Freie Universität Berlin

Softwareprojekt Übersetzerbau

SoSe 2014

Abschlusspräsentation

Haskellgruppe

Rail-LLVM-Compiler

Inhalte

Rail-Spezifikation

- **Rail-Spezifikationen des Esolang-Wikis** [1]

- **Erweiterungen der Spezifikation**
  - negative ganze Zahlen
  - positive und negative Gleitkommazahlen
  - arithmetische Operationen: *add, sub, mul, div, rem*
  - Vergleichsoperationen: *equal, greater*
  - anonyme Funktionen *…war nicht spezifiziert!*

- **Widersprüche**
  - anonyme Funkionen können auf den Stack gepusht oder vom Stack gepoppt werden
  - isList(x) ↔ ¬ isString(x)

[1] Weblink: http://www.esolangs.org/wiki/Rail

Inhalte

Funktionsumfang: Compiler

- **Movements**
- **Strings**
- **Integer**
- **Boolean**
- **Variablen**
- **Funktionsaufrufe & anonyme Funktionen**
- **Listen**
- **Kommando-Befehle**
  - i, o, e — *input, output, end of file*
  - b, u, ? — *boom, underflow, type checking*
  - p, c, s — *append, cut, size*

… und alles mit Fehlerbehandlung !!!

- invalide Programme werden als solche erkannt

Funktionsumfang: Editor

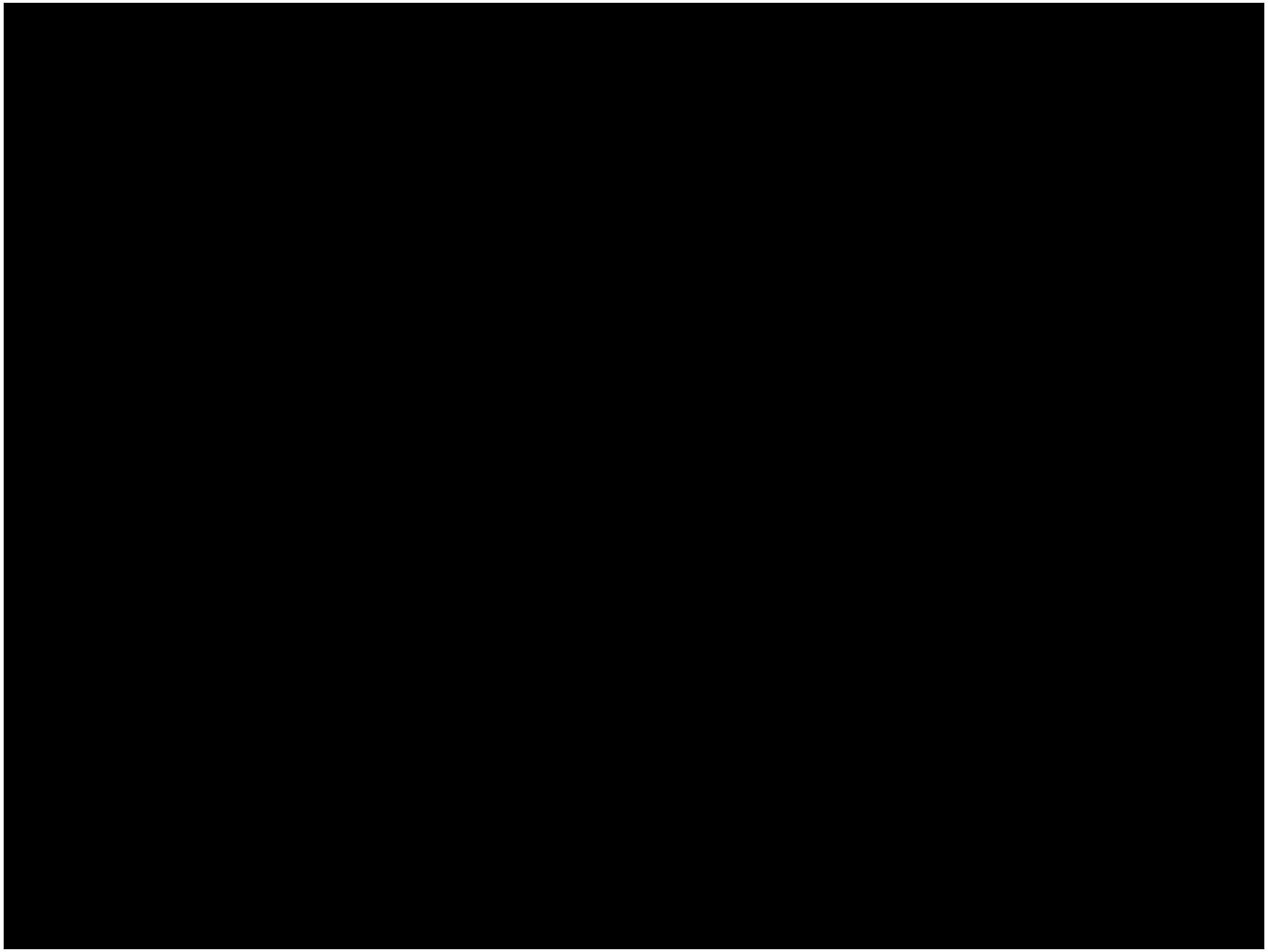
- **Dateien öffnen / speichern**
- **3 Modi**
  - insert-mode — *normaler Eingabemodus*
  - replace-mode — *überschreibt Zeichen*
  - smart-mode — *folgt der Schiene*
- **Programm ausführen**
  - input / output über Editor
- **debug-mode**
  - step by step
  - breakpoints
  - Variablen- und Funktionsstack-Ausgabe
- **dynamische Erweiterung der TextArea**

… und alles in Haskell !!!

