Universiteit Utrecht

[Faculty of **Science**
Information and Computing Sciences]

Neon: a Library for Language Usage Analysis

Jurriaan Hage Peter van Keeken
e-mail: jur@cs.uu.nl
homepage: http://www.cs.uu.nl/people/jur/

Department of Information and Computing Sciences, Universiteit Utrecht

September 29, 2008

1. Introduction

The starting point: Helium

- Helium for learning Haskell
  - Implemented in Haskell
- To get some idea where to improve further we need to know how students "use" the language?
  - do they avoid certain parts of the language?
  - which parts of the syntax are often involved in mistakes?
  - how long does it take to solve a type error?
  - when does it help to know Java, or works against them?
  - and so on...
- Each of these questions is a study by itself.
- Today I only talk about the tool, Neon, we developed to help answer these questions.

The first (easy) step: log the compilations §1

- Logging facility added was added from the outset.
- Compiler logs every compile via a socket connection to a Java server.

```
bigbrother | T |
  1.7.0 (Tue Dec 4 11:00:00 CET 2007) |
  -P/usr/local/helium/lib:. --overloading
  --enable-logging -v /tmp/Interpreter.hs |
  bigbrother/2007-12-14@13_58_20_250/Dummy.hs
```

- Helium has been in use since 2002.
  - Over 68,000 "full" compilation contexts (later is fuller)
- Collection is "in vivo", so polluted to some extent,
- but loggings have been cleaned up (by Peter).
- Now to analyze the loggings...

The design criteria of Neon

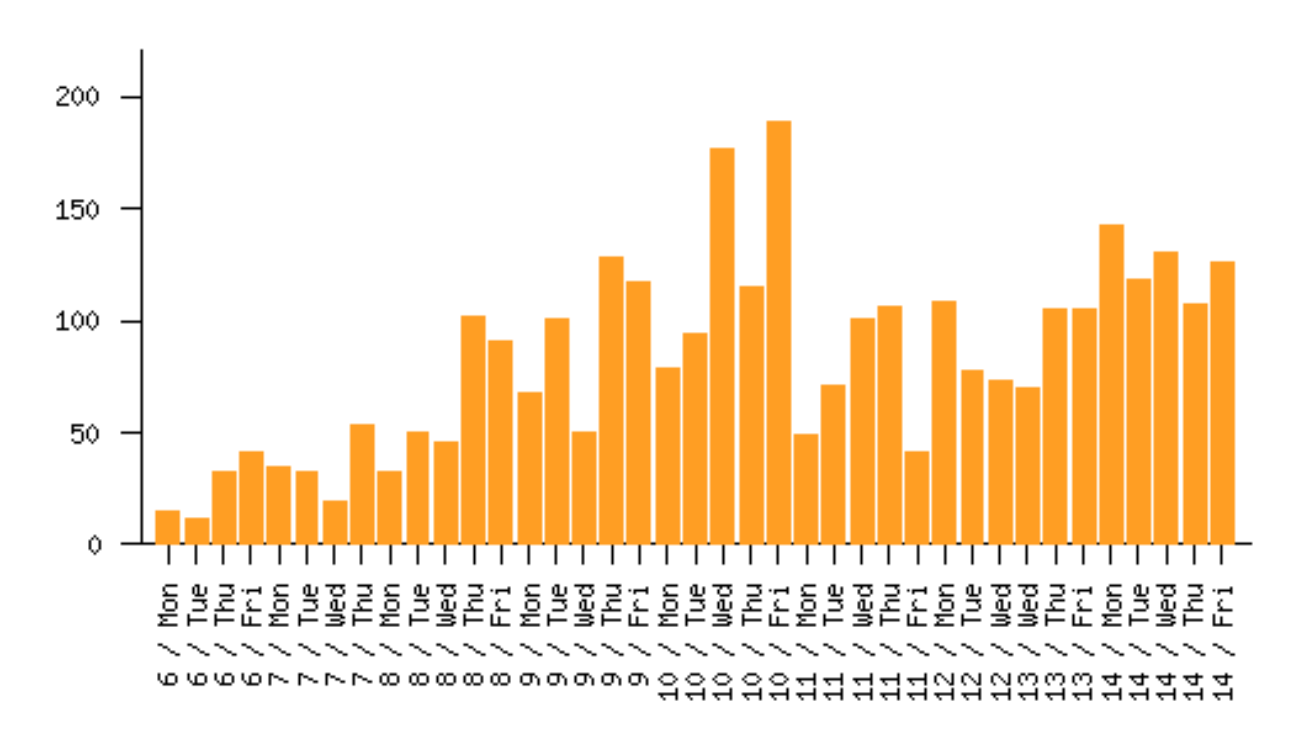
- To do this effectively, we need support. Hence, Neon.
  - Why is effectiveness so important?
- Queries should be concise and conceptually close to what they intend to express.
- Implementation based on a small set of well-understood primitives and combinators,
  - Eases argumentation that implementation of Neon is correct.
- It should be easy to reuse code from the compiler.
  - We can reuse the lexer, parser etc.
- Generate esthetically pleasing output.
  - Support multiple output formats, e.g., HTML tables and PNG files.

