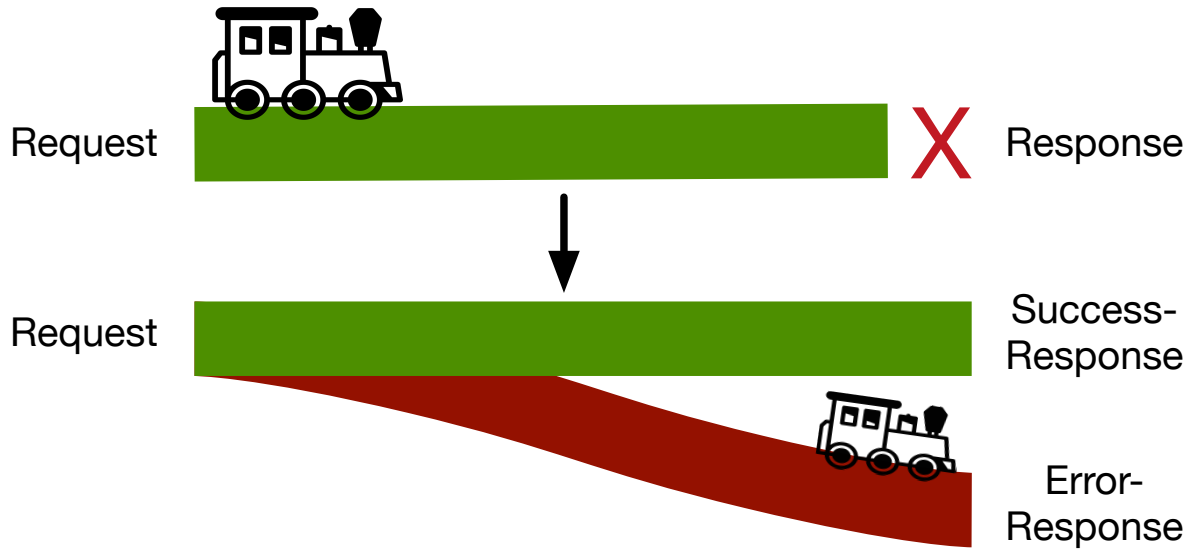
RAILWAY-ORIENTED-PROGRAMMING IN CLOJURE

Funktionales Error-Handling

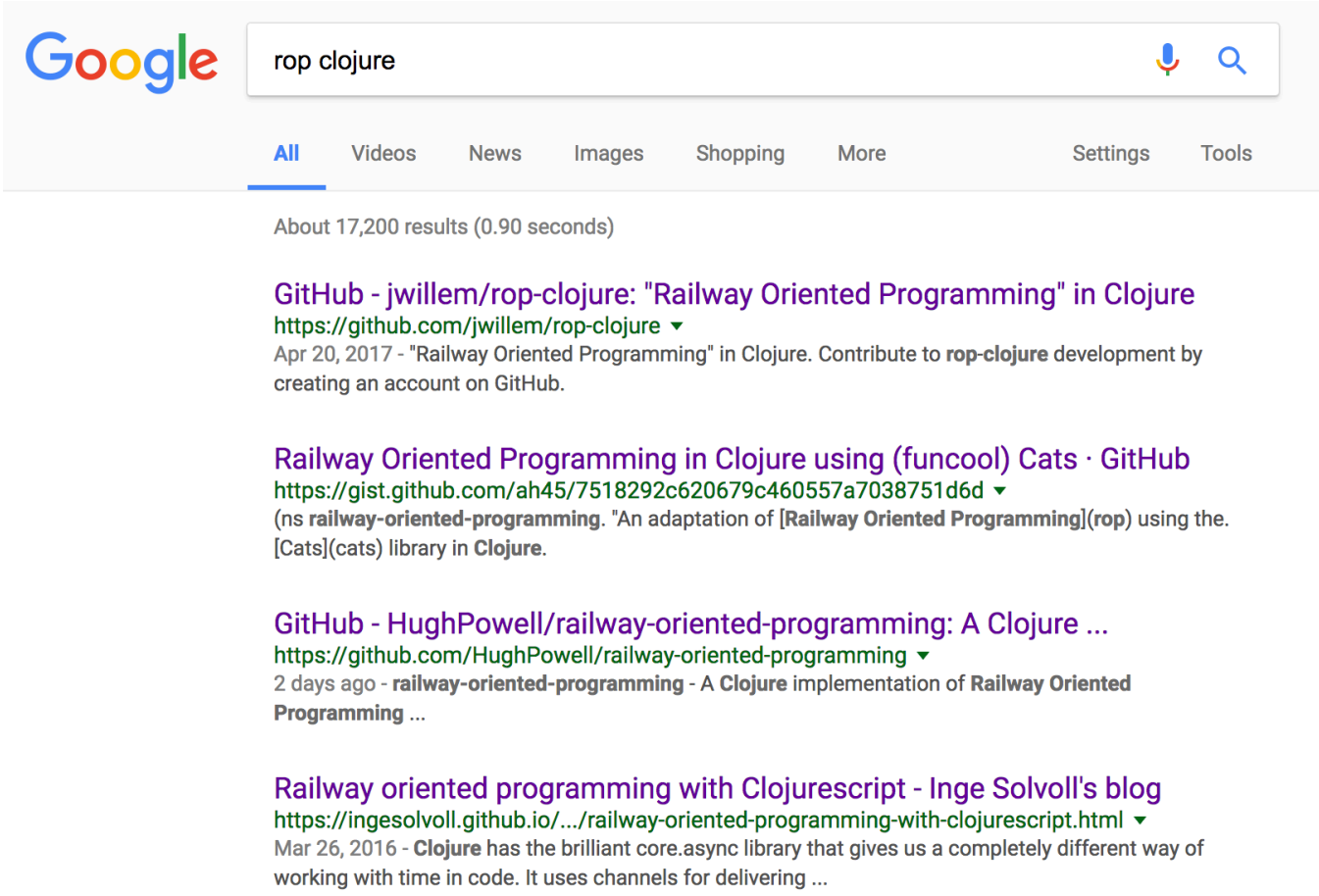
EFP, SS2017



Jan-Philipp Willem

Fakultät für Informatik
Hochschule Mannheim

PROJECT-REPOSITORY