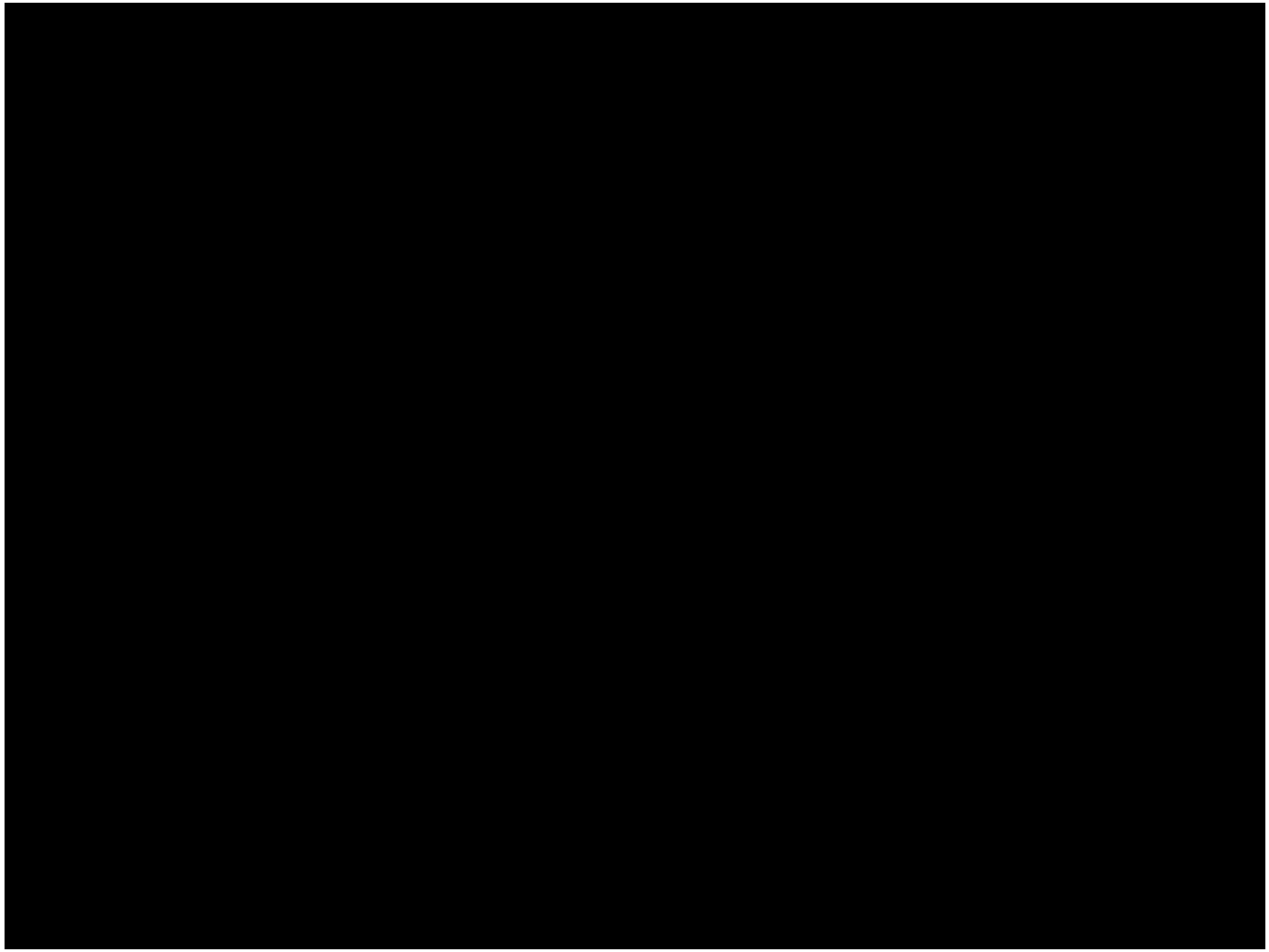
- „haskell la vista, baby“

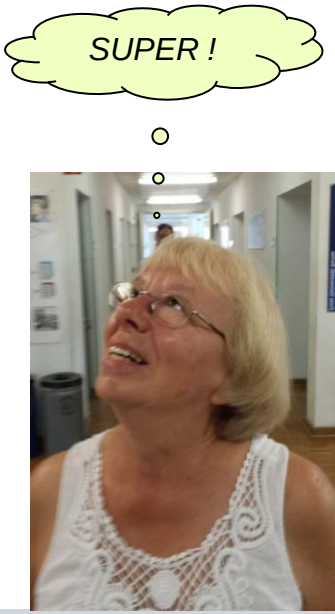
Inhalte

Screencast: Compiler



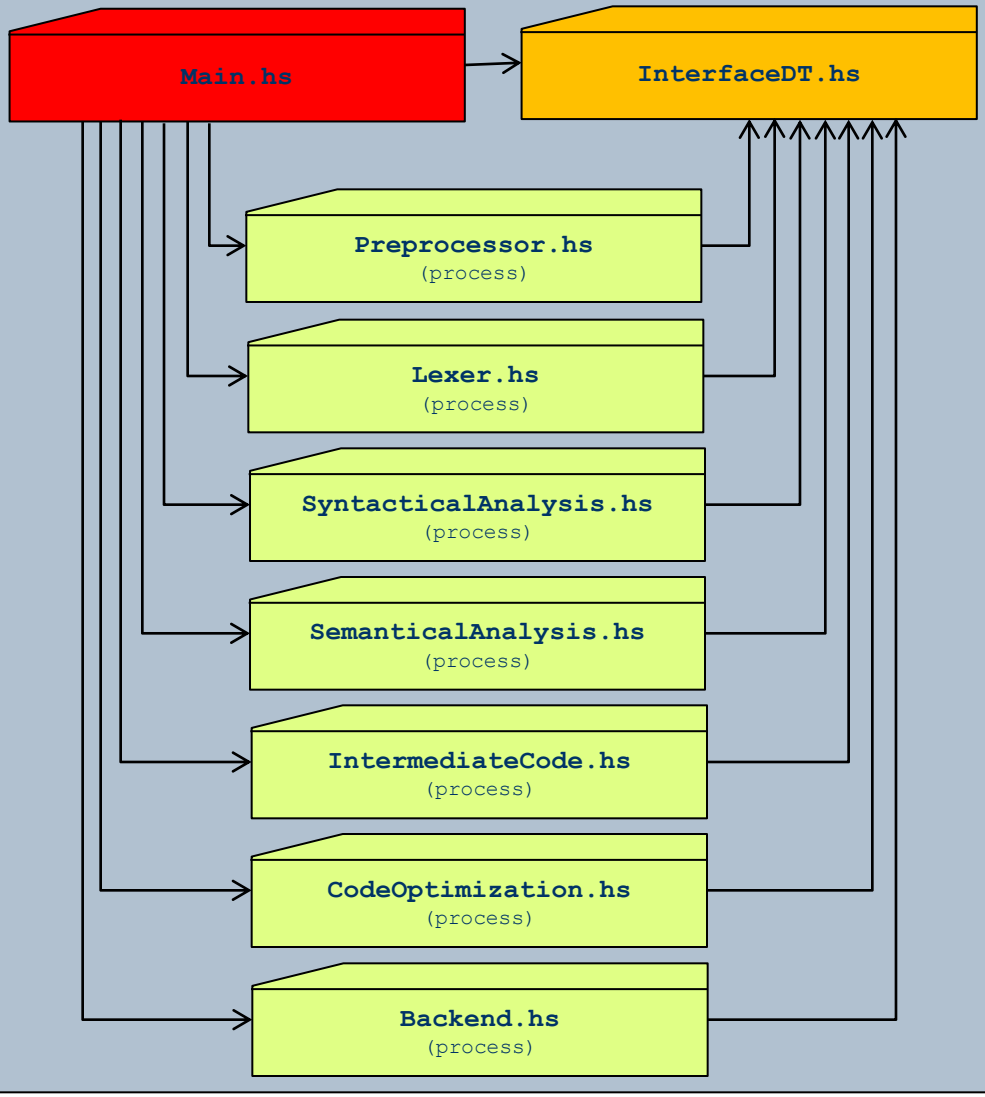
Screencast: Editor





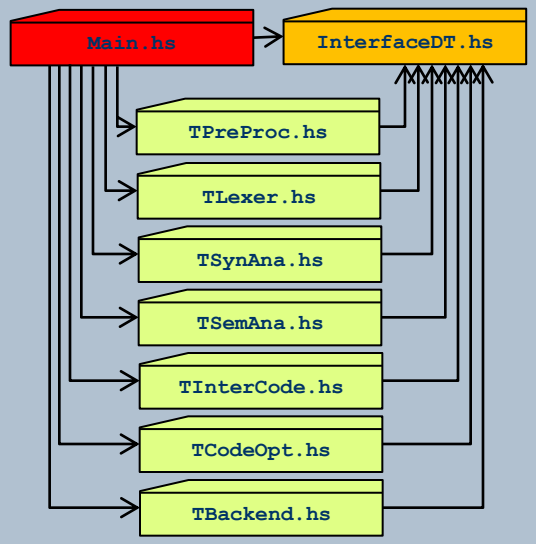
Inhalte

Programm

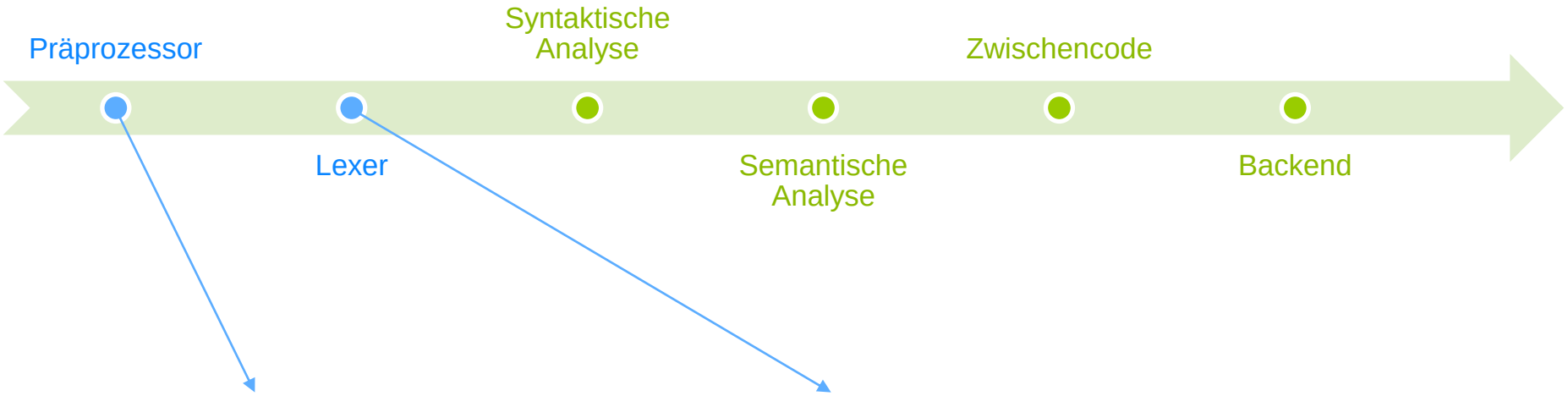


- ✓ Main startet den Rail-LLVM-Compiler
- ✓ InterfaceDT definiert algebraische Datentypen für die Datenübergabe zwischen den Submodulen der Compiler-Pipeline
