



# Softwareprojekt Übersetzerbau SoSe 2014



**Abschlusspräsentation**  
Haskellgruppe  
Rail-LLVM-Compiler

# Inhalte

## 1 Software

### 1.1 Rail-Spezifikation

### 1.2 Funktionsumfang

### 1.3 Screencast

### 1.4 Architektur

#### 1.4.1 Rail-LLVM-Compiler

#### 1.4.2 Rail-Editor

#### 1.4.3 Testing

## 2 Projektarbeit

### 2.1 Gruppenorganisation

### 2.2 Projekthierarchie

### 2.3 Herausforderungen

## 3 Fazit

### 3.1 Was haben wir gelernt?

### 3.2 Ausblick



# 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 Funktionen können auf den Stack gepusht oder vom Stack gepoppt werden
  - *isList(x) ↔ ¬ isString(x)*

# Inhalte

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang**
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie
- 2.3 Herausforderungen

## 3 Fazit

- 3.1 Was haben wir gelernt?
- 3.2 Ausblick



# Funktionsumfang: **Compiler**

- **Movements**
- **Strings**
- **Integer & Float + arithm. Operationen**
- **Boolean**
- **Variablen**
- **Funktionsaufrufe & anonyme Funktionen**
- **Listen**
- **Kommando-Befehle**
  - i, o, e
  - b, u, ?
  - p, c, s



*input, output, end of file*  
*boom, underflow, type check*  
*append, cut, size*

**... und alles mit Fehlerbehandlung !!!**

- invalide Programme werden als solche erkannt

# Funktionsumfang: Editor

- Dateien öffnen / speichern
- **3 Modi**
  - insert-mode
  - replace-mode
  - smart-mode
- **Programm ausführen**
  - input / output über Editor
- **debug-mode**
  - step by step
  - breakpoints
  - Daten- und Funktionsstack-Ausgabe
- **dynamische Erweiterung der TextArea**

... und alles in Haskell !!!

- „haskell la vista, baby“

*normaler Eingabemodus  
überschreibt Zeichen  
folgt der Schiene*



# Inhalte

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast**
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

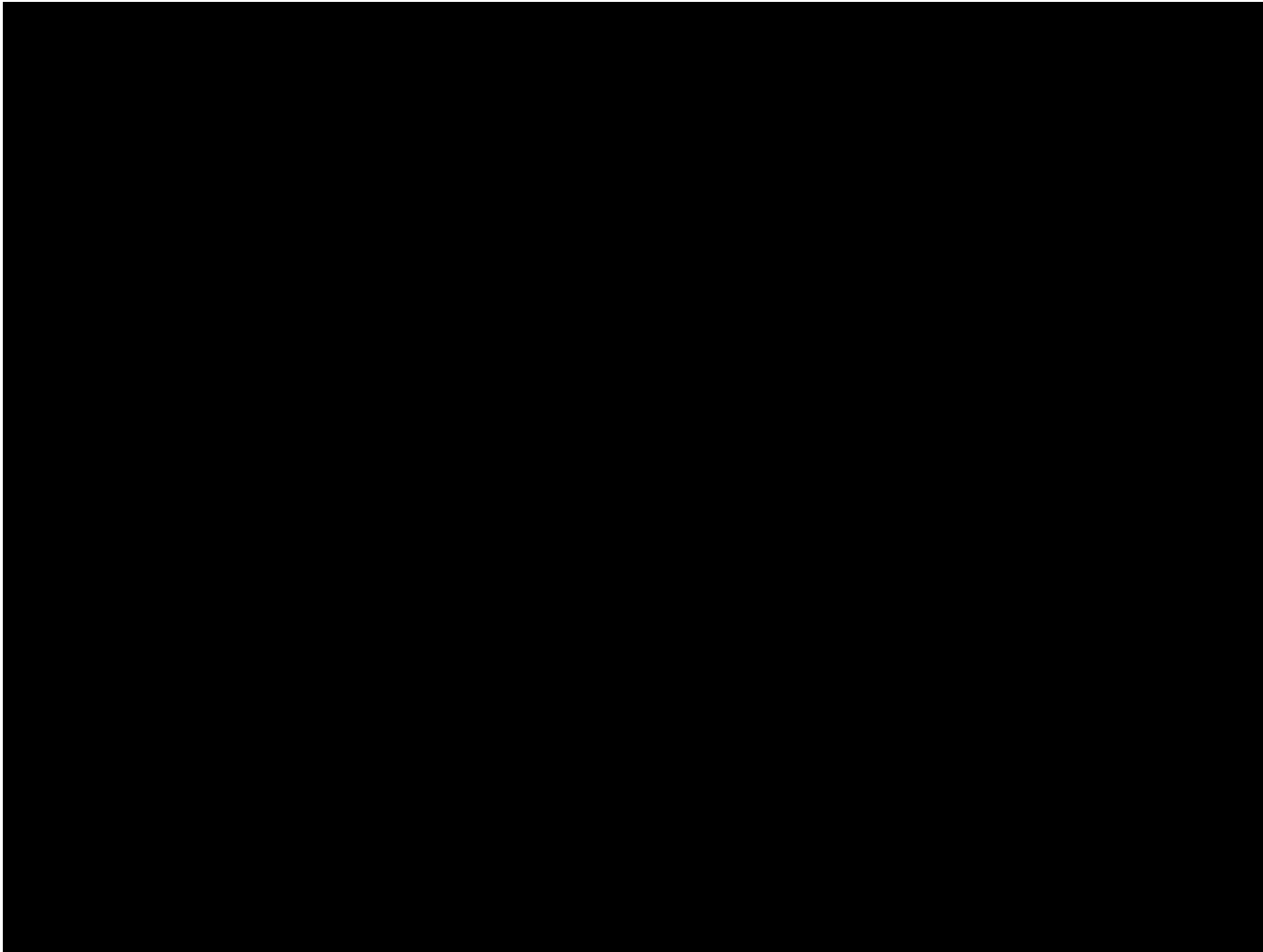
- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie
- 2.3 Herausforderungen

