Experimental Investigation of the Influence of the Introductory Programming Paradigms

HCC · Human-Centered Computing

Monday 28th February, 2011

Johan Sejr Brinch Nielsen

Email:	zerrez@diku.dk
Cpr.:	260886-2547
Supervisor:	Jakob G. Simonsen, Mikkel Rønne

Dept. of Computer Science,
University of Copenhagen

Abstract. I design an experiment with focus on novice programmers and their ability to read program code. I execute this experiment using first year students and their tutors as participants. Specifically, the experiment consists of 15 tasks and involved 12 SML students, 4 SML tutors and 3 Python students.

I provide the necessary tools for creating a similar experiment, specifically a graphical user interface to assist code and task presentation, a processing framework for the resulting data that includes a standard fixation filter for eye coordinates and generation of 3 different visualisations to assist interpretation. Specifically, line transitions with and without time, and token density.

During the experiment the developed graphical user interface is used to display program code and collect participant feedback. The data is hereafter processed by the aforementioned framework and the resulting visualisations are presented.

Due to the limitations of this project, I do not interpret these visualisations. I do however provide the data needed for such later analysis.

Contents

1 Introduction **4**
1.1 Related Work 4
1.2 Expectations of the Reader 5
1.3 Terminology 5

2 Experiment **6**
2.1 Participant session 6
2.2 Tasks 7
2.3 Participants 7
2.3.1 Invitations and scheduling 7

3 Software and Equipment **8**
3.1 The Eye-tracker 8
3.2 The graphical user interface 9
3.2.1 Design 9
3.2.2 Recovery 10
3.2.3 Backend 10

4 Data processing **11**
4.1 Error types 11
4.2 Fixation filtering 12

5 Visualisations **12**
5.1 Line Transitions 12
5.2 Token Density 13
5.3 Comparisons 14

6 Future Work **15**

A Eye-tracker output **18**
A.1 Header 18

B Tasks **18**

C Experiment Source Code **21**
C.1 Highlighting 21
C.2 Standard ML 21
C.3 Python 25

1 Introduction

There is an ongoing discussion regarding the choice of programming language in introductory courses at universities. Some rely on functional programming languages (e.g. Standard ML, Lisp), others prefer object-oriented languages (e.g. Java, Python), and others again low level languages (e.g. C, C++).

In this project, I aim to provide evidence to back up the arguments in this discussion. A necessity for this, is a better understanding of how students read programming code. Several methods for achieving such exist, e.g. the Think Aloud Method[8], eye-tracking and video recordings. Eye-tracking in particular has some interesting strengths:

Position: It tracks the position being read. This shows which parts of the code the programmer is visiting for her understanding.

Intuitive: It's a simple and intuitive way of collecting behavioural information.

Practical: The output data is easy to work with in an automated manner and collecting it does not rely on manual labour. Furthermore, DIKUA has a high quality eye-tracker at it's disposal.

On the other side, there are some clear limitations of this method:

Noise: There will always be traces of noise in the output. This can be relaxed, but not completely cleaned (see section 4 on page 11).

Dead zones: If the participant doesn't stay within an acceptable distance and angle, the eye-tracker will and stop reporting valid results.

Interpretation: The programmer's intent of the recorded actions are unknown.

In this project, I present an experiment based on eye-tracking mainly because of its ability to produce volumes of data with little manual labour. Participants are drawn from the DIKU introductory courses "Introduktion til programmering" (Introductory programming) and "Grundlæggende datalogi" (The basics of computer science). The former relies on the functional language Standard ML (SML) and the latter object-oriented Python.

1.1 Related Work

In 2006, Uwano et al. [14] looked at the "code review" process, by examining how programmers use line transitions to scan through code.

In 2007, Da Cunha and Greathead [2] investigated the "code review" process and how the review was affected by the reviewer's personality.

In 2009, Binkley et al. [1] used speed and accuracy of program manipulations in a comparison of the popular naming conventions `camelCase` and `under_score`. In a follow-up study of 2010, Sharif and Maletic [11] used an eye-tracker to provide further empirical data for such comparison.

1.2 Expectations of the Reader

The reader should have a basic understanding of Human-centered computing and the experimental investigation associated with this field(see Dumas and Redish [4]).

In order to understand the tasks used in the experiment and to interpret the resulting visualisations a basic knowledge of the programming languages Standard ML and Python is necessary (see Paulson [9] and Downey et al. [3] respectively). Specifically, an understanding of functional and imperative programming paradigm as well as the two language syntaxes.

The section on scheduling exploits basic optimisation methods from the field of operations research. This section is not critical to the understanding of the results, but in order to reuse the scheduling procedure an understanding of optimisation problems (such as what is introduced in Taha [12]) is mandatory.

1.3 Terminology

A few words aught to be in place, before continued reading:

The experiment is the full experiment conducted, consisting of many sessions and several participants.

A participant describes someone who takes part in the experiment.

A session describes the individual process of a participant taking part in the experiment.

The supervisor is a person with knowledge of the equipment used who greets participants and oversees sessions.