- ✓ Preprocessor, Lexer, SyntacticalAnalysis, SemanticalAnalysis, IntermediateCode, CodeOptimization und Backend lösen jeweils ihr Teilproblem der Compiler-Pipeline

Tests



Compiler-Pipeline



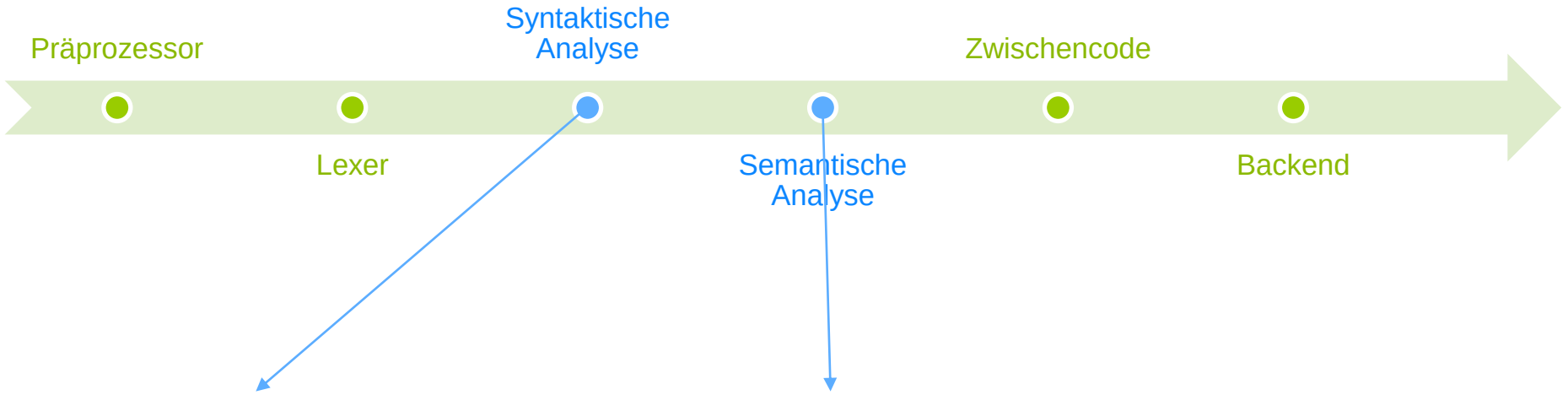
Preprozessor

- splittet nach Funktionen auf
- entfernt alle Zeilen am Anfang der Rail-Datei, bis zur ersten Zeile, die mit einem „$“-Symbol beginnt

