

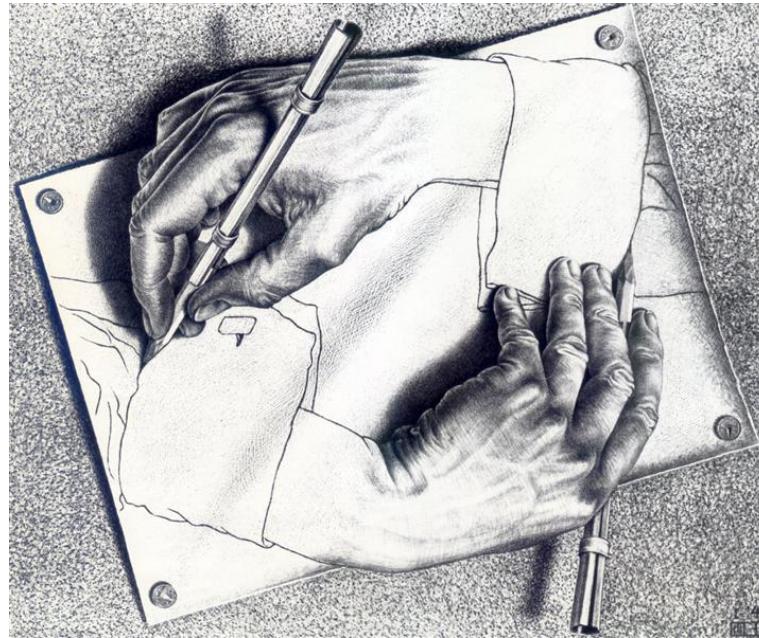
# Language Oriented Programming

Kaan Sahin, Active Group GmbH

Created: 2023-05-30 Tue 13:36

*@active group*

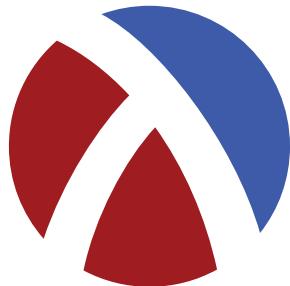
# Language Oriented Programming



*If you give someone Fortran, he has Fortran. If you give someone Lisp, he has any language he pleases.* —Guy L. Steele

@active group

# Racket – Übersicht



- Lisp-Dialekt, dynamisch getypt
- ermutigt funktionale Programmierung
- besonders für Lehre geeignet
  - funktionale Konzepte
  - Sprache anpassbar

# Racket – DrRacket

The screenshot shows the DrRacket IDE interface with the file "beispiel-drracket.rkt" open. The code editor contains the following Racket code:

```
1 #lang deinprogramm/sdp/beginner
2
3 (check-expect (summe 4 5) 9)
4 (define summe
5   (lambda (x y) Ein gebundenes Vorkommen
6     (- x y)))
7
8
```

A tooltip "Ein gebundenes Vorkommen" appears over the variable "x" in the lambda expression. The status bar at the bottom indicates "Sprache aus Quelltext ermitteln ▾" and "9:2 590.83 MB".

**Check-Fehler:**

Der tatsächliche Wert **-1** ist nicht der erwartete Wert **9**.  
in beispiel-drracket.rkt, Zeile 3, Spalte 0

*@active group*

# Racket – (Lisp-)Syntax

- Zwei Arten von Ausdrücken:
  - Atomare Ausdrücke: 17, "Hallo", sym, #f
  - Forms: (or false (> 2 1))

*@active group*

# Racket – Syntax

- Zwei Arten von Ausdrücken:
  - Atomare Ausdrücke: 17, "Hallo", sym, #f
  - Forms: (or false (> 2 1))

zudem noch:

- eckige Klammern: [1 2], u. a. in cond

*@active group*

# Racket – Syntax

```
3, "String", sym, true ; ; Atomare Werte  
(* (+ 4 3) 2) ; ; Integer, 14  
(list 1 2 (+ 1 2) "Hi") ; ; Liste, (1 2 3 "Hi")  
(if (> 2 3) "It is true" 100) ; ; Integer, 100  
(cond  
  [(> 3 4) "nicht das Ergebnis"]  
  [(= 3 3) "das Ergebnis"]  
  [(> 3 2) "nicht das Ergebnis"]) ; ; String, "das Ergebnis"
```