2 Experiment

The purpose of the conducted experiment is to better understanding how students read code. Specifically, it would be interesting to see which parts of the code a student reads to learn its meaning. Not merely which lines, but also which tokens, in which order and at what density and velocity.

I have designed and executed an experiment which requires the participant to examine program code. The intent is to monitoring the participant during the experiment and hopefully gain a better understanding of the process from the resulting data.

The experiment consists of 15 tasks where each task consists of a question and a hint regarding the presented program code. The hint is solely to help the student identify the relevant part of the code.

As an example, task 1 reads (Danish):

> Hvor mange elementer er der i listen 'instruments'?
> Hint: Se linie 1

Furthermore, each task has a *relevant code section*. This meta-data is not displayed to the participant but is required for later visualisation of eye-fixations along with relevant code lines. For task 1 this range reads $0 - 8$, meaning the question can be answered by reading lines 0 through line 8 (both included).

2.1 Participant session

The following steps describe the procedure of executing a session for a specific participant:

Greeting the student: A short small-talk introduces the participant to the Experience Lab room. It is stressed, that the experiment focuses on *the process* of task solving and not on the actual answers. This relaxes the participant and encourages her to not give up easily on the difficult tasks. The participant signs a formal agreement allowing her data to be collected and used in the project.

Calibration: This process requires the participant to follow a circle moving across the screen through a total of 9 points. The calibration is overseen by the supervisor.

Questionnaire: A questionnaire is filled out by the participant. She is asked to answer questions on age, gender, education and programming experience.

Example task: The first task presented is an example and is already answered. It serves to introduce the participant to the user interface (see section 3.2 on page 9) and give her an understanding of the expectations of an answer.

Tasks: The participant is presented with unknown code and is given a number of tasks to solve. The supervisor resides in the same room in case something unexpected should happen. When the participant has finished all tasks to the best of her ability, the GUI quits and the session is ended.

2.2 Tasks

The tasks used in the experiments attempts to involve most areas of the programming language, that can be expected to be known by a 1st semester student. Two set of tasks are used, one for Standard ML students and one for Python students. Even though the two sets are very similar, but do vary occasionally. An example of such variation can be seen in task 9 where the questions read:

SML: Hvad tester 'case' i funktionen 'update'?

Python: Hvad tester 'if'-sætningen i funktionen 'update'?

Here the difference is whether the question involves a `case` expression (SML) or an `if` statement (Python).

The tasks are written in Danish, as all participating students are native Danish speakers.

2.3 Participants

The participants are categorised in four groups, based on introductory paradigm and experience:

SML students: First year students being taught Standard ML as their first language.

SML tutors: Experienced students teaching the SML students.

Python students: First year students being taught Python as their first language.

Python tutors: Experienced students teaching the Python students.

With 200 SML students as candidates for the experiment, finding participants seems easy. As it turned out, the attention of students is hard to catch. After contacting first a random subset, and later all of them, I managed to get a total of 13 first year participants from the course "Introduktion til programmering"..

2.3.1 Invitations and scheduling

Inviting students to participate is handled using Doodle (`http://doodle.com`). Each student receives an invitation email with a unique Doodle link inside. The Doodle site allows the individual student to fill out time slices that fits her schedule.

Each Doodle link is unique to prevent students from seeing each other's reply. However, this implies a higher risk of overlapping answers and a need for solving these when making the combined schedule for all the sessions.

I solved this problem by formulating it as a binary optimisation problem and solving this with CPLEX. The object-function is defined as the count with a time slot assigned. The constraints guarantee at most 1 participant per time slot and that a participant's time slot fits her preferences.

The resulting formulation takes the following form:

Maximise:

$$\sum x_{s,t}$$

Subject to:

$$\text{for all s :} \qquad \sum_t x_{s,t} \leq 1$$

$$\text{for all s,t:} \qquad x_{s,t} = 0 \text{ if } s \text{ didn't select } t$$

Where the variable $x_{s,t}$ is a binary variable taking the state 1 iff the student s is selected for the time slot t. Hence, the object function is the count of occupied time slots. The first constraint ensures that each student is assigned at most 1 time slot, while the second ensures that time slots which are incompatible with student preferences are omitted from the chosen solution.

3 Software and Equipment

In this section, I describe the equipment and software used in the experiment. In section 4 on page 11, the methods and software used for processing the resulting data is presented.

3.1 The Eye-tracker

The eye-tracker used is an X120 from Tobii Technology configured to run at a rate of 60 Hz (see Tobii [13]). The eye-tracker is mounted on a table records eye movement using reflections from near infrared light, eliminating the need for goggles or other disturbing headwear.

Figure 1: Stock image of the X120 tobii eye-tracker (www.tobii.com).

The eye-tracker dumps the recorded data as tab separated values (TSV) proceeded by meta information about the recording. An example file header can be seen in section A.1 on page 18. Below is a sample of the TSV output (only relevant columns):

```
... Timestamp ... GazePointX  GazePointY ...
... 242       ... 559         384        ...
... 259       ... 529         322        ...
... 276       ... 532         337        ...
```

As shown by the "Timestamp" column, sampling occurs at 17 millisecond intervals, which matches the expectation of sampling at 60 Hz ($\frac{1}{60} \approx 0.0166$).