Lexer

- fährt Gleise ab
- erzeugt Graphen *(Knoten ≈ Token)*

Compiler-Pipeline



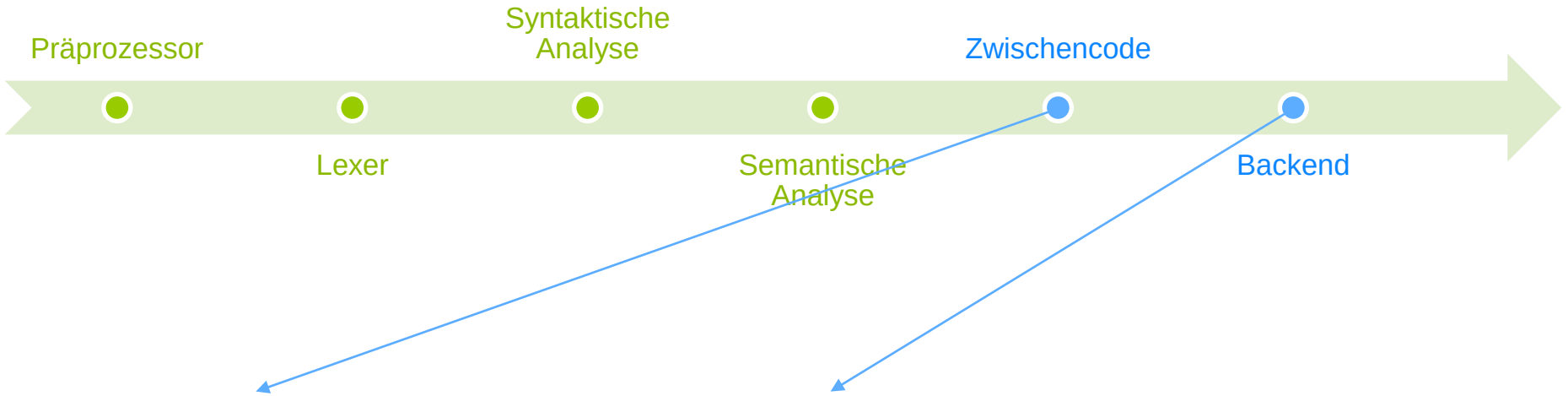
Syntaktische Analyse

- zerlegt Graphen in lineare Pfade
- „verbindet“ lineare Pfade mit „Sprüngen“

Semantische Analyse

- TODO ! → Christian Hofmann
- Welche Fehler werden abgefangen?

Compiler-Pipeline



Zwischencode

- wandelt lineare Pfade in LLVM-Zwischencode um
- verwendet „llvm-general package“

Backend

- implementiert Stack
- implementiert Rail-Operationen
- erzeugt LLVM-Code in Textform

Compiler: Wahl des Targets

- **Zielsprache ist LLVM**

- **Warum?**
  - SSA
  - unverhältnismäßig aufwendig zu programmieren !
  - Architektur unabhängig
  - *„state of art“*
    - erlaubt einfache Optimierung durch ausgelieferte Tools von LLVM
    - machen wir aber nicht

Compiler: Stack-Implementierung

- **LLVM hat keinen Stack-Datentypen**
- **Rail ist Stack-basiert**
- **linked stack**
  - dynamisch wachsend / schrumpfend
- **reference counting**

Compiler / Backend: Wie funktioniert‘s?

- **Eingabe**
  - Haskell-Code:

```
-- (FunctionID, [(PathID (start: 1), List of lexemes to be executed sequentially, following PathID)])
type AST = (String, [(Int, [Lexeme], Int)])
```

- **Ausgabe**
  - LLVM Code (Haskell-Bindings)

- **Vorgehensweise**
  - jeder Pfad wird zu einem Basic-Block übersetzt
  - Erzeugung eines Constant-Pools
  - Übersetzung einzelner Instruktionen
  - lokale Symboltabelle für Funktionen