## 3 Fazit

- 3.1 Was haben wir gelernt?
- 3.2 Ausblick

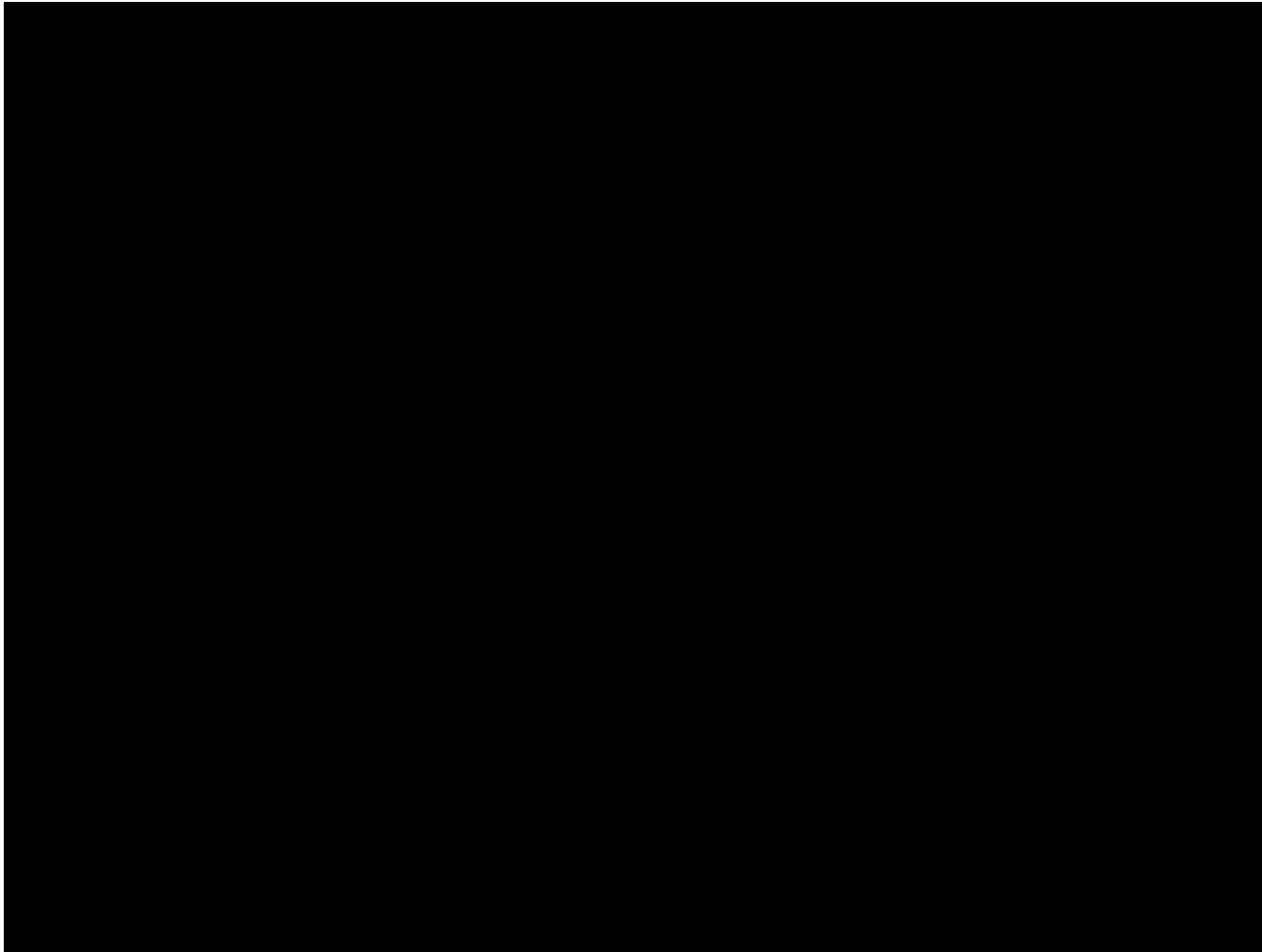


# Screencast: **Compiler**





# Screencast: Editor



SUPER!



