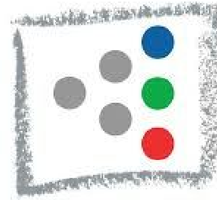
HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL

FHO Fachhochschule Ostschweiz



INSTITUTE
FOR
SOFTWARE

# Functional Reactive Programming

# Studienarbeit

Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2015


Autoren: Florian Merz, Royce Manavalan
Betreuer: Prof. Dr. Josef Joller

- Aufgabenstellung
    - Aufgabe
    - Erwartete Resultate
- Abstract
- Management Summary
- Einleitung
- Planung
- Technischer Bericht
    - Kurzbeschreibung
    - Anforderungen
        - Aufbau
        - Landschaft
        - Spiel
        - Levels
        - Optional
        - Kollision
        - Spielerbewegungen
        - Spiel Objekte
    - Use Cases
        - Use Case “Level wechseln”
        - Use Case “Würfel springen”
    - Entwicklungsumgebung
        - Eclipse Haskell Plugin
        - Emacs Haskell Plugin
        - Leksah
        - Sublime Haskell Plugin
    - Technologien
        - Yampa
        - SDL
    - Architektur
        - Module
        - Detaillierte Modul Beschreibung
        - Domain Model
    - Yampa Funktionen
        - Switch
        - Edge
    - Game Logik
        - Beschreibung
            - GameSF
            - GameLevelSF

        - GameLevelDoneSF
        - GameSessionSF
        - FallingPlayerSF
        - DrivingPlayerSF
    - Testing
- Ergebnisse
- Schlussfolgerung
- Ausblick
- Installationsanleitung
    - Ubuntu Prerequisites
    - Clean-Up
    - Set Path
    - GHC
    - Cabal (Package Manager für Haskell)
        - cabal
        - cabal-install
    - Stack (neuer Package Manager und Build Tool)
    - SDL (OpenGL library)
        - SDL2
        - SDL2-TTF (Font)
    - SublimeText Haskell
    - Entwicklung
- Glossar
- Literaturverzeichnis
- Eigentständigkeitserklärung
- Nutzungsrechte
    - Gegenstand der Vereinbarung
    - Urheberrecht
    - Verwendung

# 1. Aufgabenstellung

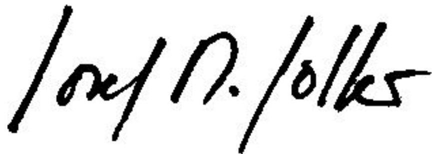
## 1.1. Aufgabe

- Einarbeiten in die funktionale Programmierung
- Erarbeitung einer Übersicht über existierende functional Reactive Programming (FRP) Frameworks (Reactive Banana, Sodium, Reactive, Yampa, Netwire)
- Einarbeiten in die FRP mit Hilfe eines FRP Packages (Tutorial)
- Umsetzung der Erkenntnisse in einem einfachen Spiel

## 1.2. Erwartete Resultate

Als erstes geht es darum, das Functional Reactive Programming (FRP) Programmierparadigma mit Hilfe der Programmiersprache Haskell zusammen mit einem FRP-Framework zu erlernen.

Dann wird das Gelernte zum Erstellen eines interaktiven Spieles angewendet.
Das Ergebnis ist ein Showcase für FRP.



Prof. Dr. Josef Joller
18.12.2015, Rapperswil

# 2. Abstract

Im Rahmen vom SE2-Modul haben wir zum ersten Mal Einblick in die funktionale Programmierwelt erhalten. Das Interesse daran wurde geweckt. Die Semesterarbeit kam gerade zur rechten Zeit, um den Einstieg in die funktionale reactive Programmierwelt zu finden. Raus aus der imperativen Programmierwelt, stiegen wir mit wenig Erfahrung in eine neue Welt ein.

Haskell empfahl sich als gute Sprache zum Einstieg in die funktionale Welt. Zu Beginn haben wir uns mit mehreren Tutorials auseinander gesetzt. Theoretisch klang alles sehr praktisch und gut. Bei der Implementierung der Prototypen zeigten sich dann die Tücken der Sprachen. Als Framework für die Abstrahierung von Zeit und Events wurde Yampa verwendet. Zur Visualisierung kam die SDL-Library zum Zuge.

Das Ergebnis ist ein funktionsfähiges Haskell Spiel. Das Spiel erkennt Eingaben von Nutzern, kann dementsprechend darauf reagieren und gibt die Ausgaben dazu auf dem Bildschirm aus. Das Ziel wurde damit erreicht, ein funktionierendes Spiel und ein gelungener Einstieg in die FRP-Welt.

# 3. Management Summary

Im Rahmen vom SE2-Modul haben wir zum ersten Mal Einblick in die funktionale Programmierwelt erhalten. Bei diesem Programmierparadigma ist es wesentlich, dass die ganze Applikation nur aus Funktionen besteht, Nebenwirkungen sind nicht wie in der imperativen Programmierwelt leicht zu erwirken.
Das Interesse daran wurde geweckt. Die Semesterarbeit kam gerade zur rechten Zeit, um den Einstieg in die funktionale reactive Programmierwelt zu finden. Raus aus der imperativen Programmierwelt, stiegen wir mit wenig Erfahrung in eine neue Welt ein.

Haskell empfahl sich als gute Sprache zum Einstieg in die funktionale Welt. Zu Beginn haben wir uns mit mehreren Tutorials auseinander gesetzt. Theoretisch klang alles sehr praktisch und gut. Bei der Implementierung der Prototypen zeigten sich dann die Tücken der Sprachen. Als Framework für die Abstrahierung von Zeit und Events wurde Yampa verwendet. Zur Visualisierung kam die SDL-Library zum Zuge.

Das Ergebnis ist ein funktionsfähiges Haskell Spiel. Das Spiel erkennt Eingaben von Nutzern, kann dementsprechend darauf reagieren und gibt die Ausgaben dazu auf dem Bildschirm aus. Das Ziel wurde damit erreicht, ein funktionierendes Spiel und ein gelungener Einstieg in die FRP-Welt.

Die Applikation lässt sich beliebig ausbauen. Die Spiellogik kann einfach um Features erweitert werden, da die Grundstruktur der Applikation steht. Mit der Dokumentation zusammen kann es als kleines Tutorial dienen, für diejenigen die in diese Welt einsteigen möchten.

# 4. Einleitung

Unsere Semesterarbeit kann als kleines Forschungsprojekt angesehen werden. Bei dieser Arbeit handelt es sich weniger um das Produkt, sondern um die Erkenntnisse aus dem Projekt. Das Projekt dient als Einstieg in das Functionale Reactive Programming. Funktionale Programmiersprachen unterscheiden sich sehr zu den bisher gelernten imperativen Programmiersprachen.  Zum Einstieg in das neue Thema verwenden wir die Programmiersprache Haskell.