The elements

- Slick examples, each with its own reservations.
  - For which we need more queries...
- Descriptive statistics
- The Neon library that
  - implements these ideas
  - ... and allows us to generate these slick examples,
  - ... by writing a bit of simple, reusable code.
- Concluding remarks, future work, points of discussion.

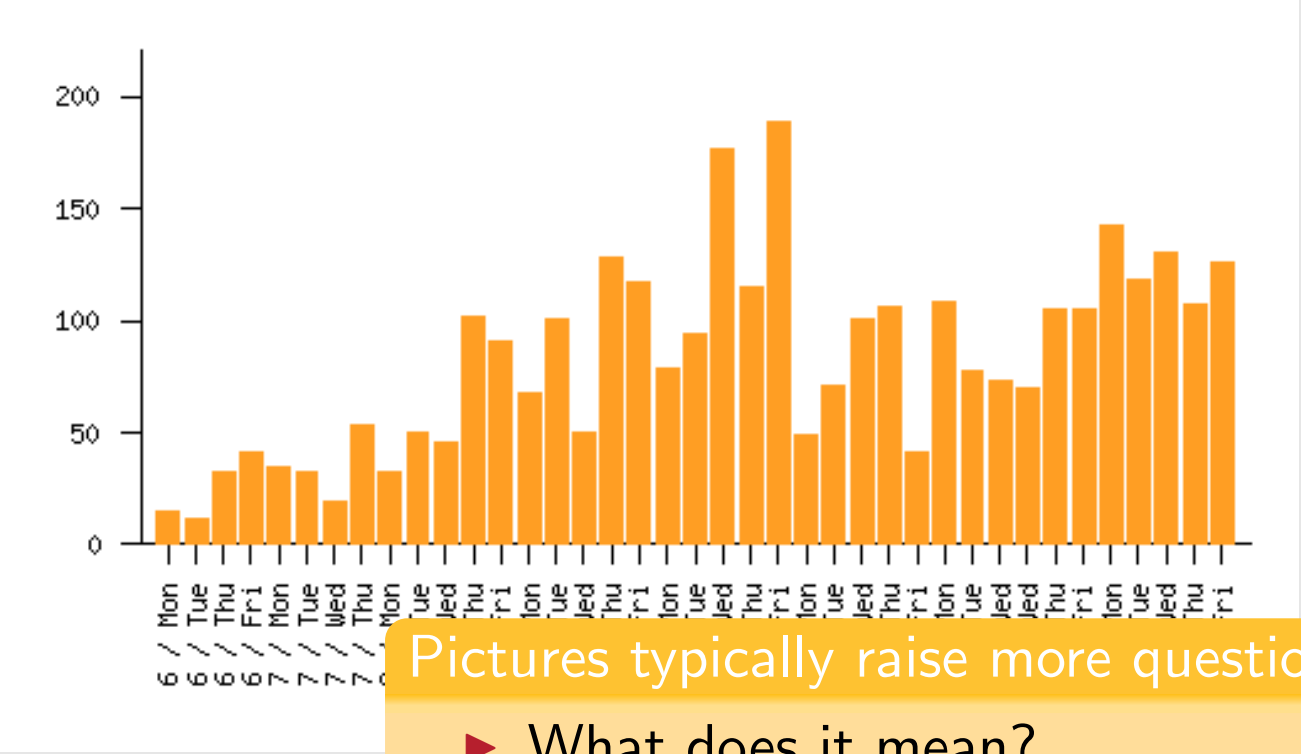
2. The slick examples

Example 1: Module Size



- Average number of lines for compiled modules, given per day (year 2003-2004)