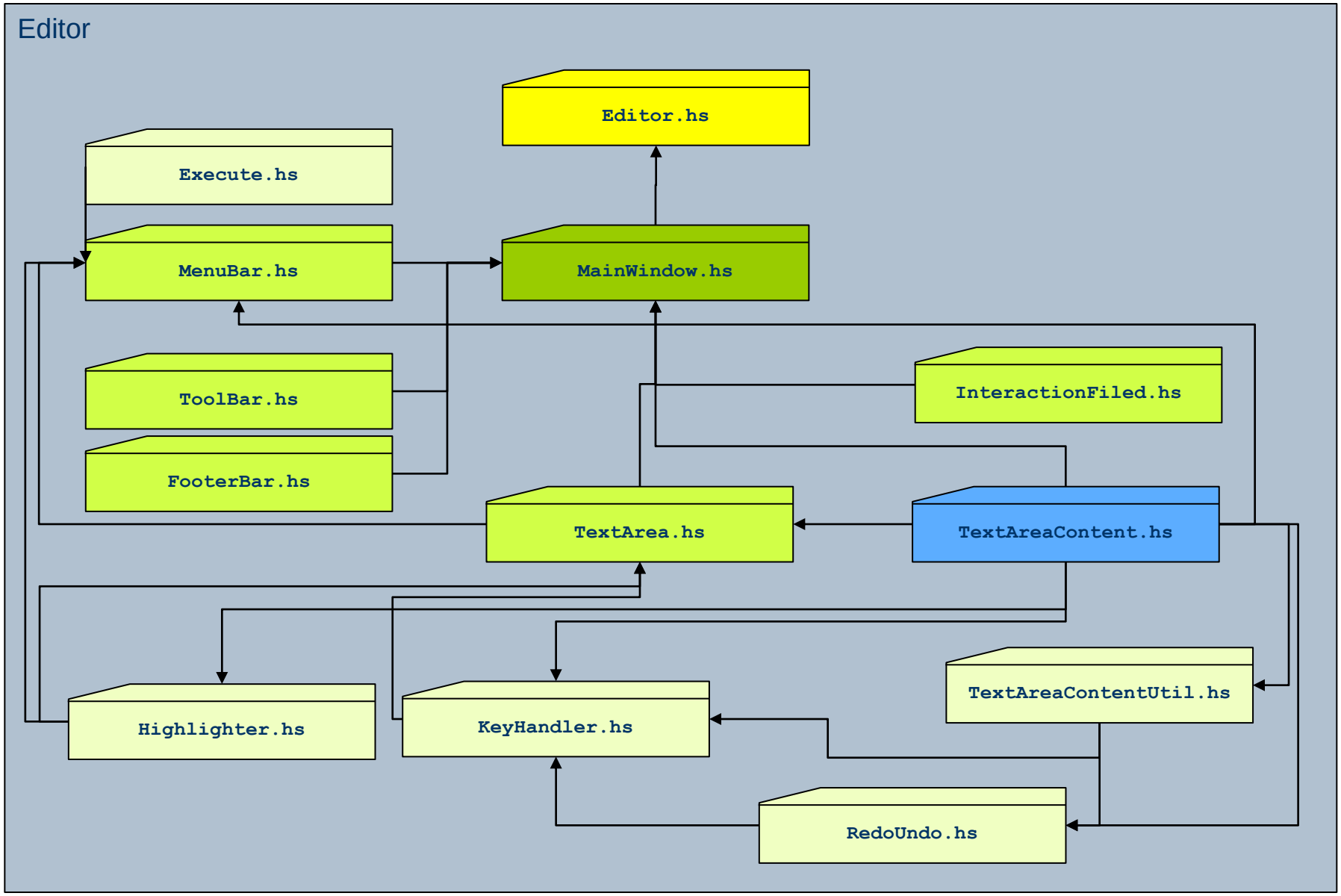
Compiler / Backend: Beispiel

```
generateInstruction Output =
          return [do LLVM.General.AST.Call {
              isTailCall = False,
              callingConvention = C,
              returnAttributes = [],
              function = Right $ ConstantOperand $ GlobalReference $ Name "print",
              arguments = [],
              functionAttributes = [],
              metadata = []
          }]
```

```
; Push a string onto the stack, creating a new stack_element struct
; with a reference count of 1.
;
; The string must already be allocated _ON THE HEAP_.
define %stack_element* @push_string_ptr(i8* %str) {
; 1. Create and push a new stack_element.
;    NB: Stack size is incremented by push_struct().
%elem = call %stack_element* @stack_element_new(i8 0, i8* %str)
call void @push_struct(%stack_element* %elem)

; 2. That's it!
ret %stack_element* %elem
}
```

Inhalte


Editor
Editor.hs
Execute.hs
MenuBar.hs
MainWindow.hs
ToolBar.hs
InteractionFiled.hs
FooterBar.hs
TextArea.hs
TextAreaContent.hs
TextAreaContentUtil.hs
Highlighter.hs
KeyHandler.hs
RedoUndo.hs

Editor: TextArea

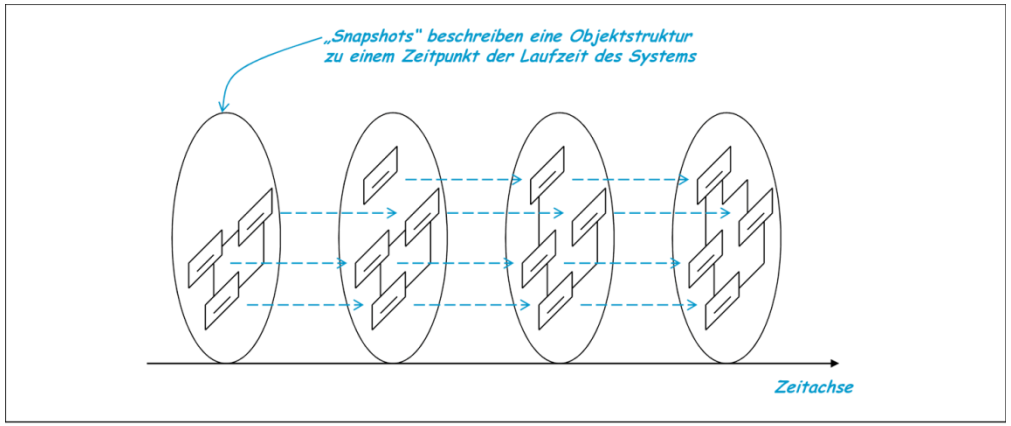
- **Problem: Standard-Textview**
  - wenige Konfigurationsmöglichkeiten
- **Eigenimplementierung eines Textviews**
  - 1. Version: Tabelle von Entries (einzeilige Eingabefelder)
  - 2. Version: Felder in einer Drawing-Area

Teuer!

- **Eigenimplementierung aller Verhaltensweisen eines Standard-Editors**
- **Trennung von Darstellung und Inhalt**
  - Darstellung: TextArea
  - Inhalt: TextAreaContent

Editor: TextAreaContent

- **zusammengesetzte Datenstruktur**
  - Map für Zeichen
  - Map für Farben
  - ActionQueues für RedoUndo
  - Interpreterkontext
- **stellt zur Verfügung**
  - Getter & Setter für Zelleninhalte
  - Getter & Setter für Farben
- **TextAreaContent speichert**
  - Zustand des Editors
  - Zustand des Debuggers
- **zentrale Datenstruktur**



Editor: Highlighting + Debugging

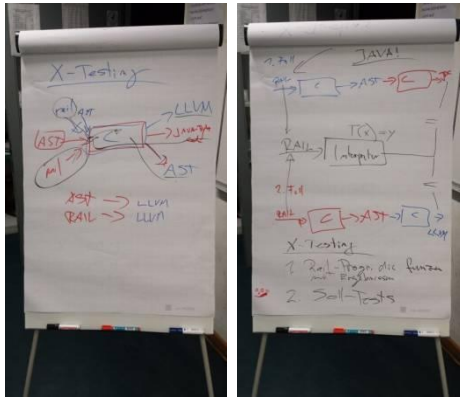
- **Highlighting**
  - verwendet Lexer, um die zu färbenden Lexeme zu ermitteln (aufwändig!)
  - nutzt TextAreaContent, um die Farbe eines Zeichens zu speichern
  - Also: Jetzt gibt‘s alles in Farbe und bunt!