The software package from Tobii Technology includes fixation filters, though a specification of the algorithms used are hard to come by. In order to present reproducible results, I have implemented and used a standard fixation filter instead. Hence, the presented framework also works on eye-trackers with no fixation filters available.

3.2 The graphical user interface

The minimum requirements for a graphical user interface (GUI) that can be used during the experiments are:

- Syntax highlighting for both Standard ML and Python. This is an important feature to match popular editors and the expectations of the participants.
- Show both the code, the current question and an input box for the provided answer.
- Provide complementary data for tracking eye positions in the source code. If the code cannot be contained on screen in its entirety an internal position is needed (e.g. a scroll bar position).
- Measure the task completion time.
- Run on both Linux and Windows (the software is developed and tested on Linux while the eye-tracker setup uses Windows).

I don't know of any GUI which provides such configuration. A standard coding editor could easily provide syntax highlighting, but not tracking of internal position and completion time. Instead, a newly developed specialised GUI is used.

3.2.1 Design

The GUI is designed with simplicity in mind. It is important, that a new participant is able to quickly understand and navigate it.

It consists of four parts (see figure 2 on the following page for an example):

1. A read-only syntax highlighted text-box presents the code on the left. The scroll bar allows for vertical navigation, while line numbers helps give the feel of a regular source code editor.
2. A read-only text-box displays the task at hand at the upper right.
3. A writable text-box allows for answer submission at the lower bottom.

4. A submit button right under the writable answer text-box.

Keeping the source code on the left and other task-specific information on the right gives a clear cut between the two. This makes it easier to distinguish eye movement in the source code from unrelated eye movement (e.g. reading the task description or writing the answer).

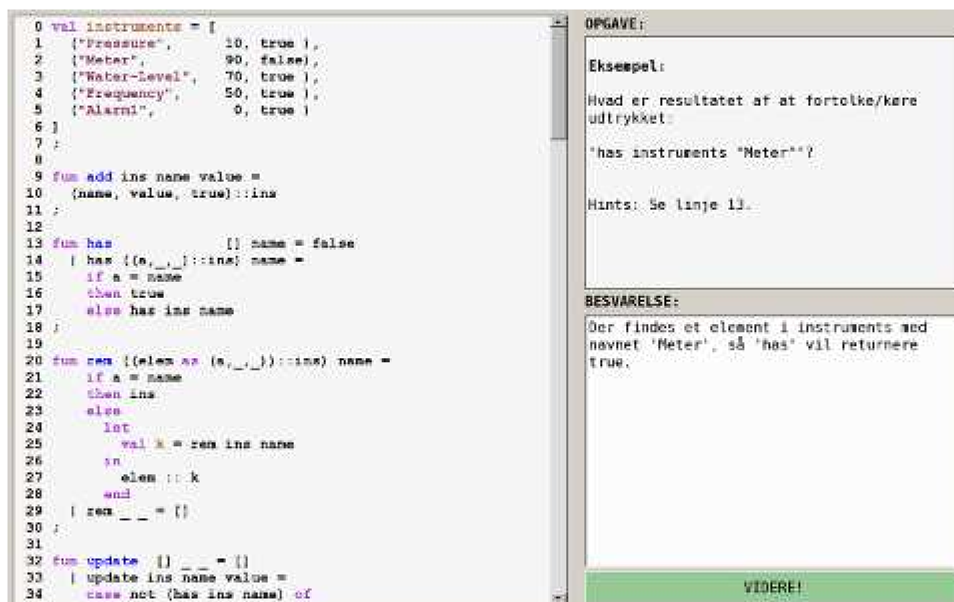

Figure 2: The GUI used to present code along with tasks.

3.2.2 Recovery

In order to ensure no loss of data, the GUI application is designed with a worst-case scenario in mind. In the rare case of a power failure, or a system crash, the GUI is able to recover to its previous state. Two mechanisms are in place to achieve this:

Auto save: The application automatically saves the current state of the experiment, including partly written answers. Writing the new state is an atomic action ensuring a consistent state.

Resume: After an unexpected interruption (e.g. power failure), the application is able to resume the session from the last auto-saved state.

3.2.3 Backend

The GUI application is written in Python using Qt as the graphical framework. Advantages of this includes Qt being cross-platform and the ability to use a drag-and-drop GUI builder. The GUI builder included with Qt generates a UI markup that is easily converted to a Python module.

While Qt provides standard widgets such as text fields and buttons, it doesn't provide a code viewer with built in syntax highlighting. It does however provide a generic HTML viewer. This can be used to present highlighted code, by first generating a HTML version of such highlighting. Here, Emacs is used to generate the HTML version of the highlighting in an automated manner. An example of the highlighting generated can be seen in section C.1 on page 21.