Example 1: Module Size

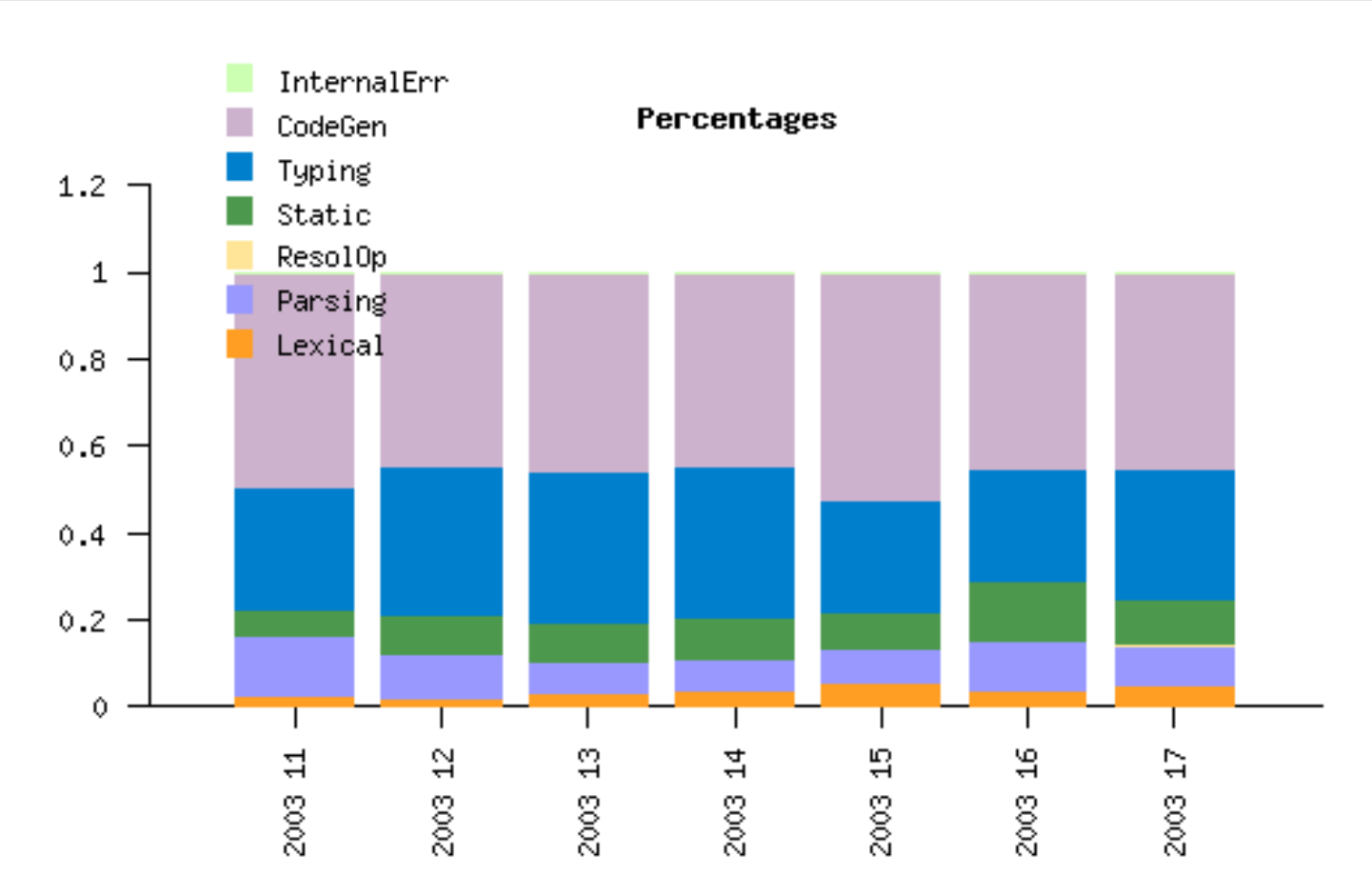


- Average nu
  day (year 2

Pictures typically raise more questions

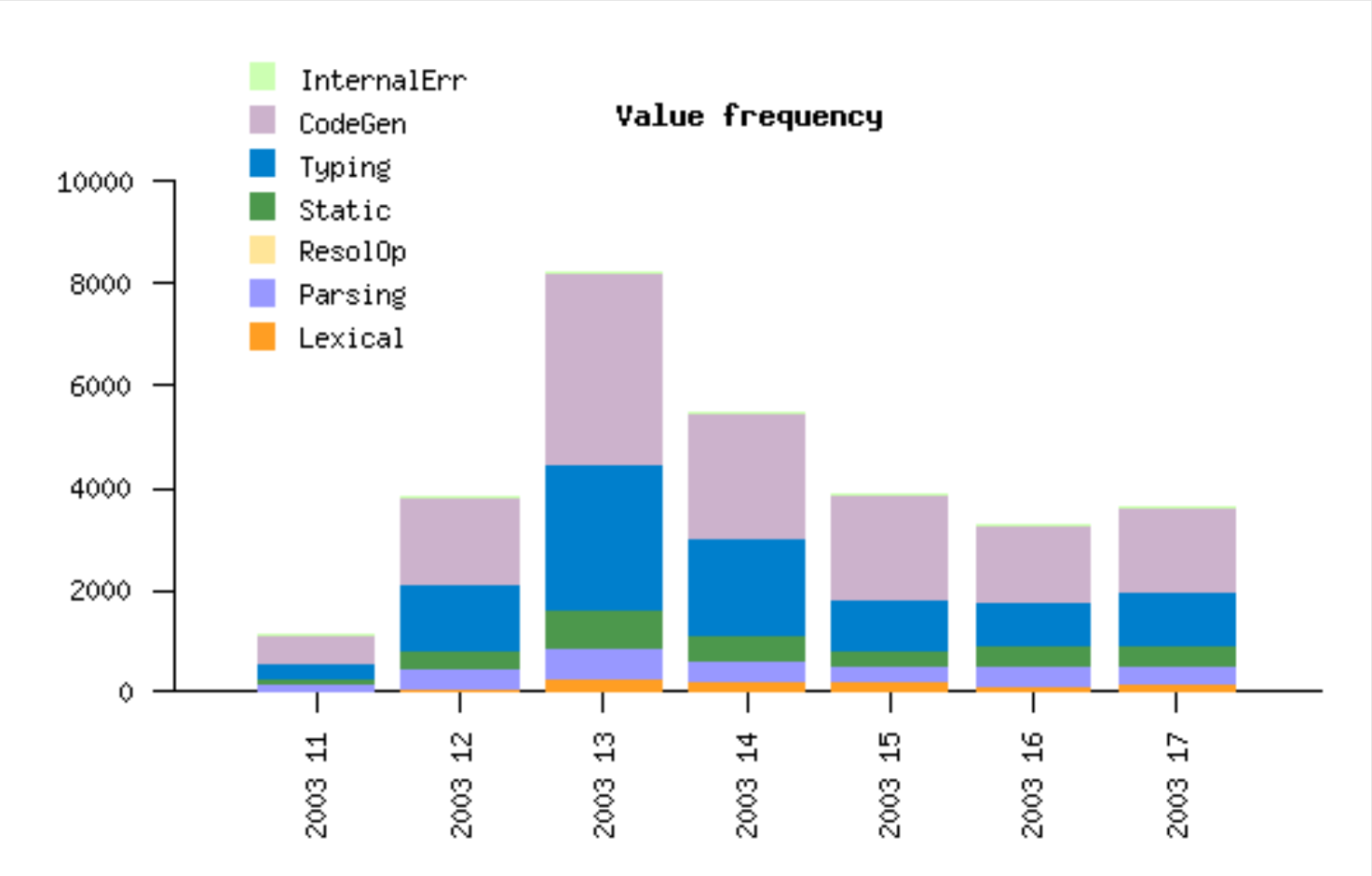
- What does it mean?
- Do all students show the same pattern?
