Racket Reference Sheet

Jonas Milkovits

Last Edited: 28. April 2020

Inhaltsverzeichnis

1 Einleitung: Funktionales Programmiern

Funktionale Programmierkonzepte	▷ Auch `Java` enthält auch funktionale Konzepte ▷ Unser gewähltes Beispiel: `HtDP-TL` ◇ Dialekt von `Racket` ◇ `Racket` Dialekt von `Scheme` ▷ Wir sprechen hier aber der Einfachheit halber von `Racket`
Funktionales Programmieren	▷ Funktionen sind zentrale Bausteine ◇ $f : D_1\ x\ D_2\ x\ ...\ x\ D_n \rightarrow R$ ▷ Programmdesign ◇ Zerlegung der zu erstellenden Funktionalität in Funktionen ◇ Funktionen rufen andere grundlegende Funktionen auf ▷ Funktionen werden variiert durch Parameter, die auch Funktionen sind
Deklaratives Programmieren	▷ Größere Sprachfamilie ◇ Funktionales Programmieren: Untersprache ▷ Grundsätzlicher Gedanke des deklarativen Programmierens: ◇ Nur Angabe der Formelfür das Ergebnis ◇ Nicht Angabe der Befehle, die ausgeführt werden sollen ▷ `Java`: imperativer Programmierstil ▷ Konsequenzen: ◇ Keine zeitlichen Abläufe ◇ Keine Vererbungskonzepte/Objektidentität ▷ Jeder Aufruf einer Funktion kann durch den Rückgabewert ersetzt werden ▷ Funktion liefert für selbe Parameter **immer** das selbe Ergebnis ▷ Funktionen haben nur Rückgabewerte, keine Seiteneffekte ▷ Fachbegriff: `referenzielle Transparenz`

2 Datentypen

Zahlen (number)	▷ Exakte Zahlen: ◇ ganzzahlig: `123` ◇ rational: `3/5` ▷ Nichtexakte Zahlen: ◇ `(sqrt 2)` - Setzen in Klammern, da Funktionsaufruf" ◇ Ergebnisdarstellung mit `#i` vor Zahl - `(sqrt 5) ; #i6.480...` ▷ Komplexe Zahlen: ◇ `3.14159+3/5i` ◇ Realteil + Imaginärteil + i
Symbole	▷ Symbol steht für nichts, hat nur für Programmierer eine Bedeutung ▷ Erzeugung: ◇ `(define last-name 'Spielberg)` ▷ Funktionen: ◇ `(symbol=? 'Hello 'World)` - Liefert genau dann `#t`, falls beide Symbole gleich sind ▷ Vergleich auf Gleichheit: `(symbol=? param1 param2)`
Boolean	▷ `#t` für `true` ▷ `#f` für `false` ▷ Boolesche Verknüpfungsoperatoren: ◇ Veroderung: `(or b1 b2 b3)` ◇ Verundung: `(and b1 b2 b3)` ◇ Negation: `(not b1)` ▷ Vergleichsoperatoren: ◇ `(= x1 x2 x3) ; (and (= x1 x2)(= x2 x3))` ◇ `(< x1 x2 x3) ; (and (< x1 x2)(< x2 x3))` ◇ `(<= x1 x2 x3)` ▷ Boolsche Funktionen ◇ `(integer? value)` - Liefert `#t` zurück, falls `value` ganzzahlig - z.B. `(if (integer? (- x y)) #t #f)` ◇ `(number? value)` - `#t`, falls `value` eine Zahl ist ◇ `(real? value)` - `#t`, falls `value` keine imaginäre Zahl ist ◇ `(rational? value)` - `#t`, falls `value` eine rationale Zahl ist ◇ `(natural? value)` - `#t`, falls `value` natürliche Zahl ◇ `(symbol? value)` - `#t`, falls `value` ein Symbol ist ◇ `(empty? list)` - `#t`, falls `list` leer ist ◇ `(struct-name? value)` - `#t`, falls `value` vom Typ des `Structs struct-name` ist

3 filter, map und fold

`filter`	▷ Fitlerung gewisser Elemente aus einer Liste mittels Prädikat ▷ Vertrag: `(X -> boolean) (list of X) -> (list of X)` ▷ `(filter pred list)` ◇ Übergabe des Prädikats als ersten Parameter ◇ Parameter des Prädikats muss hier von `X` sein ◇ Übergabe der Liste als zweiten Parameter ▷ Lambda-Beispiel: ◇ `(filter (lambda (x) -> (< x 10)) list1)` - Gibt Liste zurück mit Werten kleiner gleich 10 ◇ Verträge bei Verwendung solcher Lambda-Funktionen notwendig - z.B. innerhalb der verwendenden Methode
`map`	▷ Abbildung von Werten auf einen anderen Wert ▷ Vertrag: `(X -> Y) (list of X) -> (list of Y)` ▷ `(map fct list)` ◇ Übergabe der anzuwendenden Funktion als ersten Parameter ◇ Übergabe der Liste als zweiten Parameter ▷ Lambda-Beispiel: ◇ `(map (lambda(x) -> (* x 10)) list1)` - Multipliziert alle Werte der Liste mit 10 ◇ Vertrag genau wie bei `filter`
`fold`	▷ Faltungsoperation, in Racket `lfold` und `rfold` ◇ `lfold`: von links nach rechts ◇ `rfold`: von rechts nach links ▷ Faltungsoperation rechnet alle Werte nach Berechnungsvorschrift zusammen ▷ Vertrag: `X (X Y -> X) (list of Y) -> X` ▷ (rfold init fct list) ◇ Übergabe des Startwerts als ersten Parameter ◇ Übergabe der Funktion als zweiten Parameter ◇ Übergabe der Liste als dritten Parameter ▷ Lambda-Beispiel: ◇ `(rfold 0 (lambda (x,y) -> (+ x y)) list1)` - Rechnet alle Werte der Liste zu einem einzigen Wert mit `+` zusammen ◇ Vertrag genau wie bei `filter` ▷ Bei Listen Verwendung von `empty` als Initialwert