4 Data processing

The following steps provide a quick overview of the data processing performed:

1. The tab separated values (TSV) from the eye-tracker is parsed to an internal representation (see section 3.1 on page 8 for the file format).
2. The data is filtered, leaving only eye-movement within the box containing code.
3. Raw samples are grouped in fixations to reduce common error types.

In the following sections, I introduce two common error types and how they are relaxed. The resulting cleaned data is used to produce visualisations in section 5 on the next page.

4.1 Error types

When recording eye movement, there are two common types of error that affects the results[5]:

Variable error: An unpredictable noise in the measured data. This is a very common problem with measuring physical properties. Instead of finding the true point, some random point nearby is reported.

Systematic error: A consistent and predicable drift in the measured data. For example, if the eye-tracker is always higher than the point looked at. The consistent error may change depending on the angle and can thereby be above the true point when looking up and below the true point when looking down.

Variable error can be relaxed by never trusting a single measurement. Instead, several points are group and their mean is used. This is one purpose of fixation filters as explained in section 4.

Systematic error is relaxed during eye-tracker calibration. However, eye-trackers loose precision over time and the need for re-calibration may emerge. Some studies suggest checking the precision and recalibrating when needed, while others suggest recalibrating at chosen points during the session (see Halverson and Hornof [5] for a discussion).

A problem with recalibrating is the risk of the participant loosing track of thoughts while the session is paused. Specifically, when reading programming code, a break of thought is annoying at least.

For this reason, no recalibration is used in the experiment conducted as part of this project. The measuring relies solely on the initial calibration, which is chosen as the most tedious available.

4.2 Fixation filtering

Since the used eye-tracker output consists of raw samples, it is necessary to apply a fixation filter manually in order to relaxe variable error and remove eye flicker.

This is achieved by grouping the raw samples in fixations with a minimum attention span, using a dispersion-based algorithm (used by [5],[6] and discussed in detail in [10]):

1. Order the samples by recorded time.
2. Pick the first sample as a member of the current fixation.
3. Keep adding samples, as long as the dimensions of the bounding box are acceptable (not too large).
4. When no more samples can be added, check if the time-span of the fixation is acceptable (not too short). If so, accept the mean of the samples as a fixation and skip to the next unused sample. If not, skip the first sample.
5. Repeat from 2.

The chosen minimum duration of a fixation is 100ms (as in [7][10][6], Uwano et al. [14] used 50ms), while the chosen maximum square size is the length of two line heights (46 pixels).

5 Visualisations

In this section, I present 3 forms of visualisations that represents the results of the experiment. Two based on line transitions and one based on token density. Each visualisation presents a way of comparing the results from the sessions.

5.1 Line Transitions

A simple way to examine the programmers behaviour is through line transitions (also referred to as line scans[14]). A line transition is defined as a change of focus from one line to another in focus. That is when two following fixations appear to be on two different lines.

Uwano et al. [14] investigates at line transitions with disregard to the amount of time spend on each line, yielding a compact visual that seems easy to understand and compare. However, including the time density of each line

could give a better behavioural understanding. For example, the participant could spend a lot of time at one critical line, say an `if` condition resulting in a series of fixations on the same line, while using just a small amount of time on the surrounding lines.

In the presented framework, I have including code for extracting and generating visual presentations of line transitions with and without regard for time. In these plots, two blue lines indicate the code lines relevant to the question asked in the task (see figure 3 for an example). This is the area, where fixations is expected to occur.

Figure 3: Line transitions with time (left) and without time (right).

5.2 Token Density

An interesting aspect when comparing two different programming paradigms (functional SML and object-oriented Python) is to examine the token density. That is, the amount of attention given to each token.

The token density can be computed by correlating the measured fixation points with the original source code. Given the x and y coordinates of each fixation the corresponding token in the code is identified. Each token can now be marked according to their number of fixations.

In the framework, I have included code for marking the tokens in the code with boxes. The colour of each box spans from white to red (in 8 steps), indicating the amount of fixations located at the token. Specifically, the colour is chosen as:

$$p := 1 - \frac{\min(c, 8)}{8}$$
$$rgb := (255, 255 \cdot p, 255 \cdot p)$$

Where c is the number of fixations associated with the specific token and rgb is the Red-Green-Blue value of the colour. For an example, see figure 4 on the following page.

This colour-picking method is very simple. As it is, fixation counts are truncated at 8 and the time is disregarded. This yields a few problems:

1. The duration of each fixation is ignored - two short fixations counts for more than one long.

Figure 4: Example of token highlighting

2. A participant who spends longer on a piece of code will likely have more fixations on each relevant token. Hence, the plots are difficult to compare between sessions.

These problems could be relaxed by instead using the relative duration of each fixation to compute the percent of attention given to each token. The colour could then be defined by:

$$\begin{aligned} p :=& 1 - \frac{d}{t} \\ rgb :=& (255, 255 \cdot p, 255 \cdot p) \end{aligned}$$

Where d is the duration the participant used at the token and t is the amount of time the participant looked at any token. Hence, p is the percentage of time spend at other tokens.