- Correlation with grades, experience, ...?
- Counting code, comments and blank lines separately.

Example 2: Phase Analysis (relative)



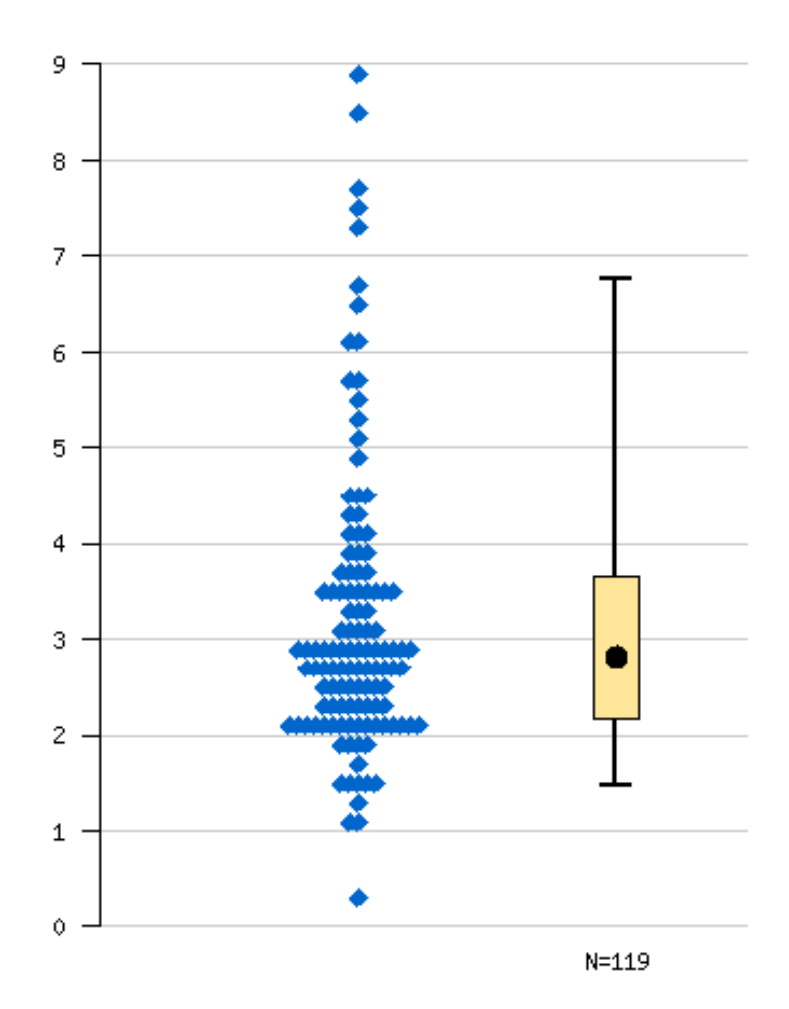
- Why does the ratio of parse errors increase again?
- Do recidivists muddy the picture?