https://github.com/jwillem/rop-clojure

GLIEDERUNG

1. Who?
2. „Happy-Path“
3. Error by design
4. „Railway-Oriented-Programming“
5. Ausblick

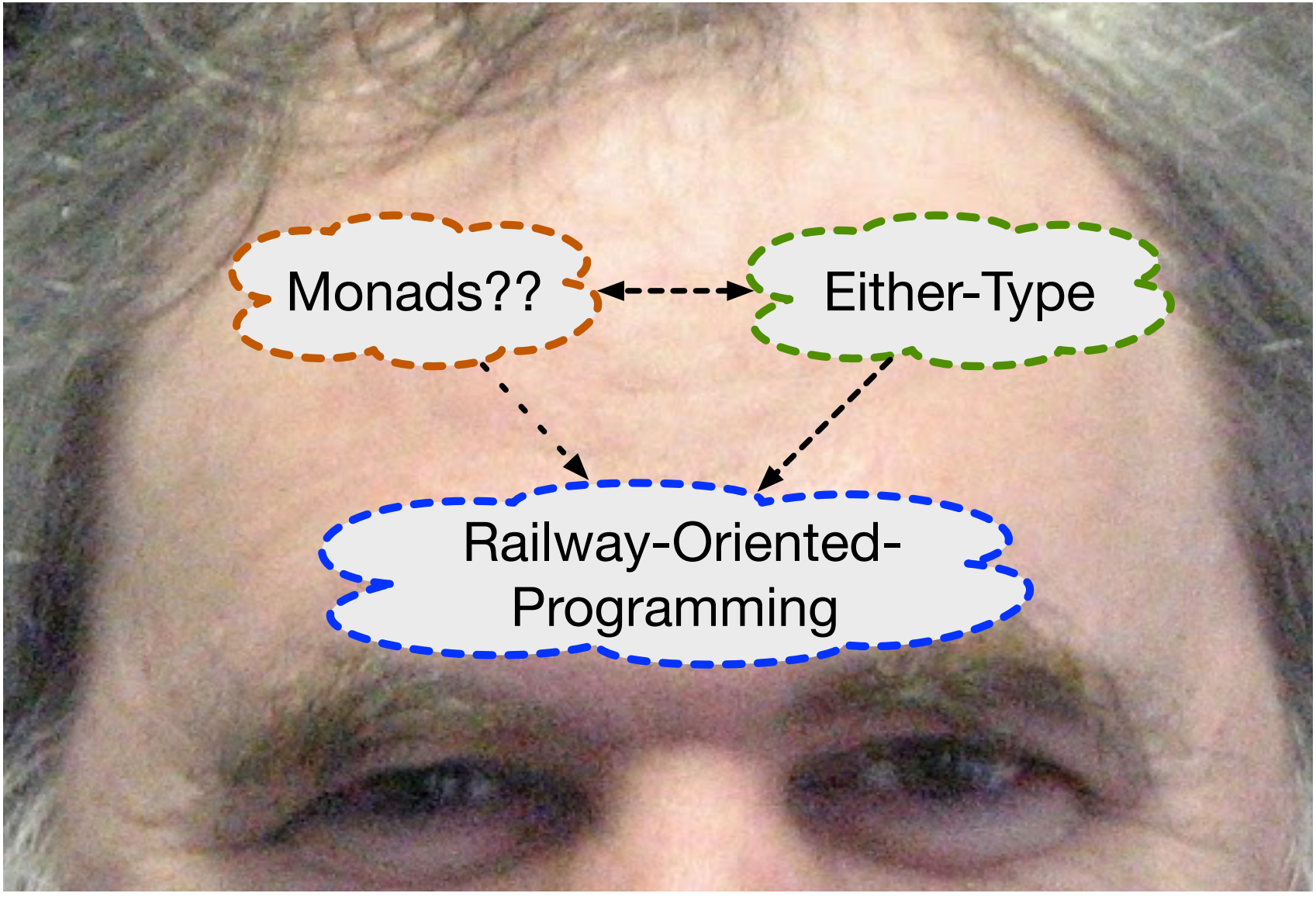
SCOTT WLASCHIN

https://fsharpforfunandprofit.com/

„WELL, I SUPPOSE YOU DON'T NEED TO KNOW ABOUT MONADS. YOU ONLY NEED TO USE 'MAYBE' -- MAYBE WHAT? -- 'MAYBE' THE MONAD. -- MAYBE THE MONAD WHAT? -- NOW I THINK ABOUT IT. 'EITHER' MIGHT BE BETTER... -- EITHER WHAT?“

(SCOTT WLASCHIN, NDC CONFERENCE 2014) [1]