A problem with this method, is that the attention given to each token tends to be low when compared against the total amount. This again, could be relaxed by using an exponential function, yielding larger gaps between the values while still allowing for comparison between sessions.

5.3 Comparisons

To enable easy comparison of visualised results between participants, I have made a Javascript based front-end that allows two visualisations to be shown side-by-side.

The Javascript-based front-end is available online at:

SML

Line transitions without time:
`http://130.225.99.208/eye/sml/lines-fix`

Line transitions with time:
`http://130.225.99.208/eye/sml/lines-fix-time`

Token plots:
`http://130.225.99.208/eye/sml/tokens`

Python

Line transitions without time:
`http://130.225.99.208/eye/python/lines-fix`

Line transitions with time:
`http://130.225.99.208/eye/python/lines-fix-time`

Token plots:
`http://130.225.99.208/eye/python/tokens`

As can be seen in figure 5 the interface allows comparison between two participants (student or tutor) on a specific question.

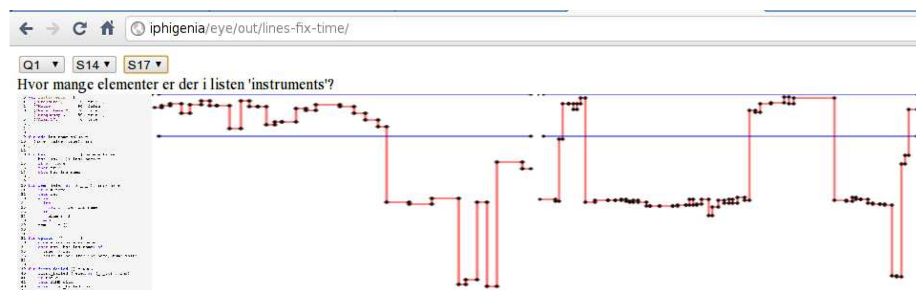

Figure 5: Comparing line transitions with time between student 14 and 17 on question 1.

6 Future Work

In this project, I have developed a graphical user interface for assisting experiments involving program code and a framework for visualising the resulting data and I have carried out a small experiment using these tools. However, the data collected during this project has yet to be analysed.

Furthermore, in order to provide the empirical data necessary to draw any conclusions, more experiments need to be executed.

When more data is at hand, it is necessary to reconsider the fixation filter used in the framework. It is very simple, and it is possible that better methods exists. If the implemented fixation filter is adequate, the code can be used as-is.

More visualisations could be added, but as the time of writing, the current visualisations seem to be the most popular. However, it may be useful to add summary generation as a quick overview of the results.

The feature for comparing different students could also be enhanced for greater usability, while adding a new feature for comparing groups of users (e.g. students vs. tutors) could prove helpful during analysis.

References

[1] D. Binkley, M. Davis, D. Lawrie, and C. Morrell. To camelcase or under_score. In ICPC, pages 158–167. IEEE Computer Society, 2009. URL `http://dx.doi.org/10.1109/ICPC.2009.5090039`.

[2] A. D. Da Cunha and D. Greathead. Does personality matter?: an analysis of code-review ability. Commun. ACM, 50:109–112, May 2007. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1230819.1241672. URL `http://doi.acm.org/10.1145/1230819.1241672`.

[3] A. B. Downey, J. Elkner, and C. Meyers. How to Think Like a Computer Scientist: Learning with Python. Green Tea Press, 2002. ISBN 0971677506. URL `http://www.amazon.com/How-Think-Like-Computer-Scientist/dp/0971677506`.

[4] J. S. Dumas and J. C. Redish. A Practical Guide to Usability Testing. Intellect Ltd, 1999. ISBN 1841500208. URL `http://www.amazon.com/Usability-Testing/dp/1841500208/`.

[5] T. Halverson and A. J. Hornof. Cleaning up systematic error in eye-tracking data by using required fixation locations. Behavior Research Methods, Instruments, & Computers, 34:592–604, 2002.

[6] K. Hornbæk and M. Hertzum. Untangling the usability of fisheye menus. ACM Trans. Comput.-Hum. Interact., 14(2), 2007. URL `http://dblp.uni-trier.de/db/journals/tochi/tochi14.html#HornbaekH07`.

[7] A. J. Hornof and T. Halverson. Cognitive strategies and eye movements for searching hierarchical computer displays. In G. Cockton and P. Korhonen, editors, CHI, pages 249–256. ACM, 2003. ISBN 1-58113-630-7. URL `http://dblp.uni-trier.de/db/conf/chi/chi2003.html#HornofH03`.

[8] M. W. M. Jaspers, T. Steen, C. van den Bos, and M. M. Geenen. The think aloud method: a guide to user interface design. I. J. Medical Informatics, 73(11-12):781–795, 2004. URL `http://dblp.uni-trier.de/db/journals/ijmi/ijmi73.html#JaspersSBG04`.

[9] L. C. Paulson. ML for the Working Programmer. Cambridge University Press, 1996. ISBN 052156543X. URL `http://www.amazon.com/ML-Working-Programmer/dp/052156543X`.