Example 2: Phase Analysis (absolute)



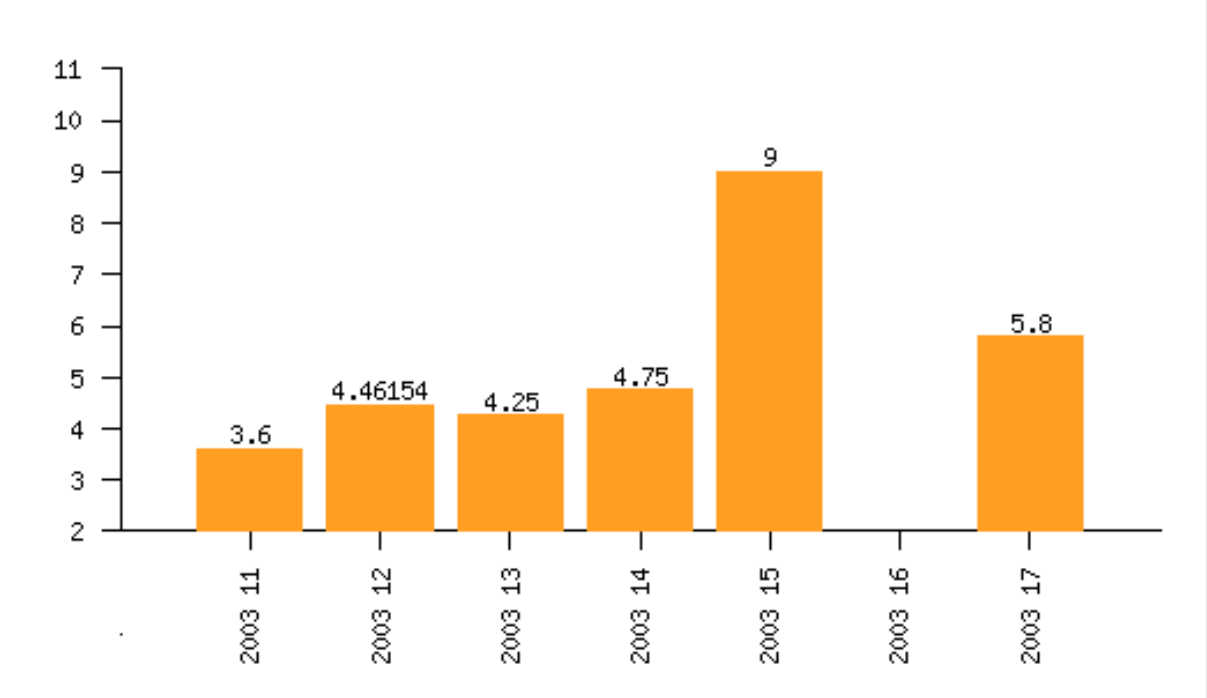
- Absolute gives an idea of weight: how significant are the ratios.
- We want these queries to be similar.