# 5. Planung

| | | Inception | | | Elaboration | | | | | Construction | | | | Transition | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | W38 | W39 | W40 | W41 | W42 | W43 | W44 | W45 | W46 | W47 | W48 | W49 | W50 | W51 |
| **Arbeitspaket** | **Soll/Ist** | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 1. Einlesung in Haskell | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 2.Projektidee besprechen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 3. Einlesung in Haskell Game Programmierun | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 4. Projektsetup erstellen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 5. Requirements definieren | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 6. Prototype | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 6.1 Prototype Uhr | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 6.2 Prototype GUI | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 6.3 Prototype Logik | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7. Implementierung | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.1 Datentypen definiert | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.2 Levels parsen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.3 Input mappen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.4 Output mit SDL | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| **7.5 Spiellogik** | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.5.1 Hintergrund bewegen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.5.2 Ende der Welt überprüfen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.5.3 Springen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.5.4 Fallen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 7.5.5 Kollision überprüfen | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 8. Dokumentation | Ist | | | | | | | | | | | | | | |
| | Soll | | | | | | | | | | | | | | |
| 9. Abgabe | Ist | | | | | | | | | | | | | | |

Bei der Planung haben wir anfangs die Phasen grob eingeteilt und gewisse Arbeitspakete geschätzt (Blau). Während der Umsetzung konnten wir die vorgegebenen Zeiten sehr gut einhalten, da sie nicht zu detailliert geschätzt wurden. Die effektiven Zeiten sind grün dargestellt.

Das Projektmanagement fiel in diesem Projekt klein und fein aus.

# 6. Technischer Bericht

## 6.1. Kurzbeschreibung

Wir entwickeln ein kleines “Jump & Run” - Spiel, bei dem der Spieler Hindernisse umgehen muss, um an sein Ziel zu kommen. Das Spiel verfügt über mehrere Levels, die von Zeit zur Zeit schwieriger werden. Der Spieler hat die Möglichkeit, über eine Eingabe die Spielfigur zum springen zu bringen. Man hat die Möglichkeit, selber Levels zu kreieren.

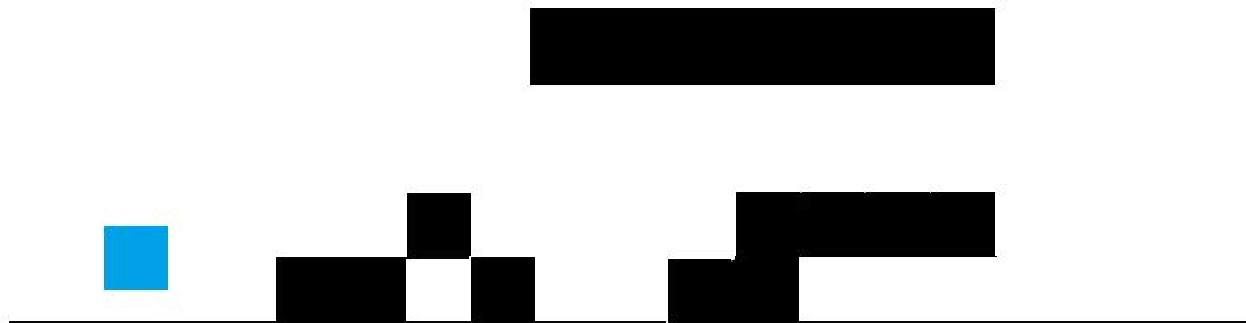
## 6.2. Anforderungen

### 6.2.1. Aufbau

Die Applikation besteht aus zwei abstrakten Objekten, dem Boden und den Hindernissen. Der Boden wird mit Würfeln repräsentiert, wobei in diesem Fall nur die obere Oberfläche als Boden dient. Gleichzeitig können die Würfel als Hindernisse dienen. Ein weiteres Hindernis ist Lava, Hindernisse können später leicht hinzugefügt werden. Eine Kollision entsteht, wenn eine Spielfigur ein Hindernisse berührt. Die einzige Ausnahme stellt den Würfel dar, auf seiner oberen Oberfläche kann die Spielfigur weiter rennen.

### 6.2.2. Landschaft

Die Landschaft wird in einem Textfile definiert. Die Zeilen stehen für die Y-Achse, die letzte Zeile stellt den tiefsten Y-Wert dar(X,0). Die Spalten stehen für die X-Achse, die erste Spalte steht bei (0,Y). Jede Zelle wird mit einem Wert befüllt, der für ein Spielobjekt steht. Die Landschaft wird dabei Spaltenweise geparsed.



Die Landschaft wurde zuerst mit Zeichnungen über SDL aufgebaut. Die Spielfigur und die Hindernisse wurden mit Würfeln dargestellt.
Eine weitere Option besteht darin, die Spiellandschaft als Grafik zu importieren und auf die Hindernisse zu mappen.

### 6.2.3. Spiel

Das Spiel hat mehrere Levels, in welchen die Spiellandschaft definiert wird.
Die Landschaft zieht an der Spielfigur vorbei, die Spielfigur hat die Möglichkeit zu springen.

### 6.2.4. Levels

Jedes Level wird in einem Text-File definiert. Beim Spielstart werden alle Levels geparsed und im Spiel zur Verfügung gestellt.
Eine Option wäre, das Parsen zu parallelisieren. Vorerst kann man zwischen den Levels nur über die Level Anzahl wechseln. Bei genügend Zeit wird ein weiterer Screen erstellt, um die Levels visuel auszuwählen. Mit dieser Umsetzung sind wir flexibel bezüglich den Levels. Wir können selber Levels kreieren, ohne dass wir uns später um die Logik oder die Ausgaben kümmern müssen.
Am Anfang werden alle Levels freigeschaltet sein, eine weitere Option wird sein, dass die Levels freigeschaltet werden müssen, durch das meistern des aktuellen Levels.

### 6.2.5. Optional

Ein Spieler kann die Farbe seiner Spielfigur auswählen. Dazu müsste ein zusätzlicher Screen erstellt werden.

### 6.2.6. Kollision

Eine Kollision ergibt sich immer, wenn die Spielfigur unerlaubt in ein Spielobjekt reinfährt. Danach wird das Level neugestartet.