[10] D. D. Salvucci and J. H. Goldberg. Identifying fixations and saccades in eye-tracking protocols. In ETRA '00: Proceedings of the 2000 symposium on Eye tracking research & applications, pages 71–78, New York, NY, USA, 2000. ACM. ISBN 1-58113-280-8.

[11] B. Sharif and J. I. Maletic. An eye tracking study on camel-case and under_score identifier styles. In ICPC, pages 196–205. IEEE Computer Society, 2010. ISBN 978-0-7695-4113-6. URL http://doi.ieeecomputersociety.org/10.1109/ICPC.2010.41.

[12] H. A. Taha. Operations Research: An Introduction (9th Edition). Prentice Hall, 2010. ISBN 013255593X. URL http://www.amazon.com/Taha/dp/013255593X/.

[13] Tobii. Tobii x120 website, Jan. 2011. URL http://bit.ly/tobii-x120.

[14] H. Uwano, M. Nakamura, A. Monden, and K. ichi Matsumoto. Analyzing individual performance of source code review using reviewers' eye movement. In ETRA, pages 133–140, 2006.

A Eye-tracker output

A.1 Header

An example file header as outputtet by the eye-tracker:

```
System Properties:
Operating System:  Microsoft Windows NT 5.1.2600 Service Pack 2
System User Name:  Experience Lab
Machine Name:  TESTPC

Data properties:

Recording name:  Rec 14
Recording date:  10/19/2010
Recording time:  9:07:26 AM
Recording resolution:  1280 x 1024

Export date:  10/19/2010
Export time:  10:54:36 AM

Participant:  P15
Filter settings:

Eye: Average
Validity: Normal
Fixation filter: Raw data
```

B Tasks

1. Question:

Hvad er resultatet af at fortolke/køre udtrykket: 'has instruments "Meter"'?

This first question serves as an example and is joined by the corresponding example answer: *Der findes et element i instruments med navnet 'Meter', så 'has' vil returnere true."* (Danish).

It's purpose is to familiarize the student with the GUI and the procedure of answering a question. It's focused on the main data structure of the code and gives the student a superficial understanding of what she can expect from the remaining tasks.

2. Question:

Hvor mange elementer er der i listen 'instruments'?

Like the example, this question focuses on the main data structure. The answer is easy, if the student understood the example.

3. Question:

Hvad indeholder listen 'instruments' og hvad ser dens elementer ud til at beskrive?

This question is a bit vague. It asks the student to give an interpretation of the contents of the main data structure, but the student has not seen it used yet and has to make a guess.

The main purpose of this question is to ensure that the student has an understanding of the data structure. The rest of the code makes repeated use of it.

4. **Question:**

   Hvad gør '::' i funktionen 'add'?

   The `cons` operator is introduced and used to add a new element to the beginning of a list.

5. **Question:**

   Hvordan bruges variablen 'a' i funktionen 'has'.

   A simple form of pattern matching is introduced, namely the unpacking of a 3-tuple as the first element of a list. The variable `a` occurs in the expression `((a,_,_)::ins)`. Right after, the check `a == name` is performed. An answer stating that `a` is the name of the current element being checked will suffice.

6. **Question:**

   Hvad er resultatet af at fortolke/køre udtrykket 'add [] "Speaker" 0'?

   The student is asked to explain of the cons performed in the function `add`. No new concepts are introduced, so the question is considered easy.

7. **Question:**

   På hvilke linjenumre kan funktionen 'rem' terminere?

   The student now has to show an understanding of function termination and recursion. The `rem` function has 3 cases where it terminates and 1 case where it continues recursively. Though no deep understanding of how the function works is required to answer the question, it is still in the harder half.

8. **Question:**

   Hvad indeholder variablen 'k' i funktionen 'rem'?

   This is a follow up on the previous question. The student is forced to gain a basic understanding of how the `rem` function works, specifically what the recursive call returns.

9. **Question:**

   Hvad tester 'case' i funktionen 'update'?

   The `case` expression is introduced. It is used in place of an `if` expression as `case not (has ins name)`. The student has to explain what is tested, however since the matched expression is a `boolean` value this is easy.

10. **Question:**

    Hvad kendetegner de elementer som funktionen 'too_high' returnerer?

Pattern matching is again used to unpack the first tuple in a list. Furthermore, the if expression is introduced and used to filter out elements with a value ≤ 90.

The question is a bit vague. Any answer that mentions the elements of the resulting list having a value greater than 90 will suffice.

11. **Question:**
    Under hvilke forhold terminerer funktionen 'first_failed'?

    An if expression is used along with recursion to find the first element with a false state. Also, an option type is used to represent the case where no such value element exists.

    To answer the question, the student has to understand why the function terminates. This can be decided simply by looking at the `if` expression.

12. **Question:**
    Hvad returnerer funktionen 'too_high'?

    This is a revisit to the function `too_high`. The question is the same as question number 10, but rephrased slightly. If the student merely answered the type `list` in question 10, then this question will be an opportunity to elaborate.