*@active group*

# Racket – Syntax, Definitionen

```
(define pi 3.14159)

(define (umfang radius)
  (* pi radius))

(define (factorial n)
  (if (= n 1)
      1
      (* n (factorial (dec n)))))
```

*@active group*

# Motivation – Makros

*@active group*

# Motivation – Makros

## List Comprehensions

Python

```
[x * x for x in range(20) if x % 2 == 0]
```

Haskell

```
[x * x | x <- [1, 2, 3, 4], x > 2]
```

*@active group*

# Motivation – Makros

## Pattern Matching

### Elixir

```
def fun(["Hi", name]), do: "What's up, " <> name <> "?"
def fun(_), do: "something else"

case {1, 2, 3} do
  {1, 2, 4} -> "this won't match"
  {1, 2, x} -> x
  _              -> "if nothing else matches"
end
```

*@active group*

# Motivation – Makros

Weitere Schreiberleichterung:

- Getter-/Setter-Funktionen
- Wiederkehrende Definitionen / Umständlichkeiten

Spracherweiterung via Libraries:

- Concurrency-System
- DSLs
- Statisches Typsystem
- ...

# Racket – Evaluation

```
repl> (define (foo x y) (+ x y))
```

*@active group*

# Racket – Evaluation

```
repl> (define (foo x y) (+ x y))  
repl>
```

*@active group*

# Racket – Evaluation

```
repl> (define (foo x y) (+ x y))  
repl> (foo (+ 2 1) (+ 2 2))
```

*@active group*

# Racket – Evaluation

```
repl> (define (foo x y) (+ x y))  
repl> (foo (+ 2 1) (+ 2 2))  
7
```

*@active group*

# Racket – Evaluationsschritte

```
(foo (+ 2 1) (+ 2 2))
```

*@active group*

# Racket – Evaluationsschritte

```
(foo (+ 2 1) (+ 2 2))
```

```
↝ (foo 3 (+ 2 2))
```

*@active group*

# Racket – Evaluationsschritte

```
(foo (+ 2 1) (+ 2 2))
```

```
↝ (foo 3 (+ 2 2))
```

```
↝ (foo 3 4)
```

*@active group*

# Racket – Evaluationsschritte

(foo (+ 2 1) (+ 2 2))

≈ (foo 3 (+ 2 2))

≈ (foo 3 4)

≈ (+ 3 4)

*@active group*

# Racket – Evaluationsschritte

(foo (+ 2 1) (+ 2 2))

≈ (foo 3 (+ 2 2))

≈ (foo 3 4)

≈ (+ 3 4)

≈ 7

*@active group*

# Racket – REPL

R EAD

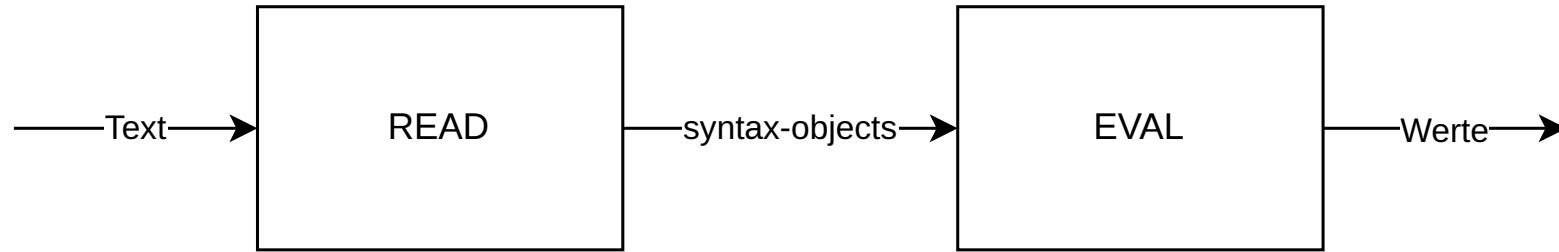
E VAL

P RINT

L OOP

*@active group*

# Kompilation – Read und Eval



Der Racket-Reader liest Text und gibt Datenstrukturen (syntax objects) zurück.