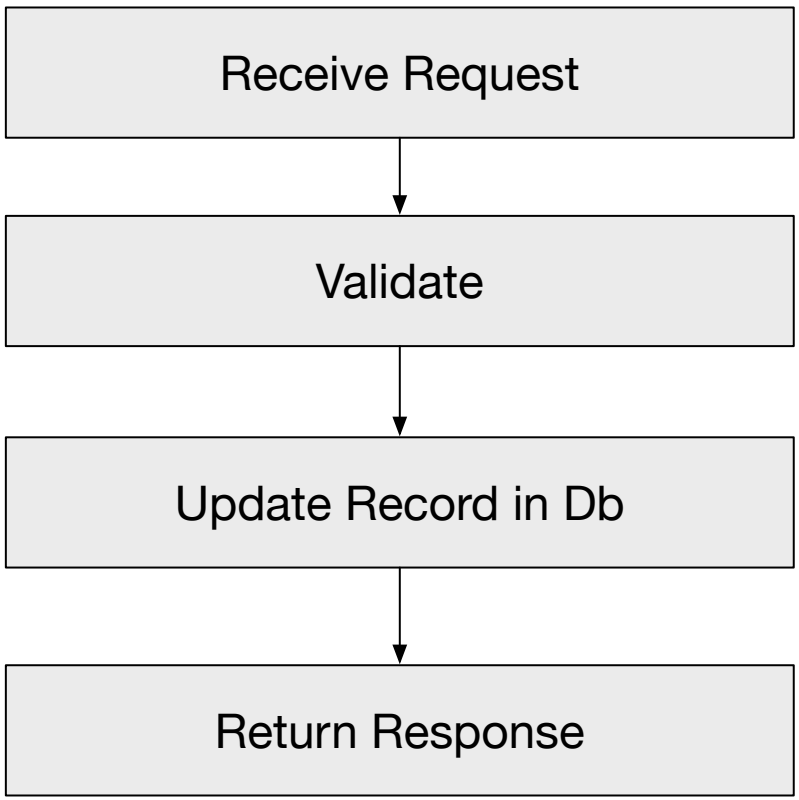
CONCEPT



[1] https://vimeo.com/113707214

„HAPPY-PATH“

USECASE: UPDATE FIRSTNAME + EMAIL OF USER (1/2)




```
Request:
{
”type”: “user”,
“id”: “k48mdsjk”,
”firstname”: ”Max”,
“email”:
    “max@muster.tld”
}
Response:
{
”type”: “user”,
“id”: “k48mdsjk”,
“key”: value
..
}
```

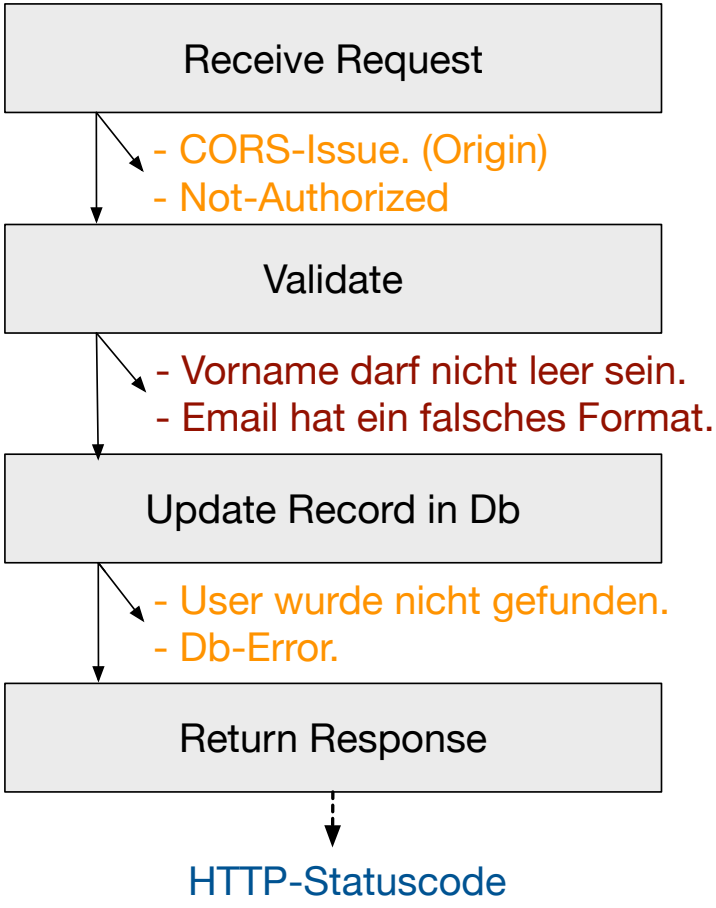
USECASE HAPPY-PATH

```
(defn update-user
“updates a given user with
 the given keys”
[request]
(-> request
    transform-from-request
    validate-data
    update-in-db
    transform-to-response))
```

ERROR BY DESIGN

USECASE: UPDATE FIRSTNAME + EMAIL OF USER (2/2)