13. **Question:**
    Hvad tester 'orelse' i funktionen 'contains'?

    The `orelse` expression is introduced. The function is simple and the question easy. However, this is leading up to the harder questions regarding the `diff` function (see next question).

14. **Question:**
    Hvilket formål har 'if not' i funktionen 'diff'?

    This question requires an understanding of the `if` expression and the `contains` function (from question 13). Furthermore, the question asks for an explanation of the purpose of this expression, not merely what it does. This implies an understanding of the `diff` function.

15. **Question:**
    Hvad gør funktionen 'raw_input'?

    The `TextIO` module is introduced and used to input a line from `stdin`. Furthermore, a simple `strip_newline` function is applied. The hints of the function names and their intuitive usage makes this question is easy.

16. **Question:**
    Forklar kortfattet funktionaliteten af 'updater'.

    The final question is everything the others aren't. The student is asked to shortly describe the functionality of the function `updater`. This function is the largest in the presented code, spanning 36 lines and using 4 local functions. It is the one function that ties together and uses all the other code. It is included merely to let the users read a larger function.

C Experiment Source Code

C.1 Highlighting

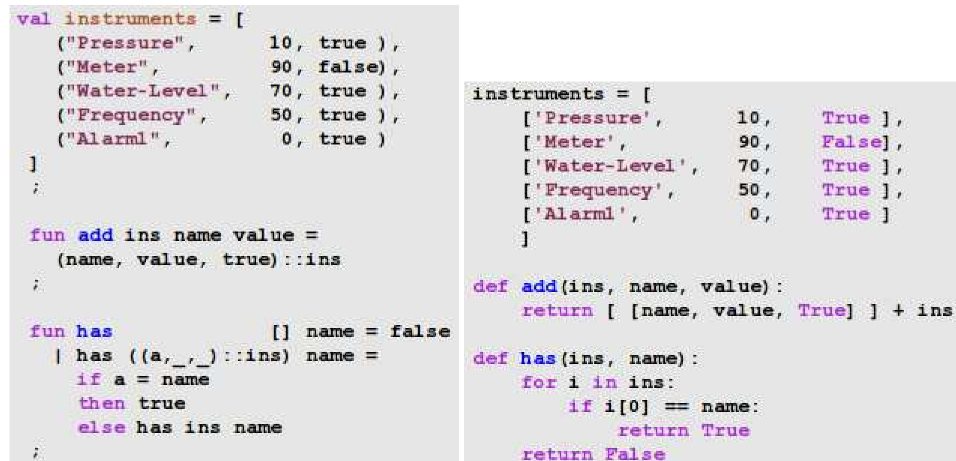

Figure 6: Syntax highlighting of Standard ML (left) and Python (right).

C.2 Standard ML

```
1   val instruments = [
2      ("Pressure", 10, true ),
3      ("Meter", 90, false),
4      ("Water-Level", 70, true ),
5      ("Frequency", 50, true ),
6      ("Alarm1", 0, true )
7    ]
8    ;
9
10   fun add ins name value =
11     (name, value, true)::ins
12   ;
13
14   fun has [] name = false
15     | has ((a,_,_)::ins) name =
16       if a = name
17       then true
18       else has ins name
19   ;
20
21   fun rem ((elem as (a,_,_))::ins) name =
22       if a = name
23       then ins
```

```
24        else
25          let
26            val k = rem ins name
27          in
28            elem :: k
29          end
30    | rem _ _ = []
31  ;
32
33  fun update [] _ _ = []
34    | update ins name value =
35        case not (has ins name) of
36          true => ins
37        | false => add (rem ins name) name value
38  ;
39
40  fun first_failed [] = NONE
41    | first_failed ((elem as (_,_,c))::ins) =
42        if not c
43        then SOME elem
44        else first_failed ins
45  ;
46
47  fun too_high [] = []
48    | too_high ((elem as (_,b,_))::ins) =
49        if b > 90
50        then elem::(too_high ins)
51        else too_high ins
52  ;
53
54  local
55    fun contains (elem1::ins) elem0 =
56          elem0 = elem1 orelse contains ins elem0
57      | contains _ _ = false
58  in
59  fun diff [] ins1 = []
60    | diff (elem0::ins0) ins1 =
61        if not (contains ins1 elem0)
62        then elem0 :: (diff ins0 ins1)
63        else diff ins0 ins1
64  end
65  ;
66
67  fun elem_toString (a,b,c) =
68    "(" ^ a ^ "," ^
69     Int.toString b ^ "," ^
70    Bool.toString c ^ ")"
```

```
71  ;
72
73  fun ins_toString [] = ""
74    | ins_toString (elem::ins) =
75      elem_toString elem ^ "," ^ (ins_toString ins)
76  ;
77
78  fun diff_all ins0 ins1 =
79    "[" ^ ins_toString (diff ins0 ins1) ^
80    "] => " ^
81    "[" ^ ins_toString (diff ins1 ins0) ^
82    "]"
83  ;
84
85  local
86    fun out ((a,b,c)::ins) =
87      let
88        val oper = if c then "Yes" else "No"
89        val s = a ^ ", " ^ Int.toString b ^
90                ", " ^ oper
91      in
92        (s ^ "\n") ^ (out ins)
93      end
94    | out _ = ""
95  in
96  fun print_status ins =
97    let
98      val failed =
99          case first_failed ins of
100           NONE => ""
101         | SOME elem => elem_toString elem
102     val high = ins_toString (too_high ins)
103     val s =
104           "====\n" ^
105           "Name: Value: Operating:\n" ^
106           out ins ^ "\n" ^
107           "High:" ^ high ^ "\n\n" ^
108           "Failed:" ^ failed ^ "\n" ^
109           "====\n"
110     in print s end
111 end
112 ;
113
114 local
115   fun strip_newline s =
116     String.substring (s, 0, String.size s - 1)
117
```

```
118    fun raw_input msg =
119      let
120        val _ = print msg
121        val s = TextIO.inputLine TextIO.stdIn
122      in
123        strip_newline s
124      end
125
126    fun add_new ins name =
127      let
128        val value = raw_input "INSERT␣value="
129      in
130        case value of
131          "" => ( print "No␣value.\n" ; ins )
132        | s => add ins name (
133                valOf (Int.fromString s))
134      end
135
136    fun change ins name =
137      let
138        val value = raw_input "CHANGE␣value="
139      in
140        case value of
141          "" => ( print "No␣value.\n" ;
142                      ins )
143        | "REM" => ( print "REMOVE\n" ;
144                      rem ins name )
145        | s => update ins name (
146                   valOf (Int.fromString s))
147      end
148  in
149  fun updater ins =
150    let
151      val _ = print_status ins
152      val name = raw_input ">␣name="
153    in
154      if name = "quit"
155      then ins
156      else
157        if String.size name = 0
158        then (print "No␣name." ; updater ins)
159        else
160          let
161            val ins_new =
162                if not (has ins name)
163                then add_new ins name
164                else change ins name
```

```
165             in
166               (print (diff_all ins ins_new ^ "\n") ;
167                updater ins_new)
168             end
169         end
170   end
```