Example 3



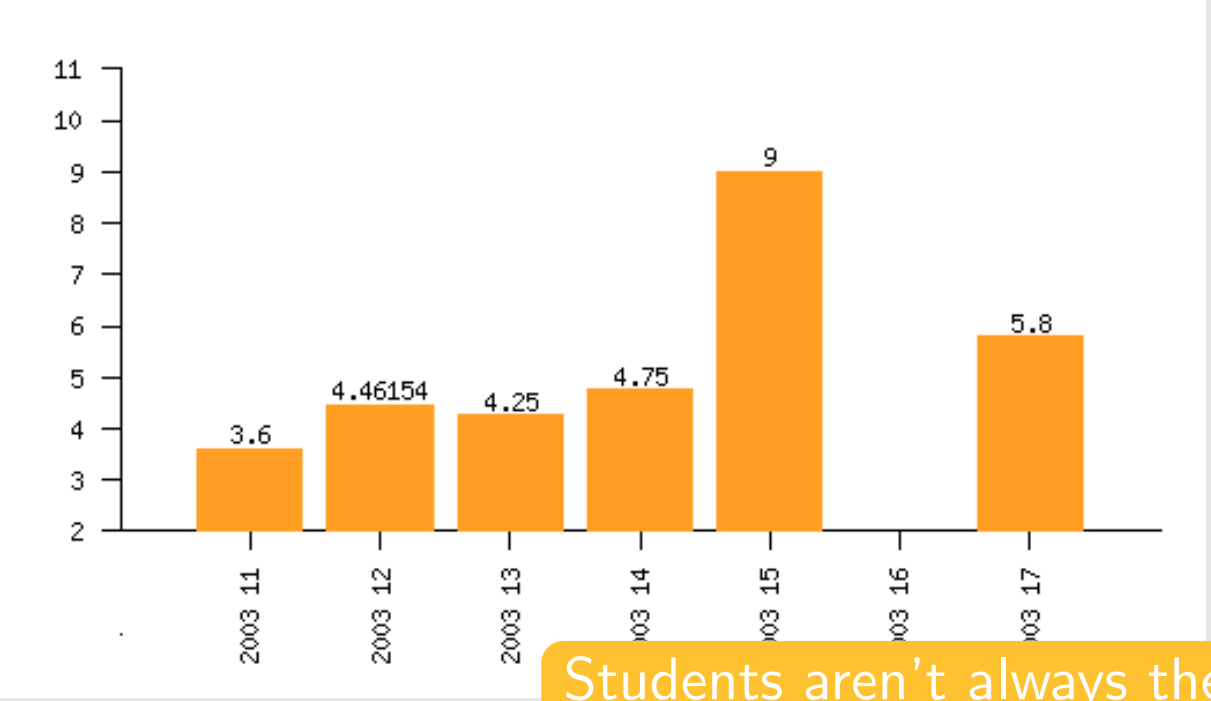
- Average in-between compile time in minutes per student

Example 4



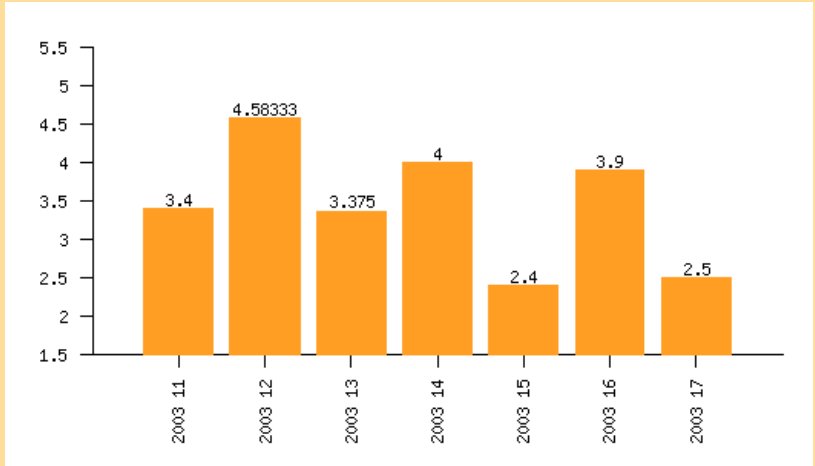
- Average number of compiles needed to "solve" a type error, for a particular student.
- How does one measure this at all?

Example 4



- Average number
  error, for a partic
- How does one m

Students aren't always the same



Universiteit Utrecht

3. The basic concepts

Descriptive statistics

- Easy presentation of results in multiple formats
  - `ploticus` pictures, HTML tables, LaTeX
- Grouping loggings, repeatedly
  - For each student, for each week, compute the list of loggings.
- Filtering on (groups of) loggings.
  - Only lists with at least 10 compiles for a given student
  - Only loggings from the 19th of September
- Computing statistics for groups and other metrics.
  - Only the lengths of the logged programs.
  - The average length of the lists of loggings.

Why not use SQL?

- Why Haskell?
  - General purpose language with strong typing.
  - Built-in support for easy composition and abstraction.
    - Particularly higher-orderness, type classes and polymorphism.
  - Reuse Helium code base
  - Library uses combinators.
    - Facilitates building analyses from others.
- Haskell drawbacks
  - Generating pictures can only be done via existing tools.
    - We use ploticus.
  - Speed could become an issue.
    - Haskell is actually doing well these days.
  - Limited audience.

4. The combinator library

The basics

- An analysis result is represented by $[(key, value)]$.
- The key datatype allows us to describe what the value represents.
- by remembering how $values$ have been computed.
- key inhabits the $DescrKey$ class so that the description can be updated automatically.
- From it, we can generate legend information, filenames and so on.