4 Funktionen

Erzeugung	▷ `(define (name param1 param2) (Ausdruck)) ; Funktion` ◇ z.B. `(define (add x y) (+ x y))` ◇ `define` sagt, dass Konstante oder Funktion definiert wird ◇ Konstante: `(define name value)` ◇ kein `return` notwendig
Aufruf	▷ `(name param1 param2)` ◇ z.B. `(add 2.71 3.14)` ◇ Ergebnis wird ins Ausgabefenster des Bildschirms geschrieben
Arithmetische Operationen	▷ `(+ 2 3) ; 5` ▷ `(- -2 3 ; -5)` ▷ `(/ 37 30) ; 1.23..` ▷ `(modulo 20 3) ; 2` ▷ Verkettung: ◇ `(* (+ 2 3) 4) ; 20` ▷ Auch mehrere Operanden ◇ `(+ 3 4 5)` ◇ `(- 1 2 3) ; 1 - (2 + 3)` ◇ `(/ 1 2 3) ; 1 / (2 * 3)`
Mathematische Funktionen	▷ `(floor 3.14) ; 3` ◇ Abrunden des übergebenen Wertes ▷ `(ceiling 3.14) ; 4` ◇ Aufrunden des übergebenen Wertes ▷ `(gcd 357 753 573)` ◇ Größter gemeinsamer Teiler ◇ `greatest common denominator` ▷ `(modulo 753 357)` ◇ Rest der ganzzahigen Division
Typ einer Funktion	▷ Prüfung erst zur Laufzeit, ob Typen der Operanden zur Operation passen ▷ Typenzusicherung deswegen über Verträge (siehe Vertrag)
Definitionen verstecken	▷ Zugriff auf definierte Funktionen nur innerhalb des `local`-Blocks ▷ Verwendung von `(local ... )` 1 `(define (fct x)` 2 `(local ( ; Öffnen des Blocks für lokale Definition` 3 `(define const 10)` 4 `(define (mult-const y) (* const y))) ; Blockschließung` 5 `(+ const (mult-const x)))) ; Schließen von local und define` ◇ `local` enthält in sich einen Block für lokale Definitionen ◇ Zeile 5: Die letzte Zeile stellt den Wert des `local`-Ausdrucks dar

5 Funktionen als Daten

Beispiele	▷ `(define add +)` ◇ Konstante `add` vom Typ: `number number -> number` ◇ Operationen sind in `Racket` auch Funktionen ▷ `(define-struct functions (fct1 fct))` ◇ Auch Abspeichern von Funktionen als `Struct` ▷ `(list fct1 3 fct2 +)` ◇ Funktionen auch als Listenelemente möglich
Funktionen höherer Ordnung	▷ Funktionen, die Funktionen als Parameter enthalten ▷ z.B. `(define (add fct1 x fct2 y) (+ (fct1 x) (fct2 y)))` ◇ Verwendung der Funktion gemäßihres erwarteten Typs (Vertrag)

6 Klassen

7 Konstanten

Allgemein	▷ In Racket stellt jeder Wert, der definiert wird, eine Konstante dar
Erzeugung	▷ (define name ausdruck) ◇ z.B. `(define my-pi 3.14159)` ◇ `(define my-pi (+ 3 0.14159))`
Wichtige Konstanten	▷ `pi` ▷ `e`

8 Lambda-Ausdrücke

Aufbau	▷ `(lambda (x y) -> (+ (* x x) (* y y))` ◇ Aufbau vergleichbar mit Kurzform in Java
Beispiel	1 `(define (add-unary-functions fct1 fct2)` 2 `(lambda (x,y) -> (+ (fct1 x) (fct2 y))))` ▷ Die Funktion liefert eine Funktion zurück, die eine Zahl zurückliefert ▷ `Closure` analog zu Java ▷ Anwendung: ◇ `(define (times10 a) (* 10 a))` ◇ `(define (div5 a) (/ a 5))` ◇ `((add-unary-functions times10 a) 3.14 2.71)` - `(+ (* 3.14 10) (/ 2.71 5))` - `add-unary-functions` liefert Lambda-Ausdruck zurück - Dieser wird dann mit `3.14` und `2.71` aufgerufen