C.3 Python

```
1  instruments = [
2      ['Pressure', 10, True ],
3      ['Meter', 90, False],
4      ['Water-Level', 70, True ],
5      ['Frequency', 50, True ],
6      ['Alarm1', 0, True ]
7      ]
8
9  def add(ins, name, value):
10     return [ [name, value, True] ] + ins
11
12 def has(ins, name):
13     for i in ins:
14         if i[0] == name:
15             return True
16     return False
17
18 def rem(ins, name):
19     ins = list(ins)
20     for i in range(len(ins)):
21         if ins[i][0] == name:
22             del ins[i]
23             return ins
24     return ins
25
26 def update(ins, name, value):
27     if not has(ins, name):
28         return ins
29     return add(rem(ins, name), name, value)
30
31 def first_failed(ins):
32     i = 0
33     while i < len(ins):
34         elem = ins[i]
35         if elem[2] == False:
36             return elem
37         i += 1
38     return None
39
```

```
40 def too_high(ins):
41     if len(ins) == 0:
42         return []
43     elem = ins[0]
44     rest = too_high(ins[1:])
45     if elem[1] > 90:
46         return [elem] + rest
47     else:
48         return rest
49
50 def diff(ins0, ins1):
51     elems = []
52     for elem0 in ins0:
53         change = True
54         for elem1 in ins1:
55             if elem0 == elem1:
56                 change = False
57         if change == True:
58             elems.append(elem0)
59     return elems
60
61 def diff_all(ins0, ins1):
62     s = str(diff(ins0, ins1)) + ' => '
63     s += str(diff(ins1, ins0))
64     return s
65
66 def print_status(ins):
67     print '===='
68     print 'Name: Value: Operating:'
69     for elem in ins:
70         oper = ''
71         if elem[2] == True:
72             oper = 'Yes'
73         else:
74             oper = 'No'
75         s = elem[0] + ', ' + str(elem[1]) + ', ' + oper
76         print s
77     print ''
78     print 'High:', too_high(ins)
79     print ''
80     print 'Failed:', first_failed(ins)
81     print '='*64
82
83 def updater(ins):
84     while True:
85         print_status(ins)
86         print ''
```

```
87          name = raw_input('> name=')
88          if name == 'quit':
89              return ins
90          if len(name) == 0:
91              print 'No name.'
92              ins_new = ins
93          elif not has(ins, name):
94              value = raw_input('INSERT value=')
95              if len(value) == 0:
96                  print 'No value.'
97                  ins_new = ins
98              else:
99                  ins_new = add(ins, name, int(value))
100         else:
101             value = raw_input('CHANGE value=')
102             if value == 'REM':
103                 print 'REMOVE'
104                 ins_new = rem(ins, name)
105             elif len(value) == 0:
106                 print 'No value.'
107                 ins_new = ins
108             else:
109                 ins_new = update(ins, name, int(value))
110         print diff_all(ins, ins_new)
111         print ''
112         ins = ins_new
```