# Inhalte

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

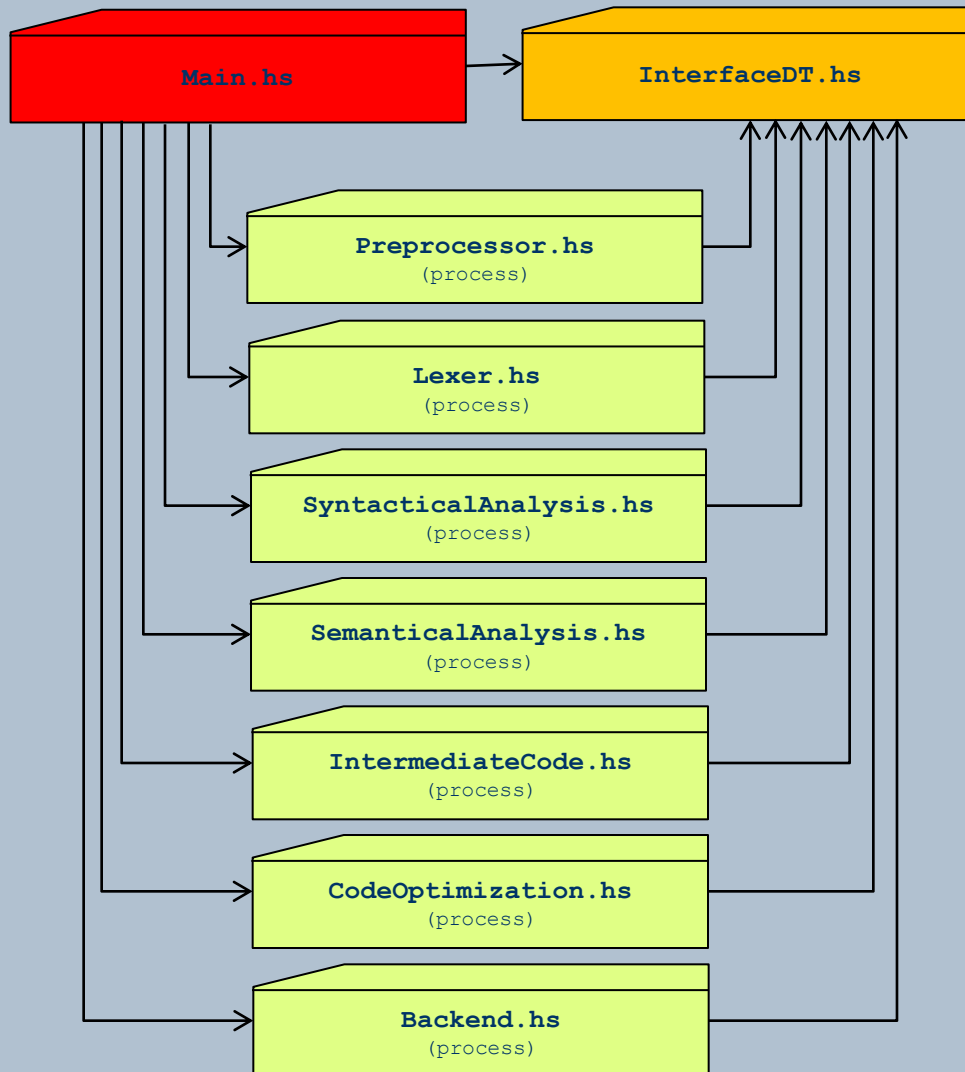
- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie
- 2.3 Herausforderungen