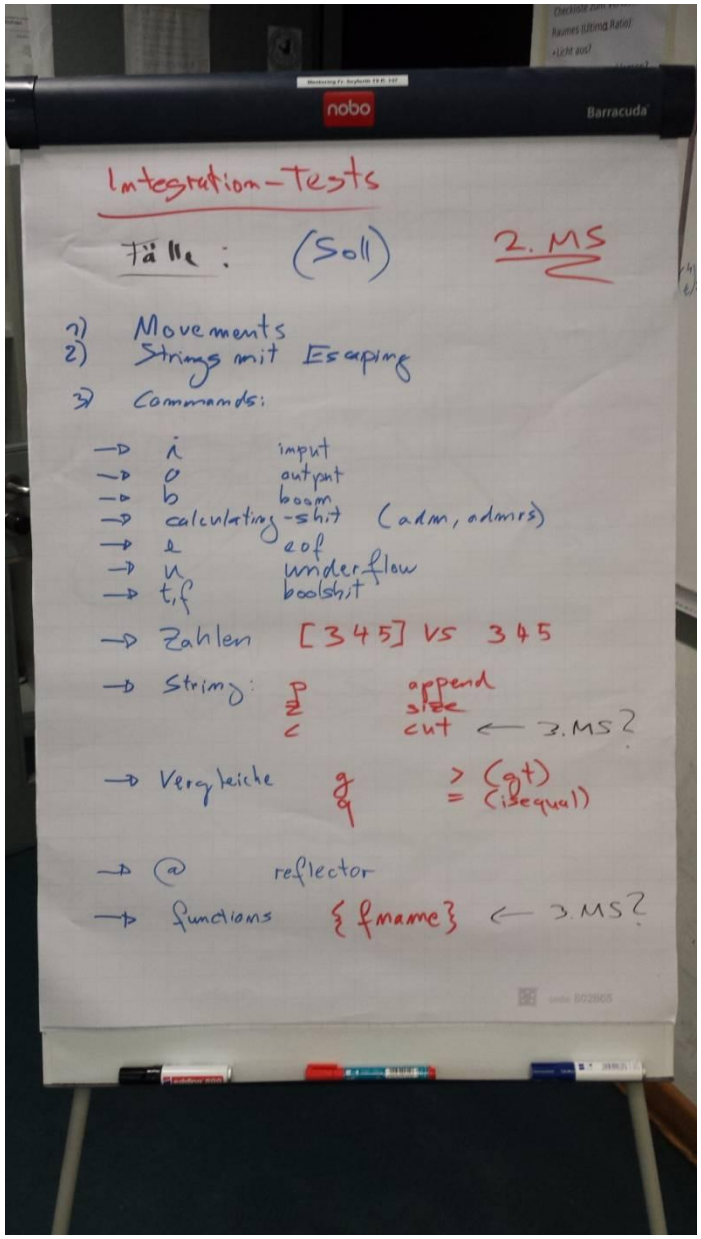
- **Debugging**
  - verwendet Lexer, ermittelt die Lexeme, matched auf die Lexeme
  - hinterlegt den aktuellen Zustand in der TextAreaContent
    - Daten-Stack
    - Funktions-Stack
    - Variablenbelegung

Inhalte

Testing

- **H-Unit**
  - testet Funktionen der einzelnen Module des Compilers
  - nicht für alle Module praktikabel
    (z.B. Zwischencode-Erzeugung)
- **Integration-Tests**
  - testet die gesamte Compiler-Pipeline
  - > 246 Tests (45% positive Tests, 55% negative Tests)
  - Travis CI *(Continuous Integration)*
- **Cross-Testing** *(…später mehr)*





Inhalte

Gruppenorganisation

Framework:
- Nicolas Lehmann

Dokumentation:
- Tobias Kranz
- Michal Ajchman

Testing + Preprocessor:
- Christopher Pockrandt
- Nicolas Lehmann
- Marcus Hoffmann

Lexer:
- Christian Hofmann
- Tilman Blumenbach
- Tobias Kranz

Syntactical Analysis:
- Kristin Knorr
- Marcus Hoffmann

Semantical Analysis:
- Christian Hofmann
- Tudor Soroceanu

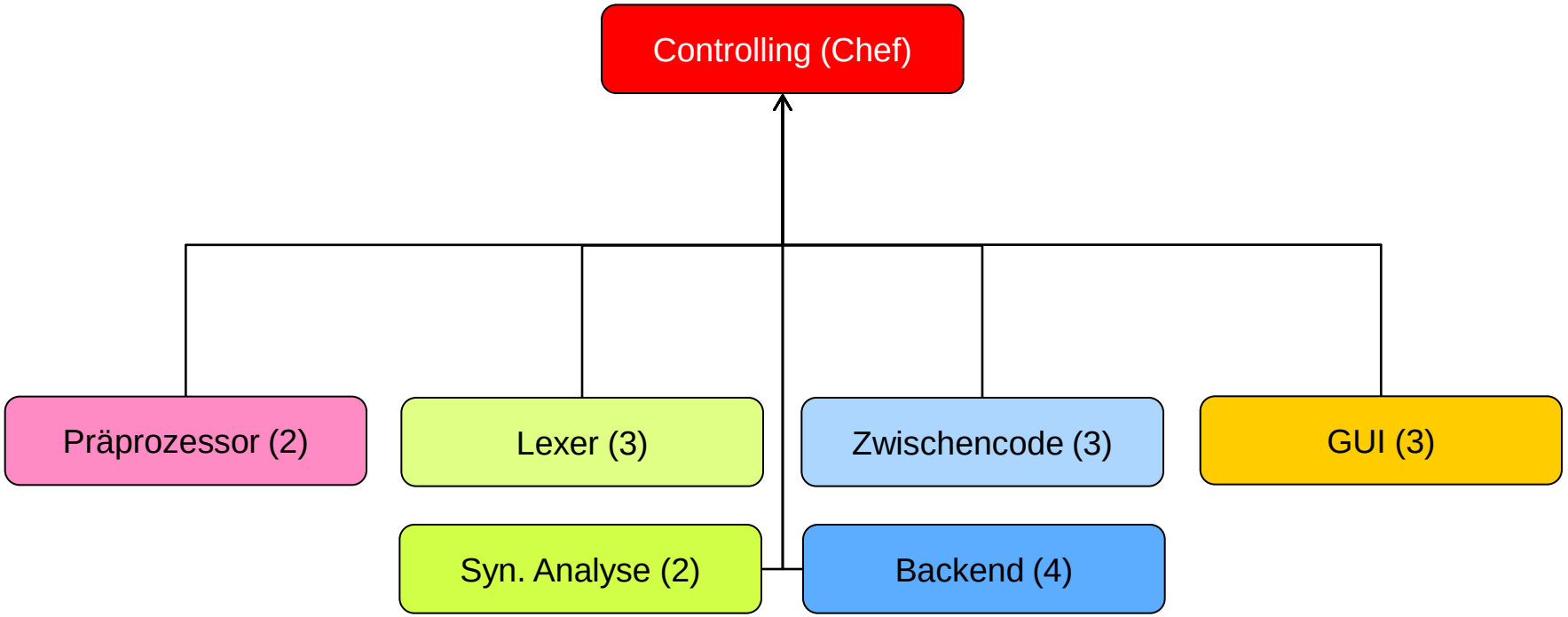
Intermediate Code:
- Philipp Borgers
- Lyudmila Vaseva
- Michal Ajchman