Some primitives

A few of the (not so primitive) primitives

$$\textbf{type}\ An\ k\ a\ b = [(k, a)] \to [(k, b)]$$

$$basicAnalysis :: (DescrKey\ k) \Rightarrow \\ \quad String \to (a \to b) \to An\ k\ a\ b$$

$$groupAnalysis :: (DescrKey\ k, Enum\ a, DataInfo\ b) \Rightarrow \\ \quad (a \to b) \to ([a] \to [[a]]) \to An\ k\ [a]\ [a]$$

$$mapAnalysis :: (DescrKey\ k) \Rightarrow An\ k\ a\ b \to An\ k\ [a]\ [b]$$

$$\diamond :: An\ k\ b\ c \to An\ k\ a\ b \to An\ k\ a\ c$$

$$runAnalysis :: \ a \to k \to An\ k\ a\ b \to [(k, b)]$$

Group analysis in more detail

$$groupAnalysis :: (DescrKey\ k, Enum\ a, DataInfo\ b) \Rightarrow (a \to b) \to ([a] \to [[a]]) \to An\ k\ [a]\ [a]$$

- First function argument describes which values belong to the same group.
- Second function computes the actual grouping.
- But why is the result type not $[[a]]$?

Group analysis in more detail

$$groupAnalysis :: (DescrKey\ k, Enum\ a, DataInfo\ b) \Rightarrow (a \to b) \to ([a] \to [[a]]) \to An\ k\ [a]\ [a]$$

- First function argument describes which values belong to the same group.
- Second function computes the actual grouping.
- But why is the result type not $[[a]]$?
- Flatten: $[[1,2,3],[2,4]]$ grouped on parity is not $[[[1,3],[2]],[[2,4]]]$, but $[[1,3],[2],[2,4]]$.
- But how do you know that $[1,3]$ and $[2]$ belonged to the same list?

Group analysis in more detail

$$groupAnalysis :: (DescrKey\ k, Enum\ a, DataInfo\ b) \Rightarrow (a \to b) \to ([a] \to [[a]]) \to An\ k\ [a]\ [a]$$

- First function argument describes which values belong to the same group.
- Second function computes the actual grouping.
- But why is the result type not $[[a]]$?
- Flatten: $[[1,2,3],[2,4]]$ grouped on parity is not $[[[1,3],[2]],[[2,4]]]$, but $[[1,3],[2],[2,4]]$.
- But how do you know that $[1,3]$ and $[2]$ belonged to the same list?
- We store that in the key: $[(k1,[1,2,3]),(k2,[2,4])]$ maps to $[(k1',[1,3]),(k1',[2]),(k2',[2,4])]$
- Avoids arbitrarily nested values.

Example: number of loggings per phase

$$groupPerPhase :: DescrKey\ key \Rightarrow An\ key\ [Logging]\ [Logging]$$
$$groupPerPhase =$$
$$\quad groupAnalysis\ phase\ (groupAllUnder\ phase)$$

$$countNumberOfLoggings :: DescrKey\ key \Rightarrow An\ key\ [a]\ Int$$
$$countNumberOfLoggings =$$
$$\quad basicAnalysis''\ \texttt{"number of loggings"}\ length$$

$$loggingsPerPhase :: An\ KeyHistory\ [Logging]\ Int$$
$$loggingsPerPhase = countNumberOfLoggings \diamond groupPerPhase$$

And now to present this


```
presentLoggingPerPhase :: FilePath → FilePath → IO ()
presentLoggingPerPhase logfile outputfp = do
  loggings ← parseLogfile logfile
  let analysisResult = runAnalysis loggings loggingsPerPhase
  barChart ← renderBarChart outputfp analysisResult
  writeFile (outputfp ++ "/analysis.tex")
    (renderLateX $ showAsTable1D analysisResult) +++
    plotToFigure barChart
```

The same, but now per week

Given the definition of $groupPerWeek$

$$
\begin{aligned}
&loggingsPerPhasePerWeek :: An\ KeyHistory\ [Logging]\ Int \\
&loggingsPerPhasePerWeek = \\
&\quad countNumberOfLoggings \\
&\quad \diamond\ groupPerPhase \\
&\quad \diamond\ groupPerWeek
\end{aligned}
$$

$$
\begin{aligned}
&phaseResearch :: FilePath \rightarrow [(KeyHistory, [Logging])] \rightarrow IO\ () \\
&phaseResearch\ outputpath\ input = \mathbf{do} \\
&\quad barChartStacked \leftarrow renderBarChartDynamic\ outputpath \\
&\qquad\qquad\qquad\qquad (loggingsPerPhasePerWeek <\$> input) \\
&\quad writeFile\ ...
\end{aligned}
$$