Und sinnvolle Fehler zurückgeben.




```
Request: {
”type”: “user”,
”firstname”: “ “ ,
“email”:
   “max@@@muster.tld”
}
Response: {
”result”: {
  “success”: {
    ”type”: “user”,
    “id”: “k48mdsjk”,
    “key”: value
    ..
  }
}
Header: 200 OK | ..
```

USECASE MIT ERROR-HANDLING

```
(defn update-user'
"updates a given user with the
given keys. follows rop-convention"
[request]
(-> request
    transform-from-request
    (>>= validate-data
         update-in-db
         transform-to-response))
```

„RAILWAY-ORIENTED-PROGRAMMING“

GRUNDLEGENDE SCHIENENTEILE

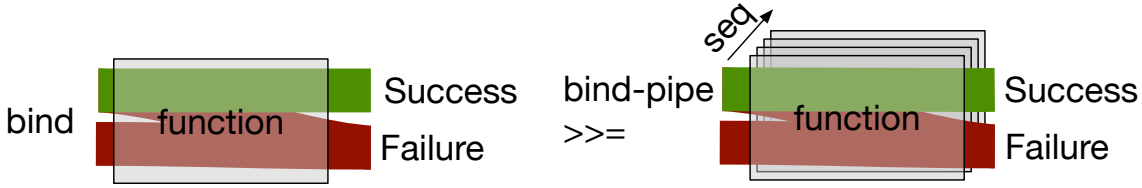
→ succeed-fail

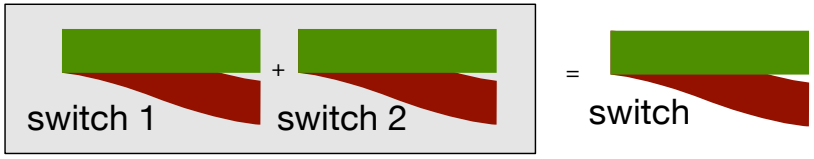
→ switch-function

→ bind

→ pipe-bind

→ switch-compose





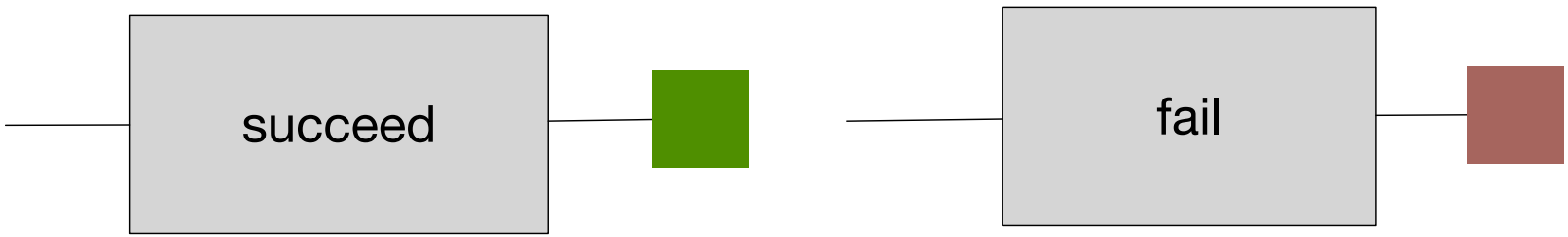
→ core, core.match

→ clojure seq-abstraction

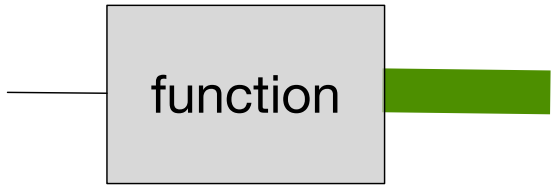
→ rest-parameters, map, apply

SUCCEED & FAIL

```
(succeed 'Hello Clojure!')
(fail 'Oh no!')
```



SWITCH-FUNCTION



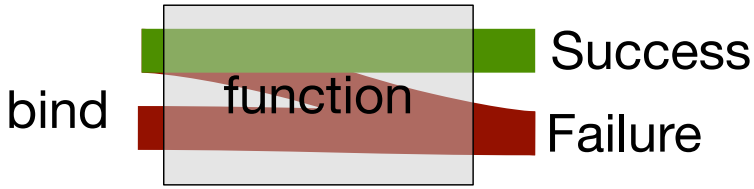
switch

```
((switch +) 2 2)
```

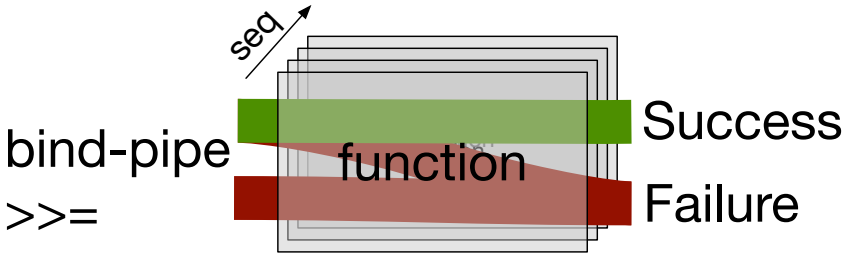