Der Evaluator nimmt Datenstrukturen (syntax objects) und evaluiert sie zu Werten.

*@active group*

# Makros – syntax objects

Ein syntax object erzeugt man mit #` :

```
#` (1 2 (+ 1 2) 4)
```

*@active group*

# Makros – syntax objects

Ein syntax object erzeugt man mit #` :

```
#` (1 2 (+ 1 2) 4)
~ #<syntax:intro.rkt:28:2 (1 2 (+ 1 2) 4)>
```

*@active group*

# Makros – syntax objects

Ein syntax object erzeugt man mit #` :

```
#`(1 2 (+ 1 2) 4)
~ #<syntax:intro.rkt:28:2 (1 2 (+ 1 2) 4)>

(define variable 5)
#`(1 2 variable 4)
~ #<syntax:intro.rkt:28:2 (1 2 variable 4)>
```

*@active group*

# Makros – syntax objects

Einzelnen Code innerhalb eines syntax objects wertet man mit # , aus:

```
#` (1 2 #, (+ 1 2) 4)
```

*@active group*

# Makros – syntax objects

Einzelnen Code innerhalb eines syntax objects wertet man mit # , aus:

```
#` (1 2 #, (+ 1 2) 4)  
~ #<syntax:intro.rkt:28:2 (1 2 3 4)>
```

*@active group*

# Makros – syntax objects

Einzelnen Code innerhalb eines syntax objects wertet man mit # , aus:

```
#` (1 2 #, (+ 1 2) 4)
~ #<syntax:intro.rkt:28:2 (1 2 3 4)>
```

```
(define variable 5)
#` (1 2 #,variable 4)
~ #<syntax:intro.rkt:28:2 (1 2 5 4)>
```

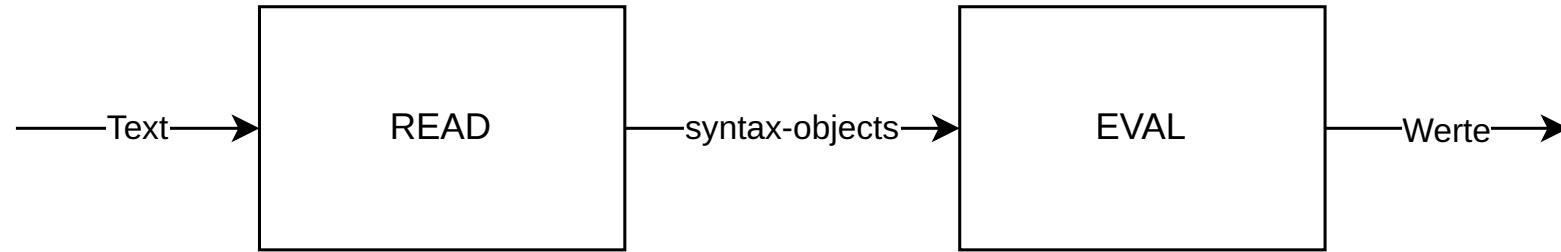
*@active group*

# Makros – syntax objects

Live-Coding

*@active group*

# Kompilation – Read und Eval



Der Racket-Reader liest Text und gibt Datenstrukturen (syntax objects) zurück.

Der Evaluator nimmt Datenstrukturen (syntax objects) und evaluiert sie zu Werten.

*@active group*

# Kompilation – Read, Eval und Makroexpansion!



*@active group*

# Kompilation – Read, Eval und Makroexpansion!



In der Makroexpansion werden Makroaufrufe getätigt.

*@active group*

# Kompilation – Read, Eval und Makroexpansion!



Makros nehmen Datenstrukturen (syntax objects) entgegen und geben Datenstrukturen (syntax objects) zurück.