The same, but now per week

Given the definitior

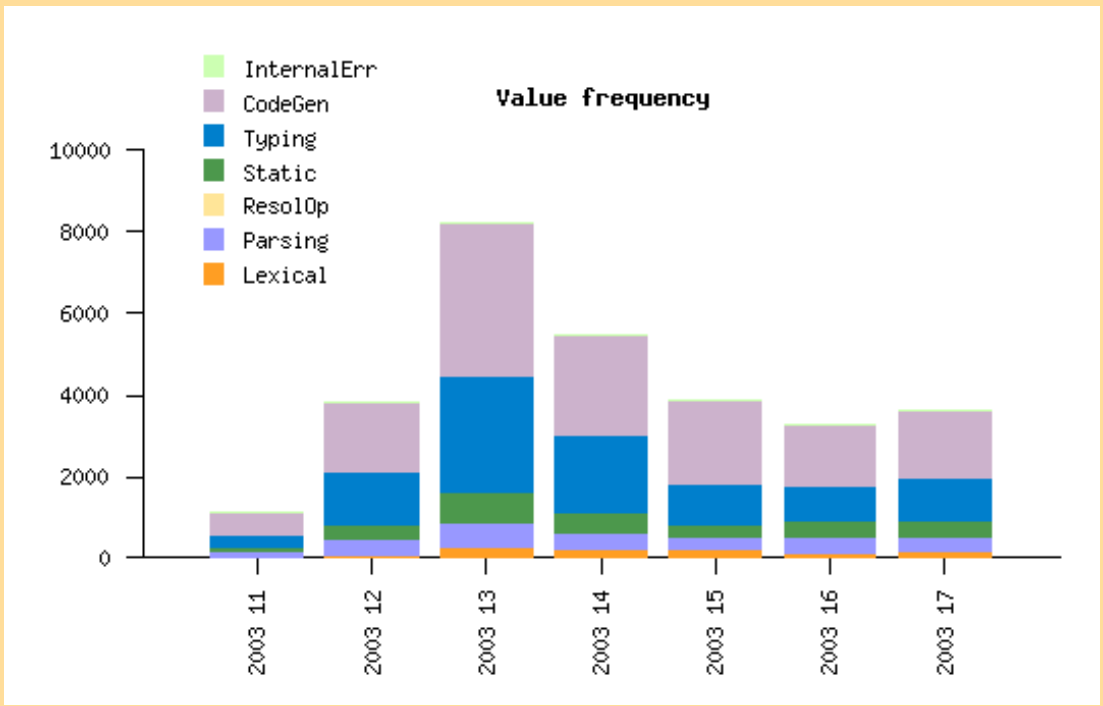
$loggingsPerPhaseF$
$loggingsPerPhaseF$
$countNumberOf$
$\diamond\ groupPerPhas$
$\diamond\ groupPerWeel$

$phaseResearch :: F$
$phaseResearch\ out$
$barChartStackea$

$writeFile\ ...$

$(loggingsPerPhasePerWeek < \$ > input)$

Slight reprise



5. To conclude

What more do we want from Neon?

- In-depth studies
  - Students do not seem particularly interested in doing this kind of study
- The use of student properties
  - Who is (s)he? What grade was obtained? First language or not?
- Easy intergration with different versions of Helium.
  - Not as easy as it may seem.

What more do we want, period?

- Money (to hire a PhD student).
- More loggings
  - In the process of extending Helium to include type classes in full.
- I am looking for expertise in empirical research.

Points for discussion

- Should I include $groupByCoherence$ in the paper?
  - a general function on top of $groupAnalysis$
  - specifies which loggings belong "together".
  - Ex.1: in the same programming session (not too far apart in time)
  - Ex.2: the same file (by name, by content)
  - Ex.3: the conjunction of these two
  - Sometimes you need coherence with lookahead.
- Why not do this for Java?
  - Bigger market
  - BlueJ
- How do we attain empirical validity, especially external validity?
  - Student properties can help.

Coherence

- Typically, we analyze sequences of similar loggings
- Two subsequent loggings can be similar if
  - files with the same name are compiled
  - the compiled files differ in at most one line
  - the time stamp between the loggings is at most 30 minutes apart (time coherence)
- Sequences of loggings grouped into a sequence of sequences by taking the reflexive, transitive closure of similarity.

An example

- Example:
  - How long, on average, to solve a type error within a session?
- Compiles not too far apart and subsequent compiles in a sequence should concern the "same" program.
- What is "same" here?
  - same name
  - the diff of the two programs is small
  - other or all of the above
- Haskell allows easy parameterization over predicates.
  - Higher-orderness and polymorphism really help.

Lookahead

- Scenario:
  - program P has a type error
  - problem reminds student of another module Q
  - student loads and compiles Q
  - session terminates
  - student solves problem in P