switch-function

```
((cond
    (> 2 %) (succeed %)
    :else (fail
            (str "Oh,no! " %)))
```

BIND



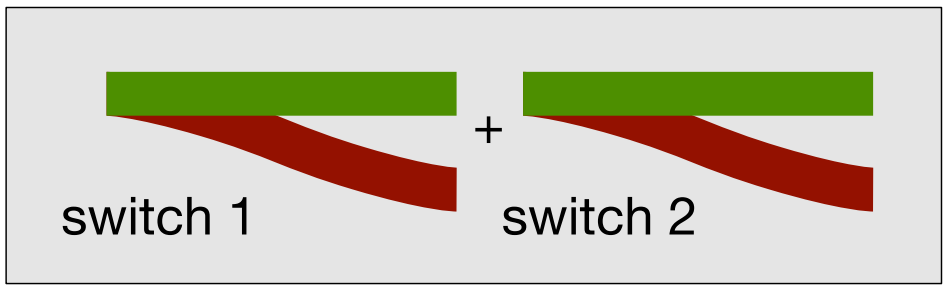

```
((bind #(succeed %))
 (succeed "42"))
```



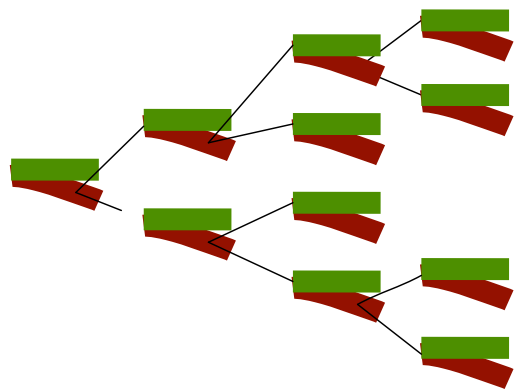

```
((>>= #(succeed %)
      #(fail
         (str "Fehler: " %)))
 (succeed "42"))
```

COMPOSE

```
((>=> ((switch +) 2 2)
      ((switch *) 2))
 (succeed "42"))
```



switch-compose
>=>



AUSBLICK

WEITERE SCHIENENTEILE

→ try/catch
→ log
→ tee
→ map-parallel
→ ..beliebig als Basis nutzbar

FAZIT

→ kann am Anfang im Vergleich sehr kompliziert sein

→ bei konsistenter Anwendung wirklich hilfreich

→ Data-Flow mit Step pro Usecase wird angestrebt

→ leicht erweiter- und wartbar

FIN

Danke für die Aufmerksamkeit. – Fragen?

REFERENCES

- → Category Theory and Algebraic abstractions in Clojure
  `https://github.com/funcool/cats`
- → Gut gelößtes Sprachkonzept mit Maybe & Either in Elm
  `https://guide.elm-lang.org/error_handling`
- → Match: wichtig überall
  https://github.com/clojure/core.match/wiki/Basic-usage