### 6.2.7. Spielerbewegungen

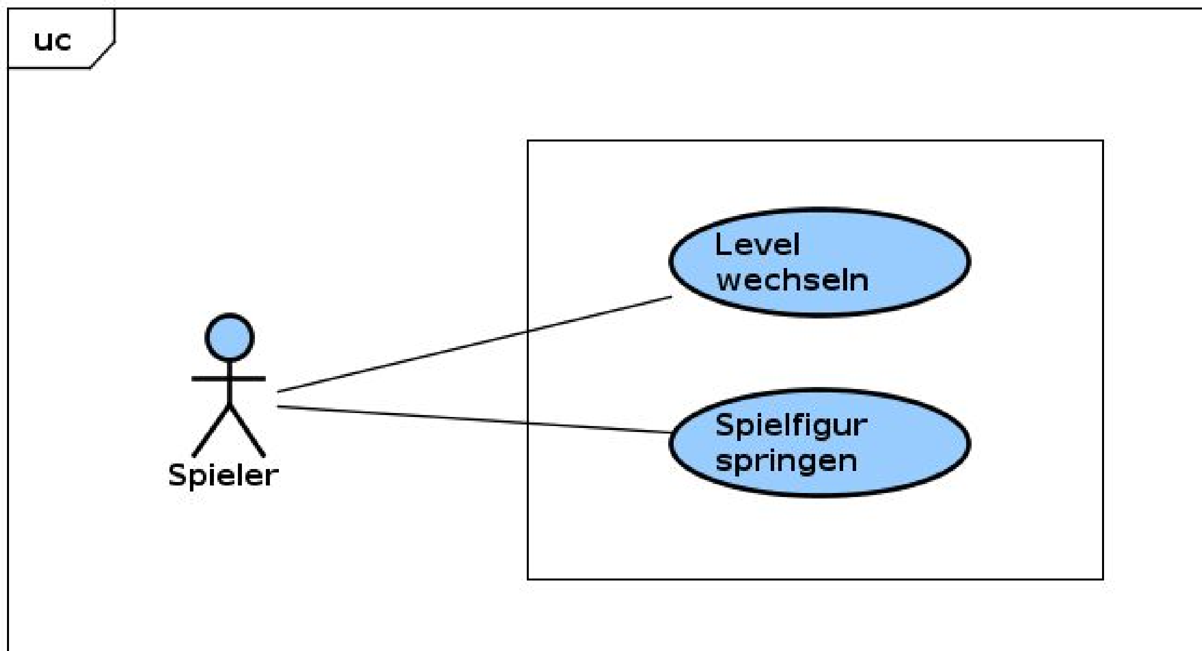
- Standardbewegung: Die Welt zieht am Spieler vorbei, so dass es den Eindruck erweckt, dass die X-Koordinate des Spieler sich ändert.
- Jumpbewegung: Der Spieler sieht ein Hindernis und drückt die Spacetaste, dabei wird der Spieler in die Luft springen.
- Fallbewegung: Nach dem Sprung fällt der Spieler wieder, bis er auf ein Objekt fällt, das befahrbar ist.

### 6.2.8. Spiel Objekte

In den Levels wird zwischen folgenden Objekten unterschieden, weitere Objekte können später problemlos noch hinzugefügt werden:

- Fahrbar (oben befahrbar, bei Berührungen auf den anderen Seiten ergibt sich jeweils eine Kollision)
    - Box
    - Boden
    - Wiese
- Freier Fall (Falls der Spieler sich in einem Luftobjekt befindet wird eine Fallbewegung eingeleitet)
    - Luft
- Hindernis (Eine Berührung mit dem Hindernis führt immer zu einer Kollision)
    - Lava
    - Speer

## 6.3. Use Cases



### 6.3.1. Use Case “Level wechseln”

Der Spieler hat während dem Spiel die Möglichkeit, zwischen den verschiedenen Levels zu wechseln, indem der Spieler die entsprechende Level Zahl auswählt.

### 6.3.2. Use Case “Würfel springen”

Während dem Spiel kann der Spieler die Spielfigur zum springen bringen.

## 6.4. Entwicklungsumgebung

### 6.4.1. Eclipse Haskell Plugin

Das Plugin für Eclipse heisst EclipseFP und kann hier bezogen werden:
http://eclipsefp.github.io/

Das Plugin bietet viele Features:
Auto-Build, Code-Compleation, In-Line Errors, Test Integration, Watch Variabeln,..

Das alles funktioniert einwandfrei für eine alte GHC / Cabal Version. Da wir in unserem Projekt jedoch mit dem neusten GHC / Cabal arbeiten werden mussten wir GHC aktualisieren. Danach ergaben sich viele Probleme. Ein essenzielles Package (BuildWrapper) konnte nicht mehr gebaut werden.

Der Entwickler des Plugins JP Moresmau hat das Projekt leider eingestellt:
http://jpmoresmau.blogspot.ch/2015/05/eclipsefp-end-of-life-from-me-at-least.html
Bis jetzt hat das Projekt noch niemand übernommen.
Somit ist diese Entwicklungsumgebung keine Option.

### 6.4.2. Emacs Haskell Plugin

Emacs ist ein sehr alter, früher konsolenbasierter Editor.
Hier gibt es eine sehr ausführliche Anleitung für die Installation des Haskell Plugins für Emacs:
https://github.com/serras/emacs-haskell-tutorial/blob/master/tutorial.md

Nach einer halben Stunde und einigen Problemen haben wir uns schnell gegen Emacs entschieden.
Für uns liegt der Fokus mehr auf der Programmierung mit Haskell und nicht in der Anwendung der IDE. Somit wollen wir nicht zu viel Zeit in dem Setup und das Eingewöhnen investieren.
Deshalb ist Emacs keine Option für uns.

### 6.4.3. Leksah

Leksah ist eine Entwicklungsumgebung, die vollständig in Haskell geschrieben wurde.
Hier ist die offizielle Seite: http://leksah.org/

Nach einigen versuchen war auch schnell klar, dass diese IDE auch nicht so komfortabel ist. Vieles, was man sich von Eclipse gewohnt ist, funktioniert bei weitem nicht so gut.
Deshalb haben wir uns gegen Leksah entschieden.

### 6.4.4. Sublime Haskell Plugin

Sublime ist ein universeller Editor mit vielen Plugins. Unter anderem das SublimeHaskell Plugin.
Sublime: http://www.sublimetext.com/
SublimeHaskell: https://github.com/SublimeHaskell/SublimeHaskell

In Sublime sind viele Tricks möglich beim Editieren von Dateien. Ausserdem bietet das Haskell Plugin viele Features: Auto-Build, Type Def, Syntax-Color,..

Wir haben uns für diese Entwicklungsumgebung entschieden.

## 6.5. Technologien

### 6.5.1. Yampa

Als Framework wurde Yampa verwendet, welches auf den Prinzipien von Functional Reactive Programming basieren. Verschiedene Haskell-Spiele basieren auf diesem Framework, was die Entscheidung erleichterte, dieses Framework zu verwenden. Yampa abstrahiert für den Benutzer die Zeit und die Events. Yampa ist ein mächtiges Framework mit vielen Funktionen.

### 6.5.2. SDL

SDL ist eine Opensource Library, die weit verbreitet ist. In der Haskell-Community gibt es einige Spiele die mit SDL umgesetzt wurden. Daher wurde zur Visualisierung SDL verwendet. Die Unterstützung von Bitmaps und Schriften kann im Projekt später von Nutzen sein.