## 3 Fazit

- 3.1 Was haben wir gelernt?
- 3.2 Ausblick

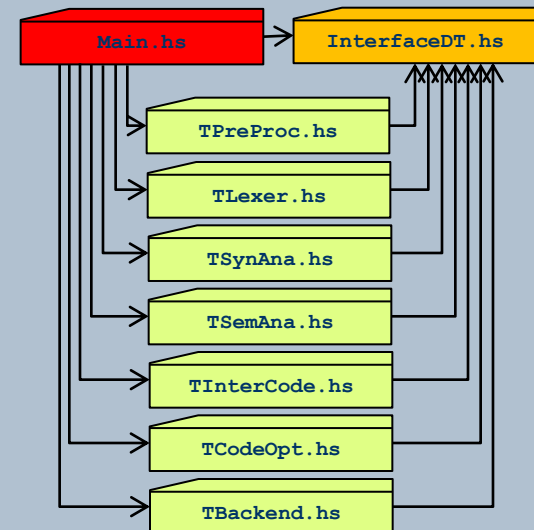


## Programm

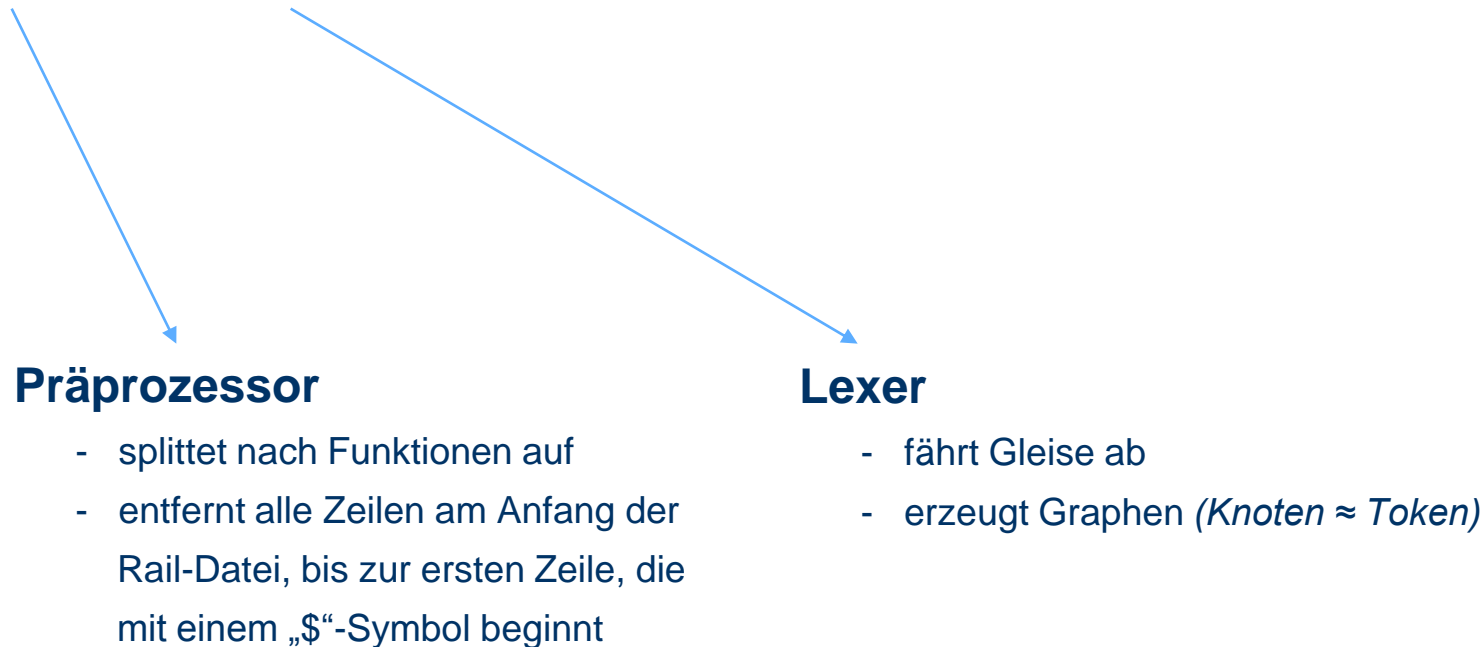


- ✓ Main startet den Rail-LLVM-Compiler
- ✓ InterfaceDT definiert algebraische Datentypen für die Datenübergabe zwischen den Submodulen der Compiler-Pipeline
- ✓ Preprocessor, Lexer, SyntacticalAnalysis, SemanticalAnalysis, IntermediateCode, CodeOptimization und Backend lösen jeweils ihr Teilproblem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

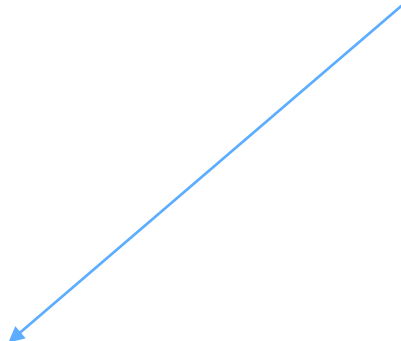
## Tests



# Compiler-Pipeline



# Compiler-Pipeline



## Syntaktische Analyse

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



## Semantische Analyse

- verwirft ungültige Pfade
- Umwandeln von anonymen Funktionen in normale Funktionen

# 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**
  - de facto Standard für Compiler Backends
  - plattformunabhängig (x86, ARM, PowerPC,...)
  - „just in time“-Übersetzung möglich
  - sehr effizient
  - hardwareunabhängig
- **LLVM-Eigenschaft**
  - „static single assignment form“
  - statisches Typsystem
  - Funktionen
  - sehr aufwändig manuell zu programmieren

# Compiler: Laufzeitbibliothek

- Rail ist eine Stack-basierte Sprache
- 3 Möglichkeiten zur Implementierung des Stacks
  - nativen (LLVM-)Stack benutzen
  - statisch allozierter Stack (als Feld)
  - dynamischer Stack (als verkettete Liste)
- Rail benötigt einen Garbage-Collector
  - bedingt durch LISP-artige Listen
  - zwei naheliegende Möglichkeiten:
    - reference counting
    - mark & sweep



