrykk. Det er ingen parenteser i språket. Grammatikken til språket ser slik ut:

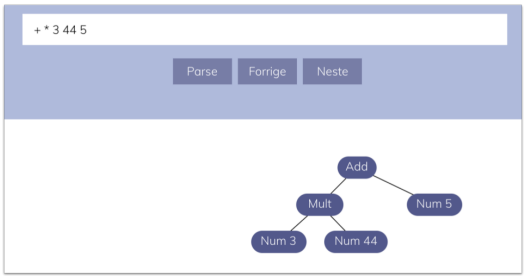
```
S ::= * S S
    | + S S
    | - S                    unær minus
    | if S then S else S
    | number_or_digit        ett eller flere siffer
```

Ifølge grammatikken kan eksempler på uttrykk i språket være:

```
+ 3 4                    dvs 3 + 4 i vanlig infiksnotasjon

* 6 7                    dvs 6 * 7 i vanlig infiksnotasjon

- 3                      unær minus, altså minus 3.

if 2 then + 3 4          sjekker om if-condition (her tallet 2) er ulik 0. Er den
      else 6             ulik 0 går man til then-branchen, ellers else-branchen.

* + 3 4 5                dvs (3 + 4) * 5 i vanlig infiksnotasjon. Man trenger
                         altså ikke parenteser i språket.

+ if 2 then + 3 4        dvs (if 2 then (3+4) else 6) + 8. Her blir if-condition
        else 6 8         true, returnerer 7, og resultatet av hele uttrykket blir
                         7 + 8 = 15.
```

Om visAST





I løpet av oppgaven vil vi bruke programmet visAST, som er et verktøy for å visualisere abstrakte syntakstrær. visAST er under utvikling, og dere vil bli de første til å teste dette i undervisningssammenheng. Vi ønsker derfor tilbakemelding på hvordan dette var å bruke, så tenk gjennom hvordan du opplever det underveis.

Sammendrag av oppgaver:
(Du finner nøyere forklaring og beskrivelse av hver oppgave lenger ned.)

1.1 Implementer en parser for språket beskrevet over.

1.2 Print abstrakte syntakstrær – Expr – på en pen måte.

1.3 Implementer en small-step evaluator for språket.

1.4 Lag et program som leser inn kommandoer fra bruker og utfører kommandoen.

2.1 Utvid 1.4 til å kunne visualisere ved hjelp av visAST.

2.2 Svar på spørsmål om visAST.

3.1 Finn typen til en funksjon ved hjelp av reglene for typeinferens og unifikasjon.

Del 1 – Parse og prettyprint expressions

I hele del 1 (oppgave 1.1-1.4) skal du jobbe i filen Oblig2.hs

Oppgave 1.1 – parse :: String → Expr

Du skal implementere en parser som oversetter strenger i språket til et abstrakt syntakstre av typen Expr:

```
data Expr
   = Num Int              et tall, feks “2” eller “222”
   | Add Expr Expr        et pluss-uttrykk, feks “+ 4 5”
   | Mult Expr Expr       et gange-uttrykk, feks “* 4 5”
   | Neg Expr             et negert uttrykk, feks “- 6” eller “- + 4 5”
   | If Expr Expr Expr    et if-uttrykk, feks “if 2 then 3 else 4”
   deriving (Show, Read)
```

Du bør begynne med en tokenizer (tokenize :: String -> [String]), som kan splitte inputstrengen i meningsfulle tokens. Tokens i språket ser man av grammatikken at er en av operatorene “+-*”, nøkkelordene “if”, “then” og “else”, i tillegg til tall, som består av ett eller flere siffer.

Videre anbefales det å lage en hjelpefunksjon parseExpr :: [String] → (Expr, [String]), som kan kalles fra parse. Parse bør da først tokenize inputstrengen til en liste med tokens (Strings) som kan sendes til parseExpr. Følg reglene i grammatikken (på side 1) slik at parseren oppfører seg som under.
NB: Denne parseren er mye enklere enn den fra Oblig 1, da alle operatorer/keywords kommer prefiks i stedet for infiks.