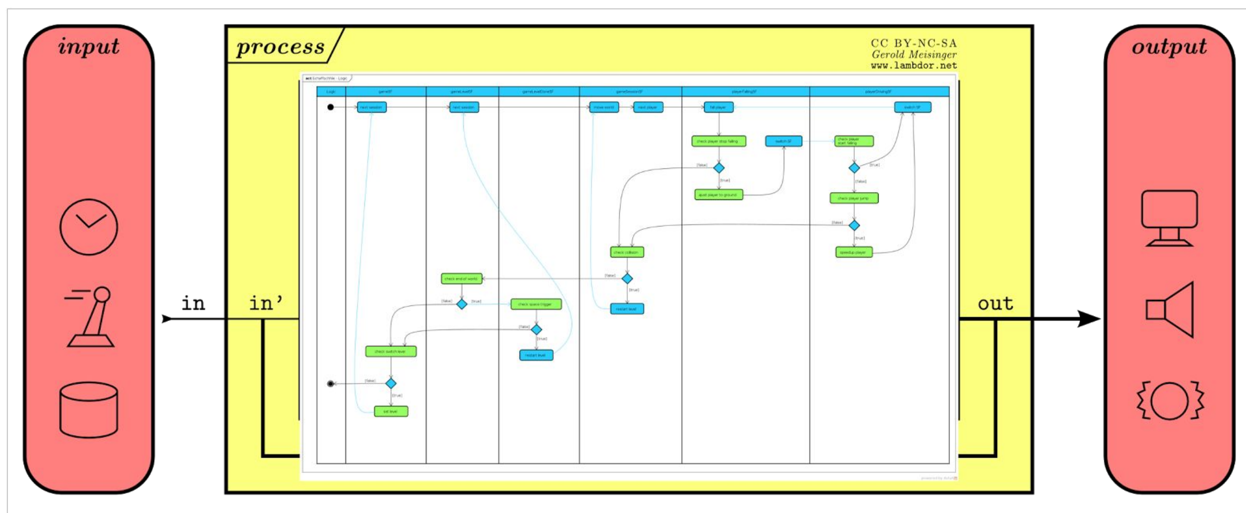
## 6.6. Architektur

Dieses Diagramm zeigt die grobe Struktur, die von dem Yampa Framework vorgegeben wird. Die Applikation ist in drei Teile aufgeteilt:

- Input: Hier werden Zeit, Key/Mouse-Inputs gehandelt. Yampa abstrahiert dabei die Zeit und arbeitet jeweils mit der Deltatime der Events.
- Process: Die Logik der Applikation wird hier implementiert.
- Output: Hier wird der Output gehandelt, Bildschirmausgaben, Tonausgaben.

Mit dieser Aufteilung ist die Applikation gekapselt, in den Input und Output - Teilen werden mit Monaden gearbeitet, während im Process-Teil nur SignalFunctions zum Zuge kommen. Somit wird sichergestellt, dass der Kern der Applikation immer 'pure' ist.

Dieser Ablauf wird in einem Loop durchgeführt, bis die Applikation gestoppt wird.



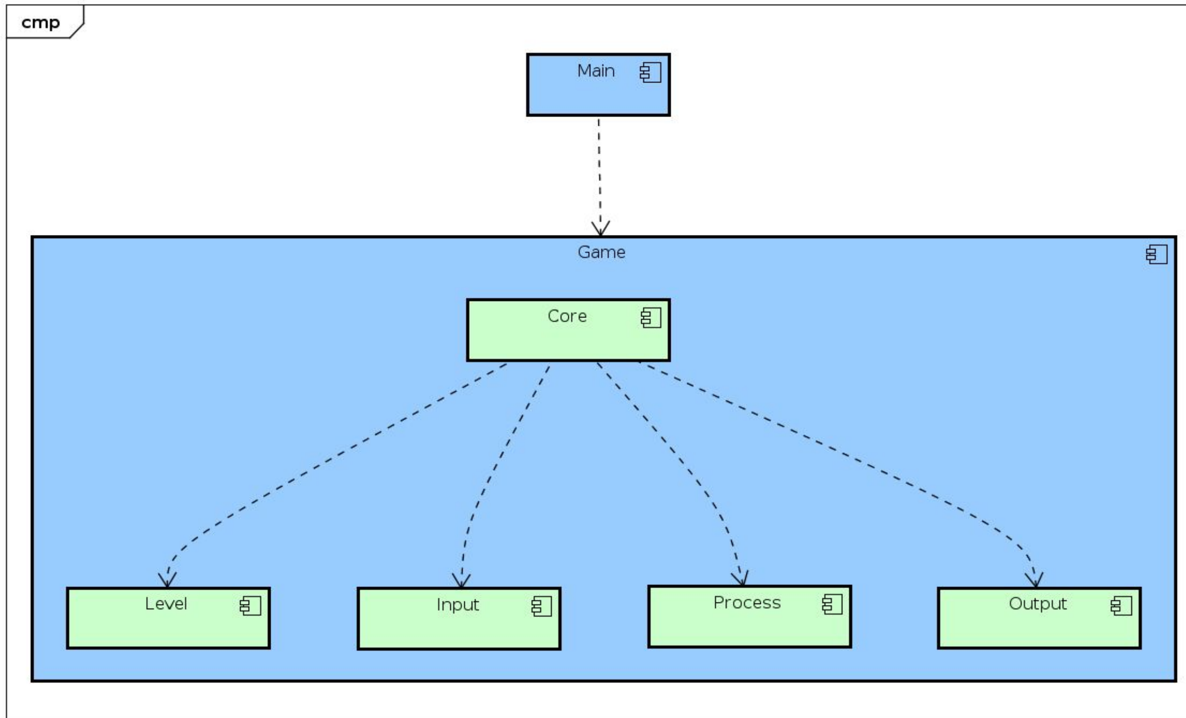

Quelle:
https://wiki.haskell.org/wikiupload/thumb/e/e1/Yampa_game_engine_architecture.png/768px-Yampa_game_engine_architecture.png

Der folgende Codeausschnitt ist ein wesentlicher Teil von Yampa. Es repräsentiert die vorherige Abbildung als Code.