*@active group*

# Makros

Makros nehmen Datenstrukturen (syntax objects) entgegen und **geben Datenstrukturen (syntax objects) zurück.**

*@active group*

# Makros – Infix

Statt

( + 2 1)

wollen wir

(2 + 1)

*@active group*

# Makros – Infix

```
(define-syntax (infix form)
  ...)
```

*@active group*

# Makros – Infix

```
(define-syntax (infix form)
  (syntax-parse form
    [ (infix ...)
      ... ])))
```

*@active group*

# Makros – Infix

```
(define-syntax (infix form)
  (syntax-parse form
    [(infix stuff)
     ...]))
```

*@active group*

# Makros – Infix

```
(define-syntax (infix form)
  (syntax-parse form
    [(infix (zahl1 op zahl2))
     ...]))
```

*@active group*

# Makros – Infix

```
(define-syntax (infix form)
  (syntax-parse form
    [(infix (zahl1 op zahl2))
     #'(op zahl1 zahl2)]))
```

*@active group*

# Makros – Infix

```
repl> (infix (2 + 1))
```

*@active group*

# Makros – Infix

```
repl> (infix (2 + 1))
```

READER

Datenstruktur: (infix (2 + 1))

*@active group*

# Makros – Infix

```
repl> (infix (2 + 1))
```

READER

Datenstruktur: (infix (2 + 1))

MAKROEXPANSION

≈ (infix (2 + 1))  
≈ #<syntax(+ 2 1)> [Datenstruktur!]

*@active group*

# Makros – Infix

```
repl> (infix (2 + 1))
```

READER

Datenstruktur: (infix (2 + 1))

MAKROEXPANSION

≈ (infix (2 + 1))  
≈ #<syntax(+ 2 1)> [Datenstruktur!]

EVAL

#<syntax(+ 2 1)>  
≈ 3

*@active group*

# String switch

<https://docs.oracle.com/javase/8/docs/technotes/guides/language/stringswitch.html>

*"In the JDK 7 release, you can use a String object in the expression of a switch statement:"*

```
public String getTypeOfDayWithSwitchStatement(String dayOfWeekArg)
    String typeOfDay;
    switch (dayOfWeekArg) {
        case "Monday":
            typeOfDay = "Start of work week";
            break;
        case "Tuesday":
        case "Wednesday":
        case "Thursday":
            typeOfDay = "Midweek";
            break;
    ...
}
```

*@active group*

# String switch

Wollen folgende Syntax implementieren

```
(define y "Work")  
  
(switch y  
  [case "Holiday" -> "I am not around"]  
  [case "Work"     -> "How can I help?"])
```

*@active group*

# String switch

```
(define y "Work")

(switch y
  [case "Holiday" -> "I am not around"]
  [case "Work"      -> "How can I help?"])
```

⇒ [MAKROEXPANSION]

```
(cond
  [(eq? y "Holiday") "I am not around"]
  [(eq? y "Work")    "How can I help?"])
```

*@active group*

# String switch

## Live-Coding

*@active group*

# Domänenspezifische Sprachen

*@active group*

# Domänenspezifische Sprachen

DSLs erleichtern Anwender:innen, Probleme mit **sprachlichen Mitteln aus ihrem Expertenbereich** zu lösen

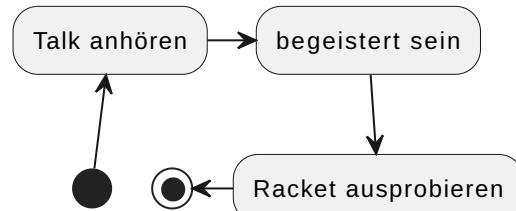
Beispiele:

- SQL
- HTML und CSS
- LaTeX
- PlantUML
- Prolog
- ...