# Compiler / Backend: Wie funktioniert's?

## - Eingabe

- Haskell-Code:

```
-- (FunctionID, [(PathID (start: 1), List of lexemes, 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

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

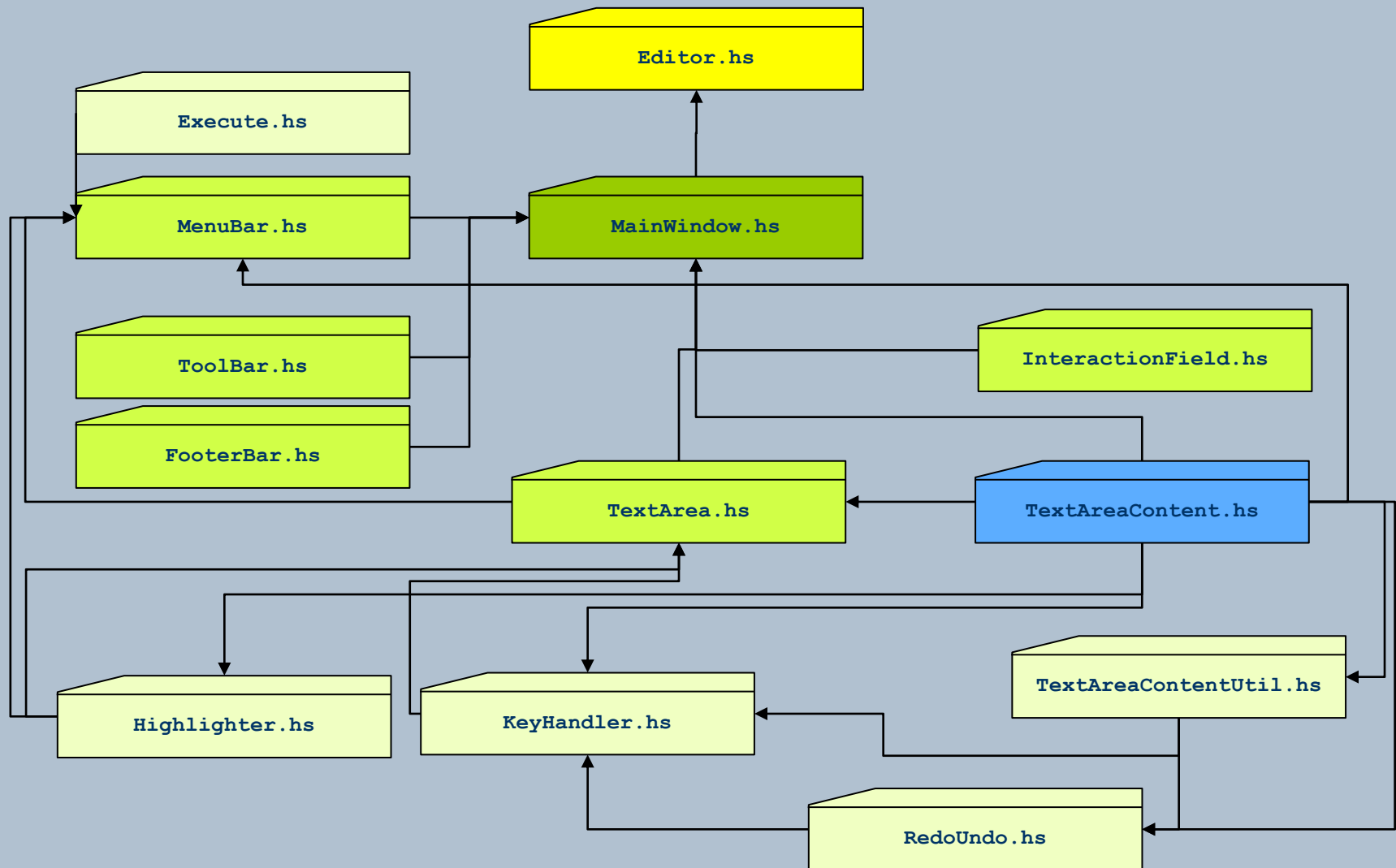
- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie
- 2.3 Herausforderungen

## 3 Fazit

- 3.1 Was haben wir gelernt?
- 3.2 Ausblick



## Editor



# Editor: **TextArea**

- **Problem: Standard-Textview**
  - wenige Konfigurationsmöglichkeiten
- **Eigenimplementierung eines Textviews**
  - 1. Version: Tabelle von Entries (einzeilige Eingabefelder)
  - 2. Version: Felder in einer Drawing-Area
- **Eigenimplementierung **aller** Verhaltensweisen eines Standard-Editors**
- **Trennung von Darstellung und Inhalt**
  - Darstellung: TextArea
  - Inhalt: TextAreaContent

Teuer!