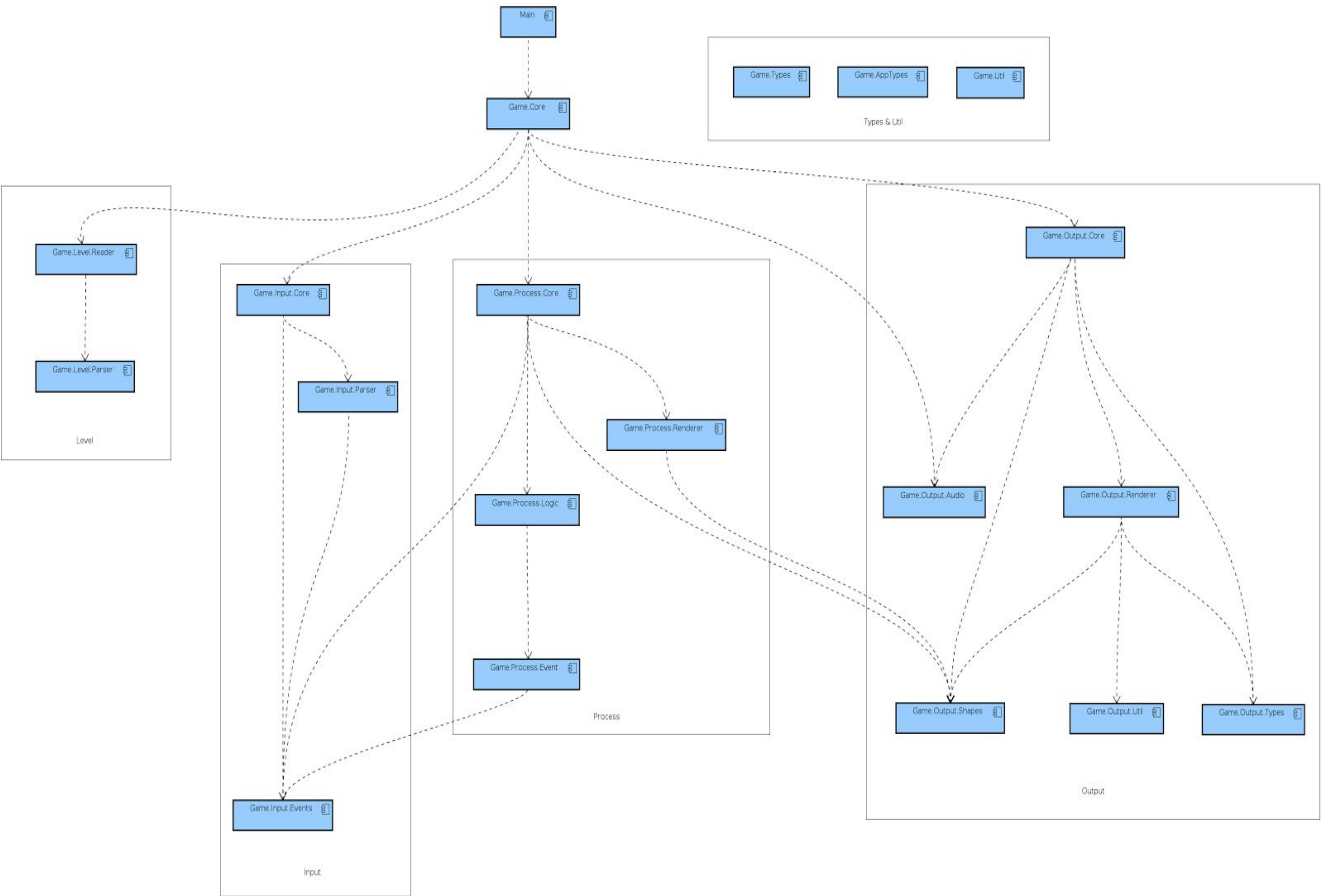
```
reactimate :: IO a                                -- init
           -> (Bool -> IO (DTime, Maybe a))  -- input/sense
           -> (Bool -> b -> IO Bool)         -- output/actuate
           -> SF a b                          -- process/signal function
           -> IO ()
```

### 6.6.1. Module

Die Applikation ist in vier Teile aufgeteilt:




- Input: In diesem Package werden die Inputs über SDL auf die lokalen Event's gemapped.
- Level:
  In diesem Package wird das aktuelle Level spaltenweise geparsed. Jedes Zeichen hat eine Bedeutung und wird zu einem GameObjekt gemapped. Die GameObjekte können folgende Eigenschaften haben: Kollidierbar, Befahrbar
  Mit diesen Gameobjekten wird die Landschaft aufgebaut.
- Output:
  Im Output-Package wird für die Bildschirmausgaben und Tonausgaben gesorgt. Aus den Gameobjekten werden SDL-Shapes gezeichnet, welche die Landschaft visualisieren. Im Renderer werden die Textausgaben vorbereitet.
- Process:
  Im Process-Package wird die Logik des Games gehandelt. Die Landschaft zieht an der Spielfigur vorbei, falls eine Spacetaste betätigt wird, wird die Spielfigur in die Luft springen. Ausserdem werden Kollisionen überprüft, bei einem Zahlentasten-Input wird das Level gewechselt.



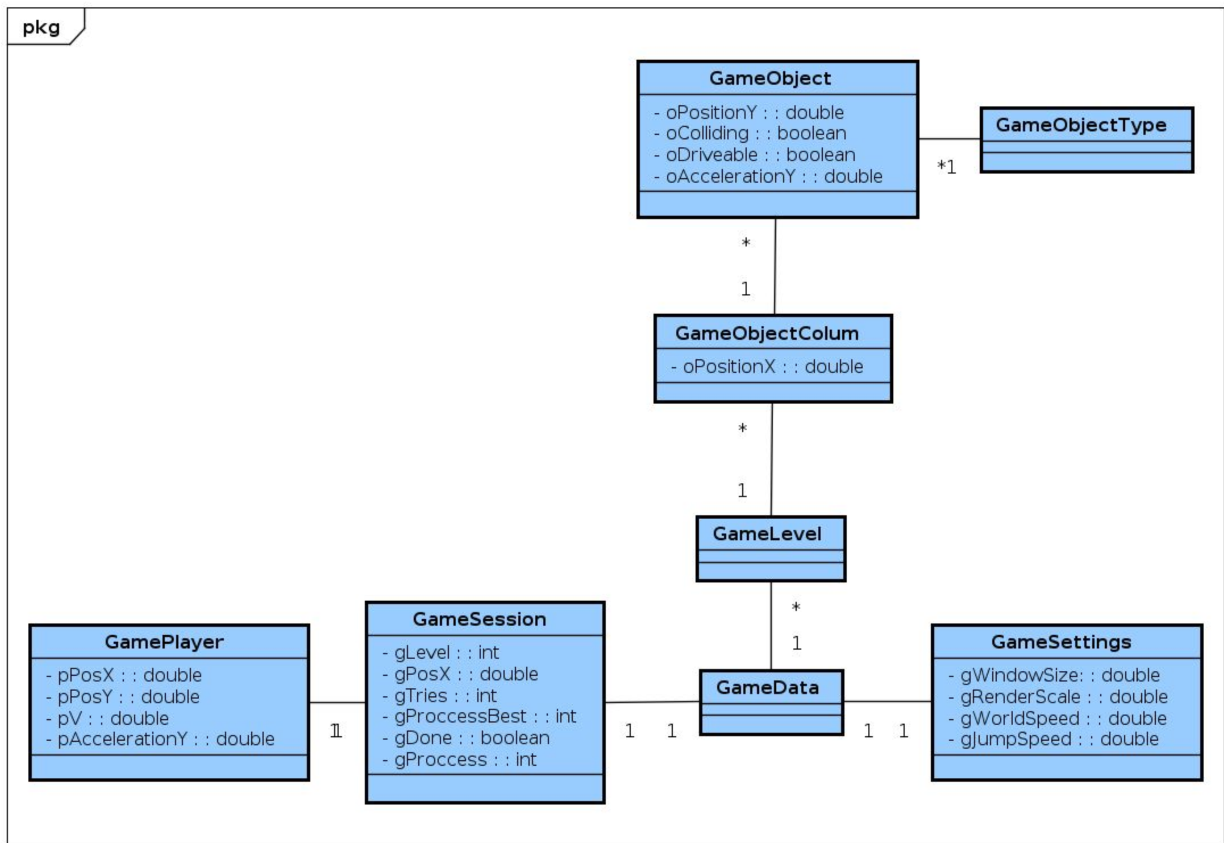
Main
Game.Core
Game.Types
Game.AppTypes
Game.Util
Types & Util
Game.Level.Reader
Game.Level.Parser
Level
Game.Input.Core
Game.Input.Parser
Game.Input.Events
Input
Game.Process.Core
Game.Process.Renderer
Game.Process.Logic
Game.Process.Event
Process
Game.Output.Core
Game.Output.Audio
Game.Output.Renderer
Game.Output.Shapes
Game.Output.Util
Game.Output.Types
Output