# Domänenspezifische Sprachen

## PlantUML

```
@startuml  
(*) -up-> "Talk anhören"  
-right-> "begeistert sein"  
--> "Racket ausprobieren"  
-left-> (*)  
@enduml
```



*@active group*

# Eigene Syntax für DSLs in Racket

Datenbank-DSL: Key-Value-Paare speichern, anfordern, ausdrucken

*@active group*

# Eigene Syntax für DSLs in Racket

Datenbank-DSL: Key-Value-Paare speichern, anfordern, ausdrucken

```
SHOW-DB
PUT "milk" 1.50
PUT "water" 1.00
x = GET "water"
PRINT x
SHOW-DB
y = GET "milk"
PRINT "Summe"
```

*@active group*

# Eigene Syntax für DSLs in Racket

Live-Coding

*@active group*

# Takeaways

*@active group*

# Takeaways

- Makros sind ein mächtiges Werkzeug

*@active group*

# Takeaways

- Makros sind ein mächtiges Werkzeug
- Racket ermöglicht es, Makros sehr einfach zu schreiben

*@active group*

# Takeaways

- Makros sind ein mächtiges Werkzeug
- Racket ermöglicht es, Makros sehr einfach zu schreiben
  - Syntax kann sehr einfach erweitert werden

*@active group*

# Takeaways

- Makros sind ein mächtiges Werkzeug
- Racket ermöglicht es, Makros sehr einfach zu schreiben
  - Syntax kann sehr einfach erweitert werden
- DSLs in Racket schreiben ist einfach und toll

*@active group*

# Takeaways

- Makros sind ein mächtiges Werkzeug
- Racket ermöglicht es, Makros sehr einfach zu schreiben
  - Syntax kann sehr einfach erweitert werden
- DSLs in Racket schreiben ist einfach und toll

↝ Racket ist toll

# Wie gehts weiter?



- **Blog:** [funktionale-programmierung.de](http://funktionale-programmierung.de)
- Wir geben Schulungen in:
  - Einführung in die **funktionale Programmierung**
  - iSAQB FOUNDATION: Grundausbildung Softwarearchitektur
  - iSAQB FUNAR: **Funktionale Softwarearchitektur**
  - iSAQB FLEX: Microservices und **Self-Contained Systems**



# Add-On – riposte

## Riposte—Scripting Language for JSON-based HTTP APIs

by Jesse Alama <jesse@lisp.sh>

“  
*Riposte is a scripting language for evaluating JSON-bearing HTTP responses. The intended use case is a JSON-based HTTP API. It comes with a commandline tool, riposte, which executes Riposte scripts.*

@active group

# Add-On – riposte

```
#lang riposte

^Content-Type := "application/json"
# set a base URL
%base := https://api.example.com:8441/v1/

$uuid := @UUID with fallback "abc"

GET cart/{uuid} responds with 2XX
```

*@active group*

# Add-On – riposte

```
#lang riposte

# Now add something to the cart:

$productId := 41966
$qty := 5
$campaignId := 1

$payload := {
    "product_id": $productId,      # extend JSON syntax:
    "campaign_id": $campaignId,   # - use Riposte variables
    "qty": $qty                  # - add comments to JSON
}
```

POST \$payload to cart/{uuid}/items responds with 200

```
$itemId := /items/0/cart_item_id # extract the item ID

$itemId is an integer
```

*@active group*

# Add-On

## Makro-Hygiene

*@active group*

# Add-On

## Makro-Hygiene

```
(define-syntax (unless form)
  (syntax-parse form
    [(_unless test consequent)
     #'(if (not test)
           consequent
           #f)]))
```

*@active group*

# Add-On

## Makro-Hygiene

```
(define-syntax (unless form)
  (syntax-parse form
    [(_unless test consequent)
     #'(if (not test)
           consequent
           #f)]))
```

```
(let [[not "komischer wert"]]
  (unless (< 3 1)
    (print "Hallo")))
```

*@active group*

# Add-On

## Makro-Hygiene

```
(define-syntax (do-smth form)
  (syntax-parse form
    [(do-smth expr pos non-pos)
     #'(let [[result expr]]
         (cond
           [(positive? result) pos]
           [else non-pos]))]))
```

*@active group*

# Add-On

## Makro-Hygiene

```
(define-syntax (do-smth form)
  (syntax-parse form
    [(do-smth expr pos non-pos)
     #`(let [[result expr]]
         (cond
           [(positive? result) pos]
           [else non-pos]))])

(let [[result "Es ist positiv"]]
  (do-smth (+ 3 2) result "nicht positiv")))
```

*@active group*