# Editor: **TextAreaContent**

## - zusammengesetzte Datenstruktur

- Map für Zeichen
- Map für Farben
- ActionQueues für RedoUndo
- Interpreterkontext

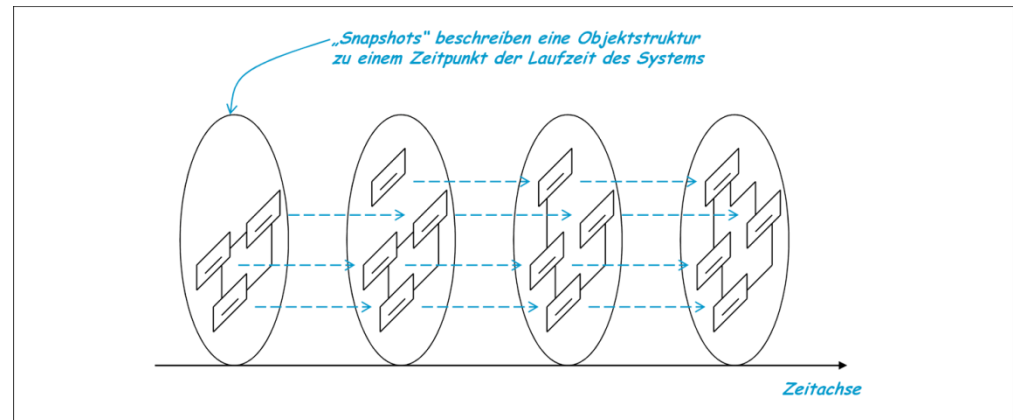
## - stellt zur Verfügung

- 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

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie
- 2.3 Herausforderungen

## 3 Fazit

- 3.1 Was haben wir gelernt?
- 3.2 Ausblick





# Testing

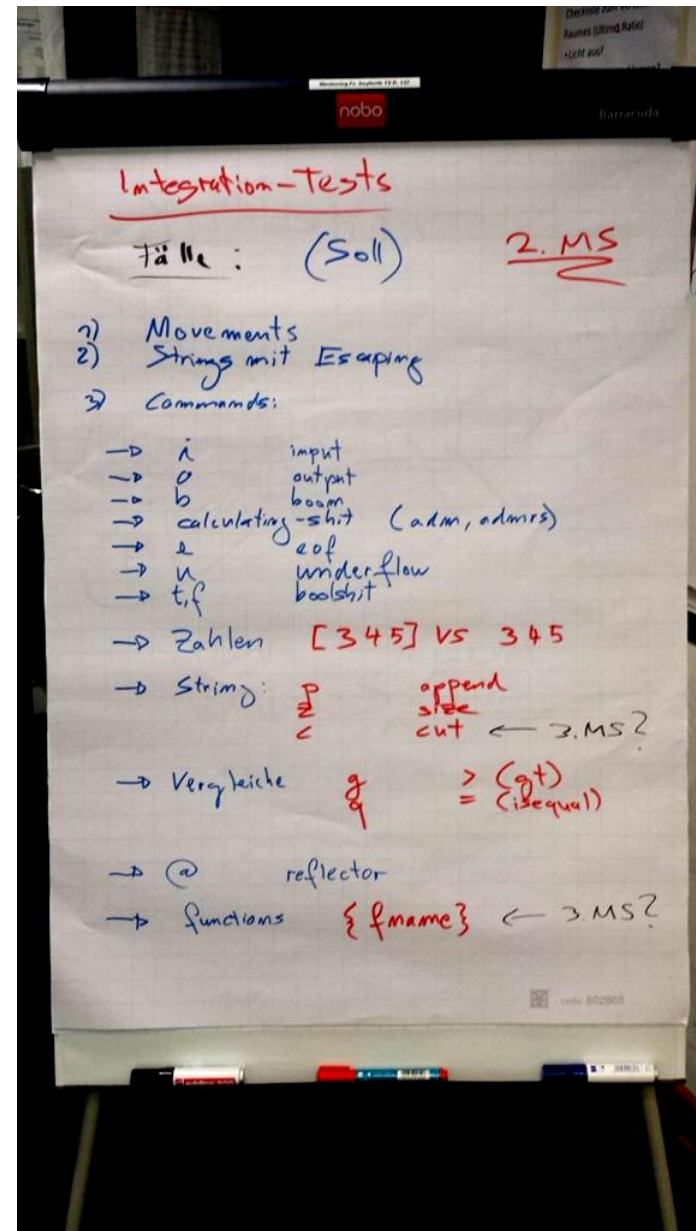
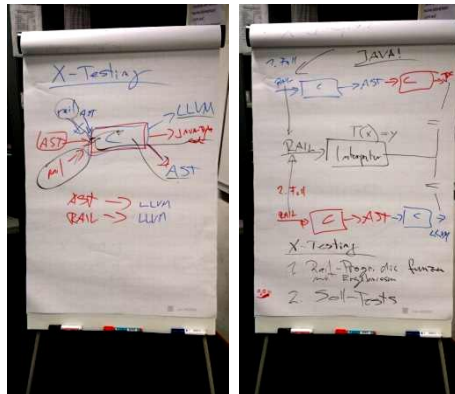
## - H-Unit

- 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

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

- 2.1 **Gruppenorganisation**
- 2.2 Projekthierarchie
- 2.3 Herausforderungen

## 3 Fazit

- 3.1 Was haben wir gelernt?
- 3.2 Ausblick



# 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 Koderer
- Kristin Knorr
- Christian Hofmann

# Inhalte

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