Eksempler på parsing av uttrykk:

```
parse “+ 4 5”                    == Add (Num 4) (Num 5)
parse “* + 5 6 70”               == Mult (Add (Num 5) (Num 6))  (Num 70)
parse “*+5 6 70”                 == Mult (Add (Num 5) (Num 6))  (Num 70)
parse “- 4001”                   == Neg (Num 4001)
parse “* 8 + 4 - 52”                    == Mult (Num 8) (Add (Num 4) (Neg (Num
52)))
parse “if 0 then 4 else + 3 3”   == If (Num 0) (Num 4) (Add (Num 3) (Num 3))
parse “3 + 4”                    == error “Illegal expression”
```

Legg merke til i tredje eksempel at operatorene kan stå inntil hverandre (også operator og tall) uten at det gjør noen forskjell. Dette må håndteres av tokenizeren.

Oppgave 1.2 – prettyPrint :: Expr → IO ( )

Du skal implementere funksjonen prettyPrint, som tar inn et expression (abstrakt syntakstre av typen Expr) og printer dette til terminal på en “pen” måte. For hver node i treet Expr skal du skrive ut navnet på noden, og så barna på hver sin linje under, indentert én tab (evt fire spaces) lenger inn. Du må altså holde styr på antall tabs man skal indentere på hvert nivå.
Tips: Lag en hjelpefunksjon `prettyPrint' :: Expr -> Int -> IO ()` som har med som parameter antall tabs man skal indentere så langt.

Eksempel 1 – prettyPrint (parse “+ 4 5”) skal printe:

```
Add
  Num 4
  Num 5
```

Eksempel 2 – prettyPrint (parse “* 8 * - 9 1”) skal printe:

```
Mult
  Num 8
  Mult
    Neg
      Num 9
    Num 1
```

Eksempel 3 – prettyPrint (parse “if * 2 3 then 10 else 20”) skal printe:

```
If
  Mult
    Num 2
    Num 3
  Num 10
  Num 20
```

Frivillig oppgave – Test parseren din med visAST

Du så kanskje at prettyPrint ikke gjorde trærne så veldig pretty likevel, derfor kan du nå teste parseren ved hjelp av visAST, som tegner trærne slik man ofte gjør på tavla. Les beskrivelsen om visAST under Del 2 lenger ned før du begynner. Fremgangsmåte (også delvis forklart i README):

1. Lim din implementasjonen av *parse* inn i filen VisAST/app/TestParser.hs.

2. Åpne terminal (den i VS Code e.l. vil ikke fungere, men vanlig terminal/cmd)
3. Naviger til mappen VisAST/
4. Skriv inn **stack build** i terminalen (dette må gjøres på nytt hver gang du gjør endringer i parseren)
5. Skriv inn **stack exec TestParser-exe** i terminalen. Du vil nå bli bedt om å skrive inn mitt.uib brukernavnet ditt og et uttrykk i språket, deretter åpner visAST seg automatisk i en browser. I browseren: klikk "Avansert", fyll inn brukernavn og du får opp treet ditt.

Oppgave 1.3 – takeOneStep :: Expr → Expr

Implementer funksjonen takeOneStep, som tar inn et abstrakt syntakstre (Expr) og utfører ett evalueringssteg før den returnerer. Det vil si, funksjonen skal ta det minste steget den kan på et Expr men fortsatt komme nærmere resultatet (en verdi). **Når et uttrykk har blitt til et Num num, så er det en verdi**, og man kan ikke ta flere evalueringssteg.

Regler for evaluering:

1. Num x evaluerer til seg selv, feks Num 5 er en verdi og kan ikke evalueres mer.
   Feks: `takeOneStep (Num 5) == Num 5`

2. Add x y skal først evaluere ferdig x til en verdi, ett steg om gangen.
   Dersom x er en verdi skal man på samme vis evaluere y til en verdi, ett steg om gangen.
   Dersom både x og y er verdier, skal man plusse dem sammen til et Num og returnere dette.