Backend:
- Tudor Soroceanu
- Tilman Blumenbach
- Maximilian Claus
- Sascha Zinke

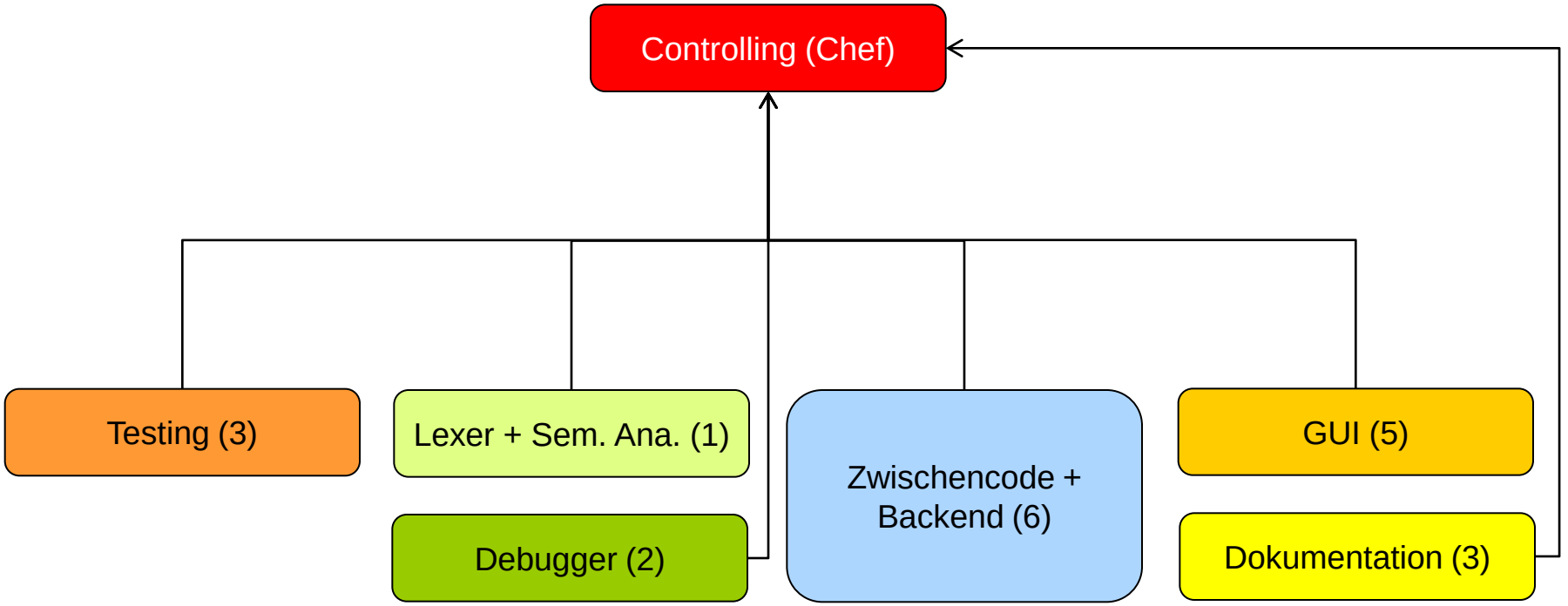
Editor:
- Christoph Graebnitz
- Kelvin Glaß
- Benjamin Kodera
- Kristin Knorr
- Christian Hofmann