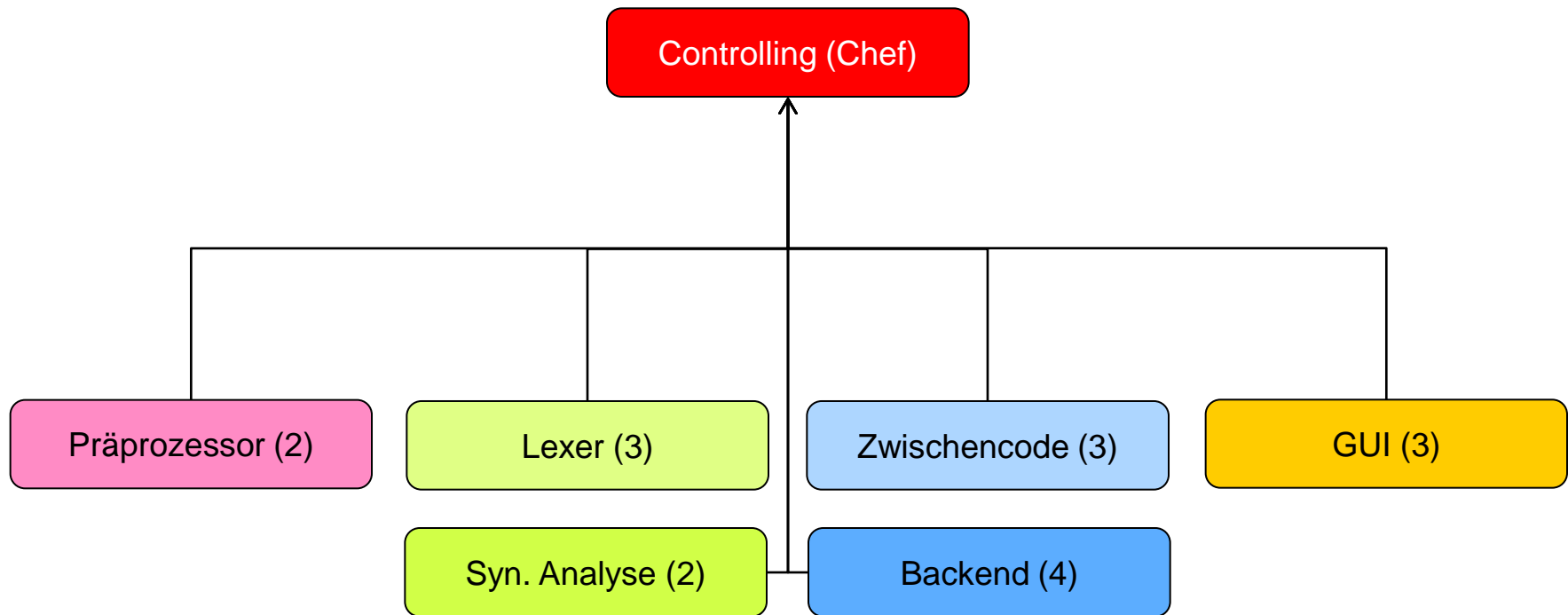
- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie**
- 2.3 Herausforderungen

## 3 Fazit

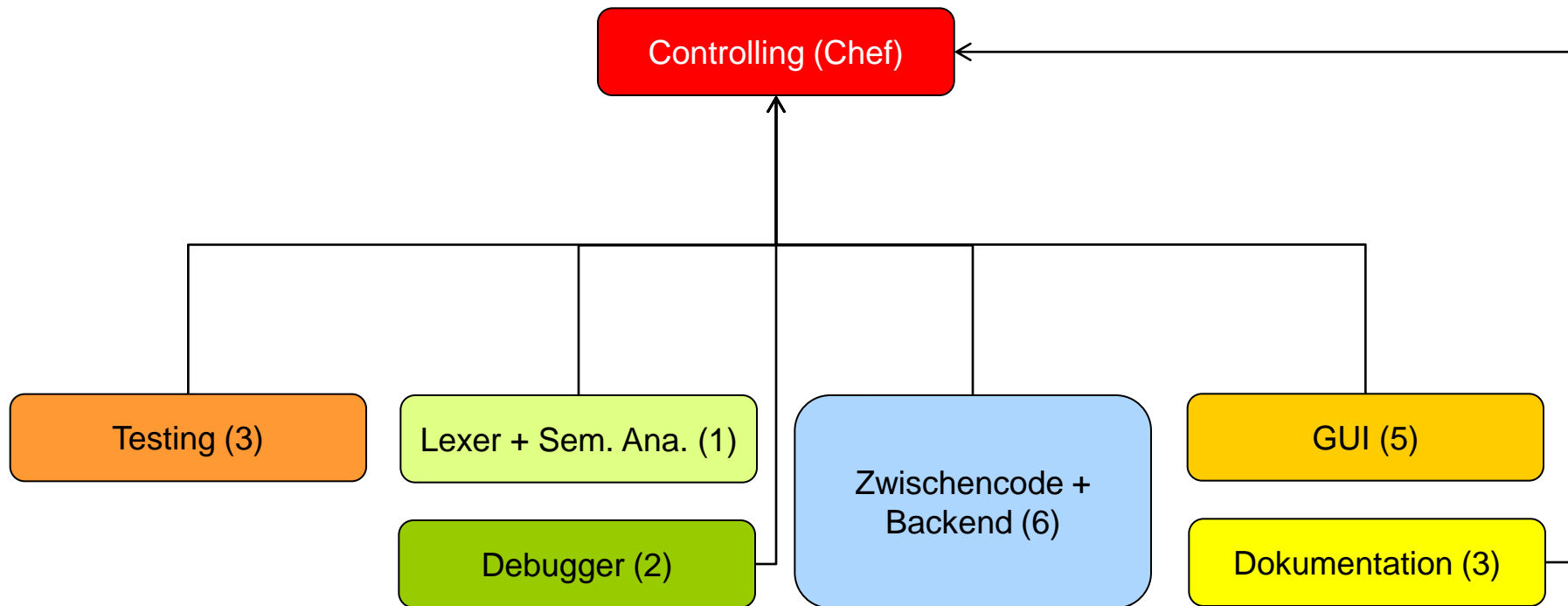
- 3.1 Was haben wir gelernt?
- 3.2 Ausblick



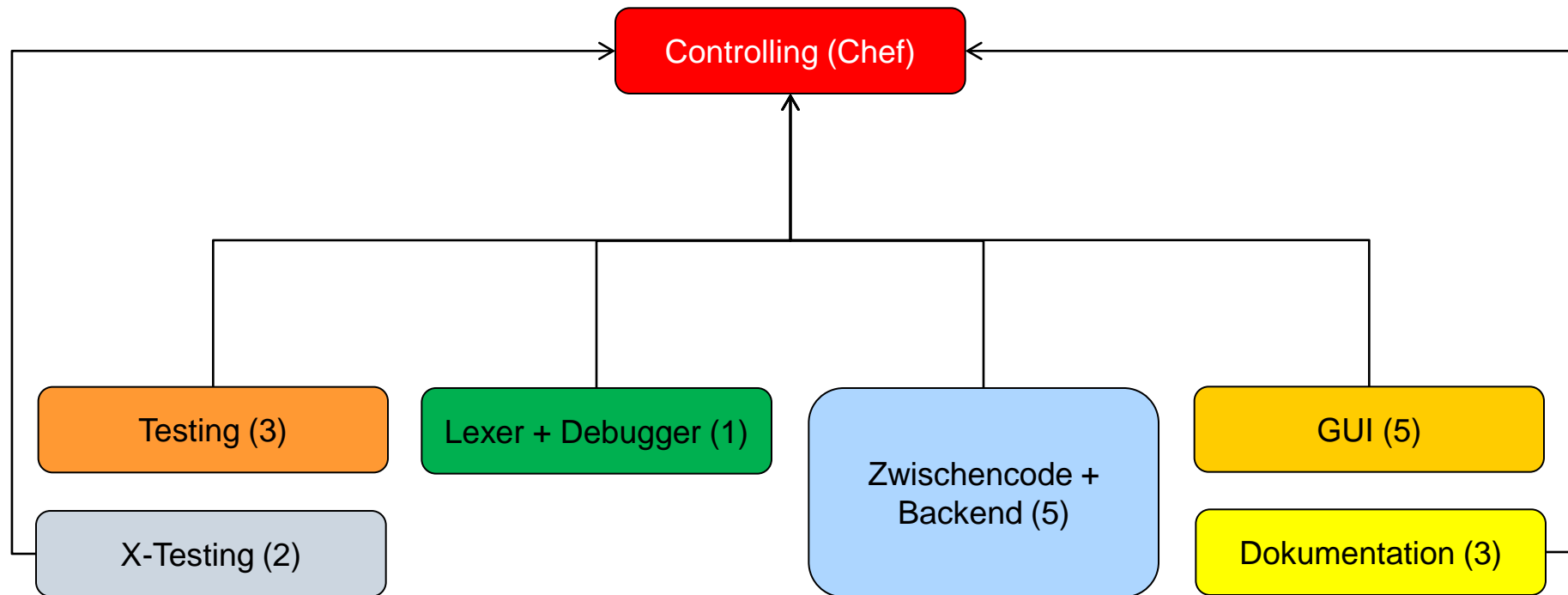
# Gruppenhierarchie: 1. Meilenstein



# Gruppenhierarchie: 2. Meilenstein



# Gruppenhierarchie: 3. Meilenstein



# Inhalte

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie
- 2.3 Herausforderungen**

## 3 Fazit

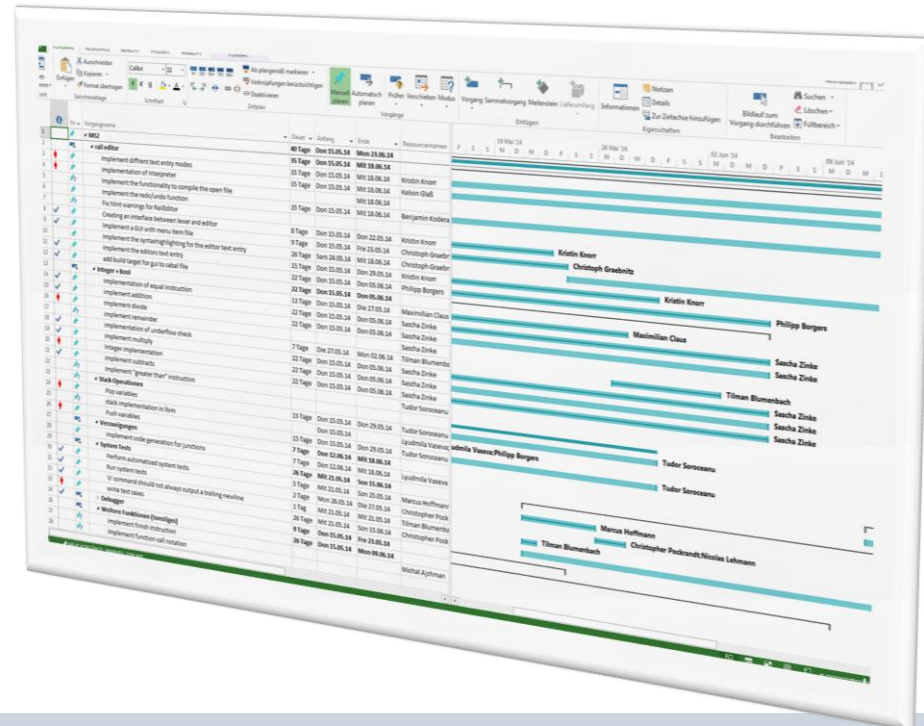
- 3.1 Was haben wir gelernt?
- 3.2 Ausblick





# 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

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie
- 2.3 Herausforderungen

## 3 Fazit

- 3.1 Was haben wir gelernt?
- 3.2 Ausblick



# 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**
- Haskell-Tooling: „**Cabal**“, **Haskell Profiling**, **Haddock**, **HLint**