3. Mult x y skal som i Add først evaluere x til en verdi, deretter y, deretter gange x og y sammen.

4. Neg x skal først evaluere x til en verdi – Num n – deretter negere n (dvs gange n med minus 1)

5. If eCond eThen eElse skal som over først evaluere eCond (altså conditionen til if-setningen) til en verdi, ett steg om gangen.
   Dersom eCond er en verdi skal man sjekke om den er ulik 0 (fordi vi kun jobber med Ints og ikke boolske verdier, så definerer vi alt ulik 0 som True). Er eCond ulik 0 returnerer man eThen, ellers returneres eElse, uten å evaluere videre.

*Et uttrykk som + 4 * 5 6 evaluerer til slutt til 34, men dette kan deles opp i to steg:*

- *Gange sammen 5 og 6 til 30, slik at man får + 4 30*
- *Plusse sammen 4 og 30 til 34*

Eksempler:

```
// x er ikke en verdi
takeOneStep (Add (Add (Num 4) (Num 4)) (Num 2)) == Add (Num 8) (Num 2)

// x er en verdi, y er ikke
takeOneStep (Add (Num 4) (Mult (Num 5) (Num 6))) == Add (Num 4) (Num 30)

// både x og y er verdier
takeOneStep (Add (Num 4) (Num 2)) == Num 6

// eCond er ikke en verdi:
takeOneStep (If (Add (Num 1) (Num 1))  (Mult (Num 2) (Num 3))  (Num 5)) ==
       If (Num 2)  (Mult (Num 2) (Num 3))  (Num 5))

// eCond er en verdi, og ulik 0:
takeOneStep (If (Num 3)  (Num 4)  (Mult (Num 2) (Num 7))) == Num 4

// eCond er en verdi, og lik 0:
takeOneStep (If (Num 0)  (Num 4)  (Mult (Num 2) (Num 7))) == Mult (Num 2) (Num 7)

// når til slutt en verdi (et Num) og man kan ikke evaluere lenger
takeOneStep (Mult (Num 2) (Num 7))      == Num 14
takeOneStep (Num 14)                           == Num 14
```

Oppgave 1.4 – mainStep :: Expr → IO ( )

Nå skal du implementere funksjonen mainStep som skal ta inn kommandoer fra bruker, printe ting til skjerm samt skrive til/lese fra fil. mainStep skal begynne med å prettyPrinte expr gitt som parameter. Deretter skal man lese inn input fra bruker, og gjøre ulike kommandoer basert på hva bruker skrev inn. Funksjonen skal kjøre videre helt til bruker velger å avslutte med kommandoen q.

new <expression>	Tar inn et nytt expression og parser det. Du skal parse alt som kom etter nøkkelordet *new*, og kalle mainStep videre med dette nye expr. Feks "new + 4 5" skal kalle parse på "+ 4 5".
ENTER-KLIKK	Dette skal skje dersom bruker trykker på enter-knappen uten å ha skrevet noe mer. Her skal du ta ett evalueringssteg (med takeOneStep fra 1.3) på expr som ble gitt som parameter til mainStep, og så fortsette programmet (kalle mainStep på nytt med det evaluerte expr).
w <filnavn>	Skriv expr til en fil med navn <filnavn>.txt, og fortsett programmet. For å gjøre uttrykket om til en streng kan du kalle **show** på det før du skriver til filen.
r <filnavn>	Les expr som er lagret i filen <filnavn>.txt, og kall mainStep videre med det nye uttrykket. Hvis du kalte show på uttrykket før du skrev til filen så kan du nå kalle **read str :: Expr** på str lest fra filen, og få ut et Expr.
q	Avslutt programmet.