### 6.6.2. Detaillierte Modul Beschreibung

In den Modulen stehen folgende wichtige Funktionn zur Verfügung

- Input
  - input
    In dieser Funktion werden die KeyInput's über das SDL geholt.
  - parse
    In dieser Funktion werden die SDL-Events auf die internen Events gemapped.
  - getTime
    Diese Funktion spricht für sich.
- Level
  - read
    In diese Funktion wird das aktuelle Level als Parameter mitgegeben, danach wird das Level eingelesen.
  - parseLevel
    In dieser Funktion werden die GameObject erstellt, mit den Bildern gemappt, in die entsprechende Position abgespeichert(Zeile und Spalte). Der Returnwert ist ein Level.
- Process
  Dieser Teil wird unter dem Punkt Logik genauer erklärt.
- Output
  - init
    In dieser Funktion wird das Spielfenster erstellt, alle Bilder werden hineingeladen und Textausgaben werden ausgegeben.
  - output
    Diese Funktion aktualisiert jeweils die Bildschirmausgabe.
  - render
    Diese Funktion mappt die internen Objekte zu den SDL-Objekten.

### 6.6.3. Domain Model

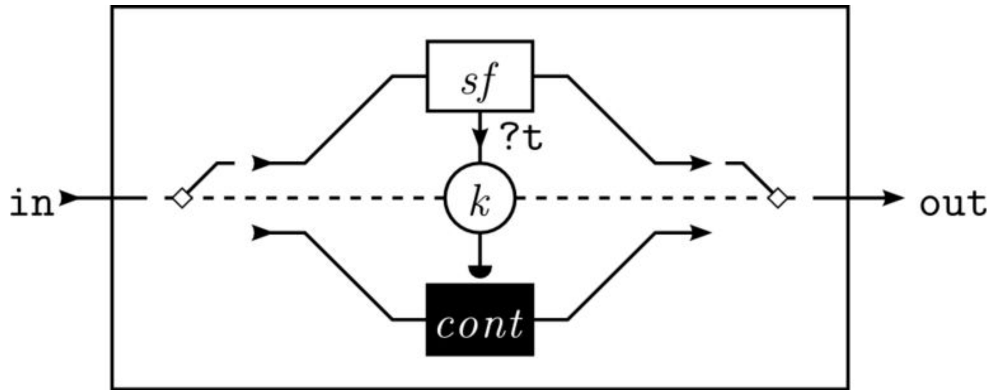



1. GamePlayer, dieses Objekt repräsentiert die Spielfigur:
    a. pPosX: X-Koordinate, wird nur zum initialisieren und Kollisionsprüfung verwendet
    b. pPosY: Y-Koordinate, wird beim Springen und der Kollisionsprüfung verwendet
    c. pV: Geschwindigkeit
    d. pAccelerationY: Beschleunigung
2. GameData, enthält je eine GameSession, GameSettings und mehrere GameLevel's
3. GameSession, dieses Objekt repräsentiert die aktuelle Session
    a. gLevel: Das aktuelle Level, das gespielt wird, in einer Zahl
    b. gPosX: Der Puffer, zwischen dem linken Rahmen und pPosX
    c. gTries: Anzahl Versuche die für dieses Level schon gemacht wurden
    d. gProccessBest: Dient zur prozentualer Anzeige des best erreichten Levels
    e. gDone: Zeigt, ob das Level fertig ist
    f. gProccess: Dient zur prozentualer Anzeige des aktuellen erreichten Level

4. GameSettings
    a. gWindowsSize: Fenstergrösse
    b. gRenderScale: Grösse der Objekte
    c. gWorldSpeed: Geschwindigkeit, mit der die Welt an der Spielfigur vorbei zieht
    d. gJumpSpeed: Geschwindigkeit, mit der die Spielfigur in die Luft springt.
5. GameLevel enthält mehrere GameObjectColumn
6. GameObjectColumn, enthält mehrere GameObject
    a. oPositionX: Zeigt, in welcher Spalte sich das GameObject sich befinden
7. GameObject
    a. oPositionY: Zeigt, in welcher Zeile sich das GameObject befindet
    b. oColliding: Beschreibt, ob das Objekt eine kollidierbar ist
    c. oDriveable: Beschreibt, ob auf diesem Objekt fahren möglich ist
    d. oAccelerationY: kann später verwendet werden, um die Level’s schwieriger zu gestalten.

## 6.7. Yampa Funktionen

### 6.7.1. Switch




Quelle: https://wiki.haskell.org/wikiupload/thumb/0/00/Yampa_switch.png/768px-Yampa_switch.png

Die obige Abbildung zeigt eine wichtige Funktion in Yampa. Mit diesem Switch kann während der Laufzeit des Programmes das Verhalten geändernt werden. In die SignalFunction kommt immer ein Input und danach kommt der Output heraus. Falls jetzt ein Event eintritt wird auf die neue SignalFunction geschaltet, mit den gleichen Input-/Outputtypen. Danach kann nicht mehr zurück geschaltet werden.