Inhalte

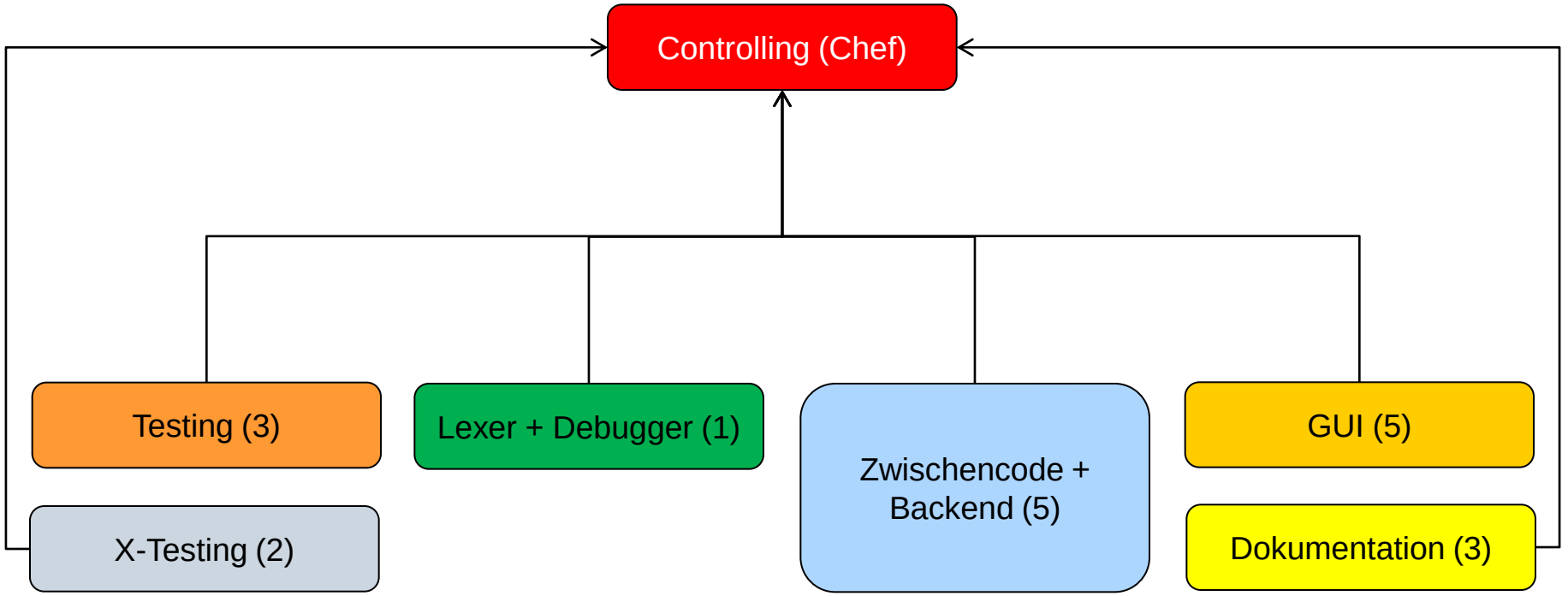
Gruppenhierarchie: 1. Meilenstein



Gruppenhierarchie: 2. Meilenstein



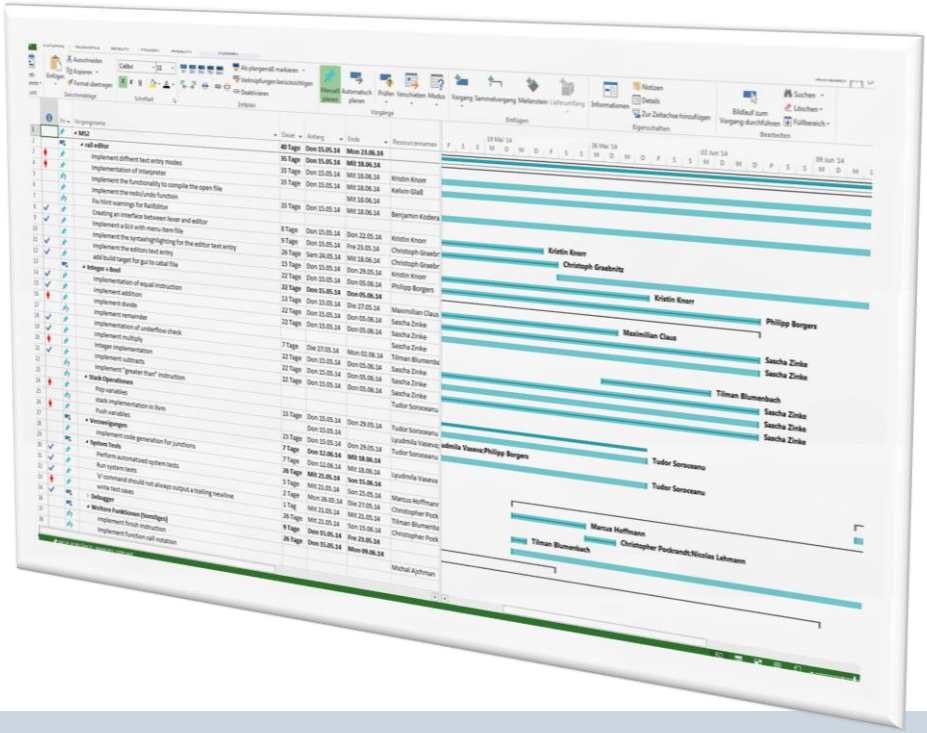
Gruppenhierarchie: 3. Meilenstein



Inhalte

Herausforderungen: Projektmanagement

- **GitHub**
  - Tickets gut für die Strukturierung von Tasks
  - Tickets schlecht fürs Controlling
  - nur eine Person zuweisbar
  - Abhängigkeiten von Tickets schlecht modellierbar
  - Probleme bei der Verwendung von GitHub wurden gelöst durch engagierte Git-Master
- **keine Userstories !**
- **Microsoft Project**
  - Controlling



Herausforderungen: Techn. Anforderungen

- **Cabal**
  - Abhängigkeiten (dependencies) auf verschiedenen Plattformen

- **Travis CI**

- **H-Unit**

Herausforderungen: Kommunikation

1. Meilenstein

- hauptsächlich eMail-Kommunikation, sehr unstrukturiert
- wichtige Informationen gingen verloren, da zu viele eMails
- kaum persönliche Kommunikation

2. Meilenstein

- Kommunikation über Ticketsystem
- Kommunikation in die Untergruppen verlegt
- Gruppentreffen, IM (Skype, Jabber), Teamviewer

3. Meilenstein

- Pair-/Group-Programming
- Kommunikation mit der C++ Gruppe (Teamspeak)

Inhalte

Was haben wir gelernt?

- **Software**
  - einen eigenen Compiler bauen
  - wie wichtig Modularität (hohe Kohäsion, geringe Kopplung) ist
  - Design-Entscheidungen beeinflussen massiv den Projektverlauf
  - Haskell *(insbesondere die allseits gefürchteten Monaden!!!)*, LLVM
  - Build-Tool „Cabal“
  - Continuous Integration-Tool „Travis CI“

- **Projektarbeit**
  - wie wichtig Kommunikation ist *!!!*
  - wie wir verschiedene Kommunikationsmittel für unterschiedliche Zwecke einsetzen können
  - wann und für welchen Zweck wir bestimmte Kommunikationsmittel einsetzen sollten
  - Strukturierung und Steuerung von Projektarbeit *(Projektmanagement, HRM)*

Inhalte

Ausblick

- **Compiler**
  - nicht-sequenzielle Rail-Programme
  - Semaphoren
- **Editor**
  - automatisches GUI-Testen
  - Crash-Tests
  - graphischer Editor (Graphiken bereits vorhanden)
    - Syntax-Highlighting für Schienen
    - Debugging: fahrender Zug
    - Schienen mit Mouse-Bewegung malen
- **zukünftige Rail-Projekte**
  - „Rail goes mobile“ ?

Softwareprojekt Übersetzerbau
SoSe 2014

Das war‘s!

Vielen Dank für Ihre Aufmerksamkeit!

Fragen? Fragen!