Eksempel på kjøring:

```
> mainStep (Add (Add (Num 4) (Num 4)) (Num 2))
Add
   Add
      Num 4
      Num 4
   Num 2
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
[Bruker trykket ENTER-knappen]
Add
   Num 8
   Num 2
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
w minFil.txt
Skrev uttrykket til fil.
Add
   Num 8
   Num 2
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
[Bruker trykket ENTER-knappen]
Num 10
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
new * 1 2
Mult
   Num 1
   Num 2
* Hva vil du gjøre? (new expr, ENTER, w filnavn, r filnavn, q)
q
```

Del 2 – Visualiser abstrakte syntakstrær

Du skal nå jobbe i mappen VisAST/app, i filen Main.hs. Før du begynner på oppgaven skal du lime inn de fire funksjonene du implementerte i Del 1:

- parse
- prettyPrint
- takeOneStep
- mainStep
- + eventuelle hjelpefunksjoner

I denne oppgaven må du bruke **stack** for å kompilere og kjøre programmet, som beskrevet i README. Du vil ikke kunne teste ting i ghci, og terminalen i feks VS Code vil mest sannsynlig heller ikke fungere (bruk vanlig terminal/iterm/command prompt etc)

Oppgave 2.1 – utvid mainStep-metoden i Main.hs til å kunne starte visAST
Nå skal du bruke visAST til å få visualisert trærne på en penere måte enn med prettyPrint. Før du begynner å kode bør du gjøre deg kjent med visAST under "Bli kjent" her, der du kan få visualisert uttrykk du skriver inn uten å ha skrevet noe kode.

Etter å ha gjort deg kjent med visAST skal du utvide mainStep (som du har limt inn i VisAST/app/Main.hs) med en kommando som starter visAST. visAST vil da visualisere trær **ved hjelp av din egen takeOneStep fra 1.3.** Utvid mainStep med følgende kommando:

vis <brukernavn>	<brukernavn> skal være mitt.uib brukernavnet ditt – på formen abc123. Du skal så evaluere expr slik at du får en liste med expressions – alle "tilstandene"/evalueringsstegene til expr frem til det blir en verdi. Det gjør du ved å kalle eval-funksjonen (som bruker din egen takeOneStep). Til slutt skal du bruke den importerte funksjonen **visualise :: String -> [Expr] -> IO ()** til å starte visAST. visualise tar inn brukernavn og listen med expressions og vil åpne en browser for deg på nettsiden til visAST. På nettsiden velger du "Avansert" -> Skriver inn brukernavnet du brukte, og så vil du få visualisert listen med evalueringssteg som du sendte til visualise. Det første treet du ser er startuttrykket, og du kan steppe frem og tilbake med knappene og se alle evalueringsstegene frem til du står igjen med en verdi (et Num).

Oppgave 2.2 – Svar på spørsmål om visAST
Svar på spørsmålene her: goo.gl/forms/nELqqXVZ5lVnedCp1**.**
Du skal ikke levere inn noen fil i denne oppgaven, kun fylle ut skjemaet i lenken.

Del 3 - Finn typen til en funksjon

Oppgi svaret i en fil del3.pdf som du oppretter selv. Denne skal inneholde hele utledningen.

Oppgave 3.1 - Finn typen til \x -> \y -> \z -> z (x y)
Bruk reglene for typeinferens og unifikasjon oppgitt på siste side til å finne typen til funksjonen.
Du kan løse oppgaven for hånd og ta bilde/scanne utledningen, eller skrive direkte inn på pc. Husk at du må vise hele utledningen.

VEDLEGG – Regler for typeinferens og unifikasjon

Typeinferens – Hindley-Milners algoritme

(t1, konstant)	$E(\Gamma \mid c :: \tau)$	$= \{ \tau = \theta \}$	der θ er typen til konstanten (feks Int / Bool etc, en kjent type)
(t2 - variabel)	$E(\Gamma \mid x :: \tau)$	$= \{ \tau = \theta \}$	der θ = lookup(x, Γ) for variabel x
(t3 - funksjonskall)	$E(\Gamma \mid f\ x :: \tau)$	$= E(\Gamma \mid x :: a) \cup E(\Gamma \mid f :: a \rightarrow \tau)$	
(t4 - abstraksjon)	$E(\Gamma \mid \backslash x \rightarrow y :: \tau)$	$= \{ \tau = a \rightarrow b \} \cup E(\Gamma, x :: a \mid y :: b)$	
(t5 - tuppel)	$E(\Gamma \mid (ex1, ex2) :: t)$	$= \{ t = (a, b) \} \cup E(\Gamma \mid ex1 :: a) \cup E(\Gamma \mid ex2 :: b)$	
(t6 - ikke-tom liste)	$E(\Gamma \mid x{:}xs :: t)$	$= \{ t = [a] \} \cup E(\Gamma \mid x :: a) \cup E(\Gamma \mid xs :: [a])$	
(t7 - tom liste)	$E(\Gamma \mid [] :: t)$	$= \{ t = [a] \}$	