### 6.7.2. Edge

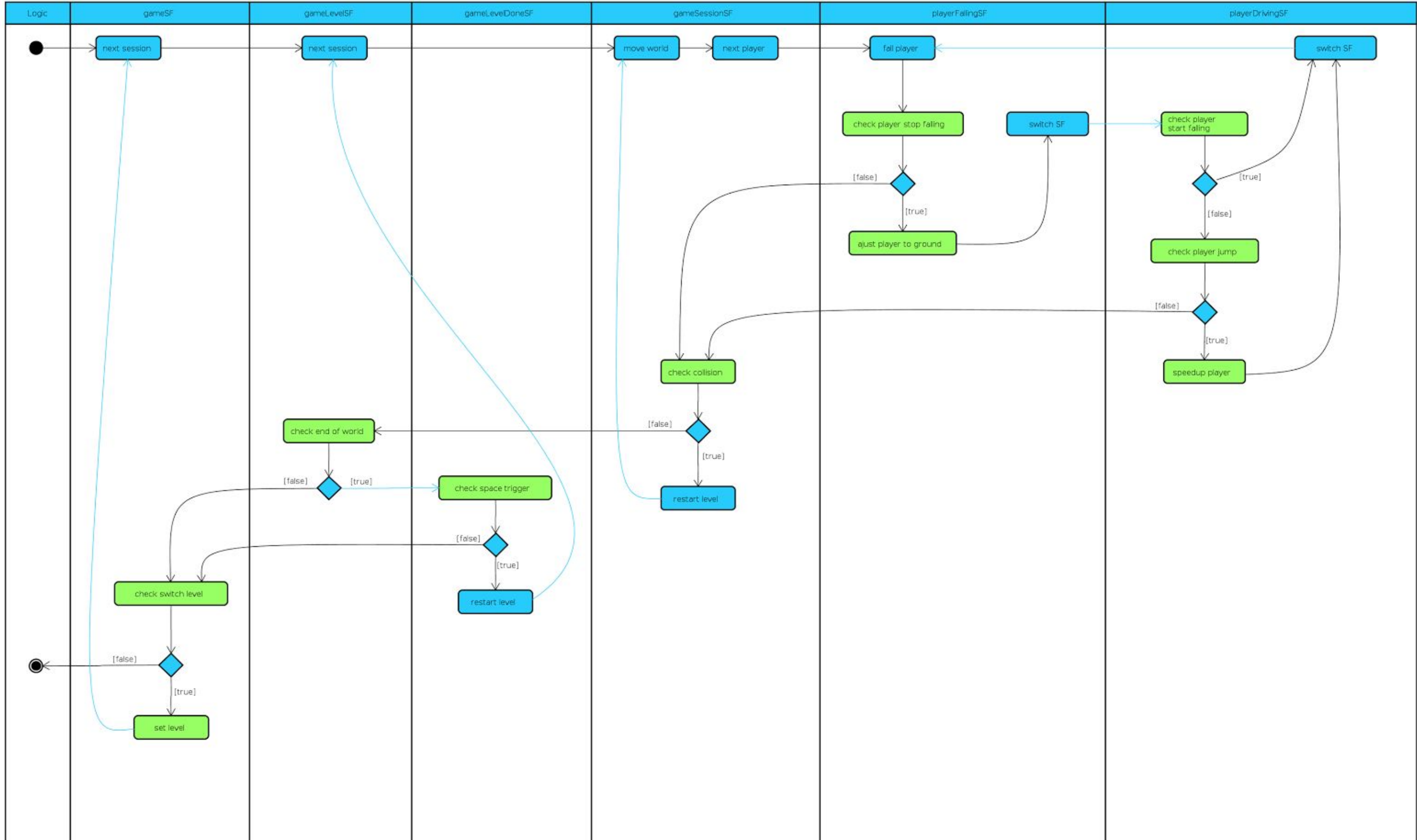
Definition von Yampas Edge Funktion.

```
edge :: SF Bool (Event ())
```

Diese Funktion ist ein Anstiegsflanken-Detektor, das heisst, wenn der Boolean Input von False auf True wechselt, wird einmalig ein Event geworfen. Das ist zum Beispiel wichtig, um das Level zu wechseln, da der Key Event beim drücken solange geworfen wird, bis der Benutzer den Key wieder loslässt. Das Level soll aber nur einmal gewechselt werden.

## 6.8. Game Logik

Die Game Logik wird innerhalb der Isolierten Process Function von Reactimate definiert. Bei der Grafik wird zwischen Signal Functions (Blaue Boxen) und Pure Functions (Grüne Boxen) unterschieden. Der Normale ablauf wird mit schwarzen Pfeilen dargestellt und der Switch einer Signal Function wird mit blauen Pfeilen dargestellt.



act SchaffschNe - Logic
Logic
gameSF
gameLevelSF
gameLevelDoneSF
gameSessionSF
playerFallingSF
playerDrivingSF
next session
next session
move world
next player
fall player
switch SF
check player stop falling
switch SF
check player
start falling
[false]
[true]
[true]
[false]
ajust player to ground
check player jump
[false]
[true]
check collision
speedup player
[false]
check end of world
[true]
[false]
[true]
check space trigger
restart level
[false]
[true]
check switch level
restart level
[false]
[true]
set level
powered by Astah

### 6.8.1. Beschreibung

#### 6.8.1.1. GameSF

Dieses ist die Hauptfunktion für die Gamelogik. Sie lädt die nächste Session und kann das Level ändern.

#### 6.8.1.2. GameLevelSF

Diese Funktion überprüft, ob das Ende erreicht wurde. Falls es erreicht wurde, schaltet sie auf die GameLevelDoneSF, um den Erfolgsscreen anzuzeigen.

#### 6.8.1.3. GameLevelDoneSF

Hier wird nur der Erfolgsscreen angezeigt und gewartet bis Space gedückt wird, um das Level neu zu starten.

#### 6.8.1.4. GameSessionSF

In dieser SF wird die Welt bewegt und auf Kollisionen überprüft. Wenn eine Kollision eintritt, wird diese SF neu mit den initialen Werten geladen. Dabei wird die Anzahl der Versuche erhöht.

#### 6.8.1.5. FallingPlayerSF

In der FallingPlayerSF wird der Player beschleunigt. Wenn der Player wieder auf dem Boden ankommt, schaltet der switch auf die DrivingPlayerSF um.

#### 6.8.1.6. DrivingPlayerSF

DrivingPlayerSF hält die Position des Players Konstant und schaltet auf die FallingPlayerSF um, sobald kein Boden mehr unterhalb des Players ist.

## 6.9. Testing

In der Elaborations-Phase wurden die drei Testing-Framework’s Hspec, HUnit, QuickCheck angeschaut. Die Auswahl fiel auf Hspec, jedoch wurde im Verlauf der Elaboration auch klar, dass das Testing nicht im Vordergrund steht, sondern das Erlernen von Haskell. Aus diesem Grund wurde kein Framework zum Testing verwendet. Der Fokus wurde viel mehr auf die Implementierung gesetzt.

Die Applikation wurde viel mehr von Hand getestet.

# 7. Ergebnisse

Das Endprodukt der Semesterarbeit ist ein Spiel, welches wir mit Haskell implementiert haben. Dazu haben wir eine Dokumentation geschrieben, die Einsteigern helfen könnte. Das wichtigste aus der Semesterarbeit ist, dass wir vieles in der funktionalen Programmierwelt erlernen konnten.

# 8. Schlussfolgerung

## 8.1. Ausblick

Das Spiel hat mehrere Möglichkeiten zum Ausbau.

- Leben
  Zur Zeit fängt das Spiel neu an, wenn der Spieler in ein Hindernis fährt. In Zukunft könnte man das so ausbauen, dass der Spieler zwei Leben hat. Somit würde das Spiel erst neu anfangen, wenn diese zwei Leben durch eine Kollision verloren gehen.
- Hindernisse
  Weitere Hindernisse können einfach zum Spiel hinzugefügt werden. Man muss nur das entsprechende Zeichen in das Parse-File hinzufügen und das entsprechende Bild hinzufügen.