- 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

## 1 Software

- 1.1 Rail-Spezifikation
- 1.2 Funktionsumfang
- 1.3 Screencast
- 1.4 Architektur
  - 1.4.1 Rail-LLVM-Compiler
  - 1.4.2 Rail-Editor
  - 1.4.3 Testing

## 2 Projektarbeit

- 2.1 Gruppenorganisation
- 2.2 Projekthierarchie
- 2.3 Herausforderungen

## 3 Fazit

- 3.1 Was haben wir gelernt?
- 3.2 **Ausblick**



# Ausblick

## - Compiler

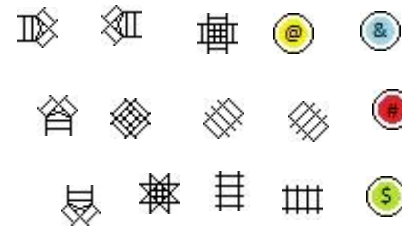
- nicht-sequenzielle Rail-Programme
- Semaphoren

## - Editor

- automatisches GUI-Testen
- Crash-Tests
- graphischer Editor (Graphiken bereits vorhanden)
  - Syntax-Highlighting für Schienen
  - Debugging: fahrender Zug
  - Schienen mit Maus-Bewegung malen

## - zukünftige Rail-Projekte

- „Rail goes mobile“ ?







# Softwareprojekt Übersetzerbau SoSe 2014



**Das war's!**

Vielen Dank für Ihre Aufmerksamkeit!

**Fragen? Fragen!**