Forklaringer:

1. ***Γ*** *står for environmentet, altså typene vi kjenner fra før*
2. ***E(Γ | expr :: myType)*** *kan leses slik:*
   *"Et uttrykk **E** som inneholder et environment **Γ** hvor **expr** har typen **myType**"*
3. *Eksempel på t1-uttrykk:* **$E(\Gamma \mid 13 :: \tau) \mid\mid E(\Gamma \mid$ "Hei" $:: \tau)$**
   Vi kjenner typen til konstanten 13, og kan sette den rett inn: **$= \{ \tau =$ Int $\}$**

Unifikasjonsregler

(u1)	E,t = t	⇒ E		
(u2)	E, f (t1...tn) = f (s1...sn)	⇒ E, t1 = s1, ..., tn = sn		
(u3)	E, f (t1...tn) = g(s1...sm)	⇒ ⊥ (occurs check)	der	f /= g ∨ n /= m
(u4)	E, f (t1 ...tn) = x	⇒ E, x = f (t1...tn)		
(u5)	E, x = t	⇒ E[x/t], x = t	der	x not∈ Var(t)
(u6)	E, x = t	⇒ ⊥	der	x ∈ Var(t)

Appendix B

Original questionnaire from the experiment

Spørsmål om visAST - oblig 2, INF122

Utfylling av dette skjemaet er svar på oppgave 2.3. Svarene dine teller ikke på poengsummen i obligen, vi sjekker kun om du har svart.

visAST er et program under utvikling, og er ment å være et undervisningsverktøy for å bedre forstå hvordan abstrakte syntakstrær (AST-er) fungerer og ser ut. Vi trenger derfor tilbakemelding på hvordan det har vært å bruke, om det hjalp noe med forståelsen etc.

Takk for din tilbakemelding! Dette hjelper både vår research og potensielt fremtidige studenter.

*Må fylles ut

1. **Ditt Mitt UiB brukernavn (på formen abc123)** *

_____

2. **Hvor godt forsto du abstrakte syntakstrær (AST-er) før denne obligen?** *
   Markér bare én oval.

	1	2	3	4	5	
Forsto lite	◯	◯	◯	◯	◯	God kontroll

3. **Hvor mye brukte du visAST til denne obligen?** *
   Markér bare én oval.

	1	2	3	4	5	
Så lite som mulig	◯	◯	◯	◯	◯	Masse

4. **Hvor nyttig var visAST for din forståelse av abstrakte syntakstrær (AST-er) i denne obligen?** *
   Markér bare én oval.

	1	2	3	4	5	
Ikke nyttig	◯	◯	◯	◯	◯	Veldig nyttig

5. **Hvor nyttig tror du visAST ville vært for din forståelse første gang du ble introdusert for AST-er?** *
   Markér bare én oval.

	1	2	3	4	5	
Ikke nyttig	◯	◯	◯	◯	◯	Veldig nyttig

6. **Hjalp visAST med å forstå parsing bedre?** *
   Markér bare én oval.

   ◯ Ja
   ◯ Nei

7. **Hjalp visAST med å forstå evaluering av uttrykk bedre?** *
   Markér bare én oval.

   - ◯ Ja
   - ◯ Nei

8. **Vi ønsker dine tanker om visAST, med dine egne ord. Hva var nyttig, hva var ikke? Hva var lett, hva var ikke? Har du noen forslag til forbedringer? Andre tanker?**

_____

_____

_____

_____

_____