# 9. Installationsanleitung

## 9.1. Ubuntu Prerequisites

```
# Multiprecision arithmetic library developers tools, zlib
sudo apt-get install libgmp-dev zlib1g-dev -y
```

## 9.2. Clean-Up

```
# remove old
rm -rf ~/.cabal ~/.ghc
```

## 9.3. Set Path

Exports in ~/.profile definieren:

```
# add this lines to ~/.profile
export GHC_HOME=/opt/haskell/ghc
export CABAL_HOME=~/.cabal
export STACK_HOME=/opt/haskell/stack
export PATH=$GHC_HOME/bin:$CABAL_HOME/bin:$STACK_HOME:$PATH
```

Danach Neustart oder:

```
source ~/.profile
```

## 9.4. GHC

Homepage: [https://www.haskell.org/ghc](https://www.haskell.org/ghc)

Source download:

http://downloads.haskell.org/%7Eghc/7.10.2/ghc-7.10.2-x86_64-unknown-linux-deb7.tar.bz2

```
tar xfj ghc-7.10.2-x86_64-unknown-linux-deb7.tar.bz2
cd ghc-7.10.2

# install to
sudo mkdir -p /opt/haskell/ghc
# parallel 4 jobs
sudo ./configure --prefix=/opt/haskell/ghc && sudo make -j 4 install
```

## 9.5. Cabal (Package Manager für Haskell)

Homepage: https://www.haskell.org/cabal

### 9.5.1. cabal

Source download: https://www.haskell.org/cabal/release/cabal-1.22.4.0/Cabal-1.22.4.0.tar.gz

```
tar xzf Cabal-1.22.4.0.tar.gz
cd Cabal-1.22.4.0

# build
ghc --make Setup.hs && ./Setup configure --user && ./Setup build && ./Setup install
```

### 9.5.2. cabal-install

Source download:
https://www.haskell.org/cabal/release/cabal-install-1.22.6.0/cabal-install-1.22.6.0.tar.gz

```
tar xzf cabal-install-1.22.6.0.tar.gz
cd cabal-install-1.22.6.0

# install
./bootstrap.sh

cabal update
```

## 9.6. Stack (neuer Package Manager und Build Tool)

Homepage: https://www.stackage.org/
Bin download:
https://github.com/commercialhaskell/stack/releases/download/v0.1.5.0/stack-0.1.5.0-x86_64-linux.tar.gz

```
tar xfz stack-0.1.5.0-x86_64-linux.tar.gz
sudo mv stack-0.1.5.0-x86_64-linux /opt/haskell/stack

sudo apt-get install libtinfo-dev
```

## 9.7. SDL (OpenGL library)

### 9.7.1. SDL2

Homepage: https://www.libsdl.org

Source download: https://www.libsdl.org/release/SDL2-2.0.3.tar.gz

```
tar xfz SDL2-2.0.3.tar.gz
cd SDL2-2.0.3

# install
./configure && make -j 4 && sudo make install
```

### 9.7.2. SDL2-TTF (Font)

Homepage: https://www.libsdl.org/projects/SDL_ttf/

Source download: https://www.libsdl.org/projects/SDL_ttf/release/SDL2_ttf-2.0.12.tar.gz

```
sudo apt-get installl libsdl2-ttf-dev
```

## 9.8. SublimeText Haskell

Installation:

1. Get Sublime Text 2: http://www.sublimetext.com/
2. Install the Sublime Package Control package: http://wbond.net/sublime_packages/package_control/installation
3. Use Package Control to install this package (SublimeHaskell)

```
cabal install aeson happy haskell-src-exts haddock
cabal install ghc-mod stylish-haskell hdevtools
```

## 9.9. Entwicklung

Für die Entwicklung eigenen sich folgende Commands:

- cabal repl (lädt ghci und alle Module die zu diesem Projekt gehören)
- cabal run (lässt das Spiel direkt laufen)

# 10. Glossar

| | |
|---|---|
| SF | Signal Function |
| SDL | Simple DirectMedia Layer |
| TTF | True Type Font |
| FRP | Functional Reactive Programming |
| GHC | Glasgow Haskell Compiler |
| GHCI | Glasgow Haskell Compiler Interactive Environment |
| Cabal | Haskell Package Manager |
| Stack | Haskell Package Manager |
| IDE | Entwicklungsumgebung |
| Yampa | Funktional Reactive Library |

# 11. Literaturverzeichnis


1. https://www.haskell.org/, 18.12.2015
2. http://learnyouahaskell.com/, 18.12.2015
3. https://wiki.haskell.org/Yampa, 18.12.2015
4. https://wiki.libsdl.org/, 18.12.2015
5. https://www.libsdl.org/projects/SDL_ttf/docs/SDL_ttf_frame.html, 18.12.2015
6. https://hackage.haskell.org/package/sdl2, 18.12.2015
7. https://hackage.haskell.org/package/SDL2-ttf, 18.12.2015

# 12. Eigentständigkeitserklärung

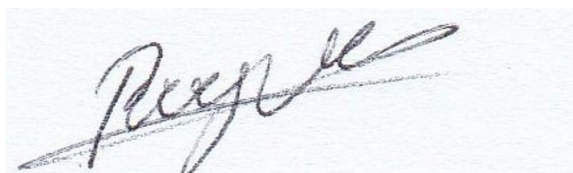
Ich erkläre hiermit,

- dass ich die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt habe, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde,
- dass ich sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben habe.
- dass ich keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt habe.

Ort, Datum: Rapperswil, 18.12.2015

Name, Unterschrift:

Florian Merz
18.12.2015, Rapperswil



Royce Manavalan
18.12.2015, Rapperswil

# 13. Nutzungsrechte

## 13.1. Gegenstand der Vereinbarung

Mit dieser Vereinbarung werden die Rechte über die Verwendung und die Weiterentwicklung der Ergebnisse der Studienarbeit Functional Reactive Programming von Florian Merz und Royce Manavalan unter der Betreuung von Prof. Dr. Josef Joller geregelt.

## 13.2. Urheberrecht

Die Urheberrechte stehen der Studentin / dem Student zu.

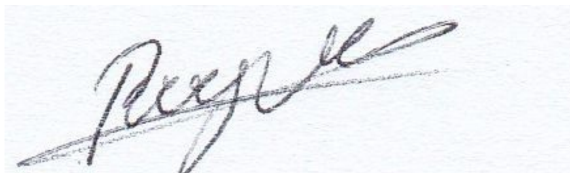
## 13.3. Verwendung

Die Ergebnisse der Arbeit dürfen sowohl von der Studentin / dem Student, sowie von der HSR nach Abschluss der Arbeit verwendet und weiterentwickelt werden.

Beilage/n:

- Code

Florian Merz, Student
18.12.2015, Rapperswil



Royce Manavalan, Student
18.12.2015, Rapperswil

Prof. Dr. Josef Joller, Betreuer
18.12.2015, Rapperswil