9 Laufzeitchecks und Fehler

Allgemein	▷ Möglichkeit des Testens von Funktionen zur Laufzeit
Verwendung	▷ `(check-expect param1 param2)` ◇ Abbruch mit Fehlermeldung, falls inkorrekt ◇ z.B. `(check-expect (divide 15 3) 5) ; #t` ▷ `(check-within param1 param2 param3)` ◇ Test, ob Werte ausreichend nahe beeinander liegen ◇ `param3` ist dieser maximale Abstand ◇ z.B. `(check-within (divide pi e) 1.15 0.01)` ▷ `(check-error (divide 15 0) "/: division by zero")` ◇ Test, ob Fehler im Fehlerfall wirklich geworfen wird ◇ Fehlermeldung des 1. Parameters muss dem 2. Parameter entsprechen ◇ `""` geben hier einen String an ◇ Nachgucken der entsprechenden Fehlermeldung in Racket Dokumentation ▷ Wichtig: Abprüfung aller Randfälle
Werfen eines Fehlers	▷ Laufzeittests"können auch innerhalb einer Methode ausgeführt werden ▷ Bei falschemParameter kann man selbst einer `Error` werfen ▷ `(if (= y 0) (error "Division by 0") (/x y))` ◇ `error` führt zum Programmabbruch und Ausgabe der Fehlermeldung

10 Listen

Erzeugung einfacher Listen	▷ `(define list1 (list 1 2 3))` ◇ Erstellt eine Liste mit den Werten 1,2,3 ◇ Funktion `list` kann beliebig viele Parameter haben ◇ Die Elemente der Liste sind ihr Rückgabewert ▷ Listen müssen nicht homogen sein (auch heterogen möglich)
Erzeugung von Listen aus Listen	▷ `(define list2 (cons 7 list1))` ◇ Funktion `cons` fügt `7` und `list1` zu `list2` zusammen ◇ Zweiter Parameter muss zwingend eine Liste sein ◇ Erster Parameter in Liste dann auch an erster Stelle
`empty`	▷ Name für die leere Liste ▷ z.B. `(define list1 (cons 1 empty))` ▷ `empty?` überprüft, ob die Liste leer ist ◇ Wird gerne als Rekursionsanker verwendet
Funktionen auf Listen	▷ `(first list1)` ◇ Liefert den ersten Wert der Liste zurück ◇ Erwartet, dass die Liste nicht leer ist ▷ `(rest list1)` ◇ Liefert Liste zurück, die alle Elemente außer dem Ersten enthält ◇ Erwartet, dass die Liste nicht leer ist

11 Objektmodell

Allgemein	▷ Es gibt keine Objekte, nur Werte ◇ Werte sind **immer** Konstante, **nie** Variable ◇ Werte werden **immer** kopiert - Formaler Paramater innerhalb Funktion ist Kopie des aktualen Parameters ▷ Laufzeitsystem kann intern zur Optimierung von Grundlogik abweichen
Aufweichung des Objektmodells	▷ `begin` ◇ `(begin (fct1 ...) (fct2 ...))` ◇ `begin` leitet mehrere Anweisungen ein ◇ Die letzte Anweisung ist der Wert des `begin`-Ausdrucks ▷ `set!` ◇ (set! new 'test) ◇ Überschreibt Wert des ersten Parameters mit dem des Zweiten ◇ `set!` hat keinen Rückgabewert ▷ `#<void>` ◇ z.B. `(define new #<void>)` ◇ Konstante hat damit einen leeren Wert ▷ Objektidentität: ◇ generelle Vermeidung wird empfohlen ◇ `(eq? a b)` ◇ Testet ob zwei Objekte dasselbe Objekt sind ◇ Systemabhängig, interne Implementation

12 Rekursion

Allgemein	▷ Grundlegendes Konzept zur Steuerung des Programmablaufs in Funktion ◇ Verwendung anstatt von Schleifen wie in z.B. Java ◇ Schleifen widersprechen funktionaler Programmierung

Beispiel Normale Berechnung

▷ z.B. `(define (factorial n) (if (= n 0) 1 (* n (factorial (- n 1)))))`
- ◇ Zurückliefern von `1`, falls `n` gleich 0 ist
  - Ermöglicht Multiplizieren mit 1 auf niedrigster Rekursionsstufe
  - Verändert damit den Rückgabewert nicht und beendet Rekursion
- ◇ Beispiel für `factorial 2`

```
1  (factorial 2)
1  (* 2 (factorial 1))
1  (* 2 (* 1 (factorial 0)))
1  (* 2 (* 1 1)) ; Ergebnis: 2
```

Rekursion auf Listen

▷ z.B.: Summe einer Listen von Zahlen:
- ◇ Falls Liste leer ist: Summe = `0`
- ◇ Sonst Summe = erstes Elemente plus Summe der Restliste
- ◇ Folgendes Beispiel im Rahmen einer Methode `sum`
- ◇ Addiert rekursiv die Werte der Liste
- ◇ Falls Liste leer ist, wird 0 zurückgegeben und Rekursion "fällt zusammen"

```
1  (if (empty? list)
2    0
3    (+ (first list) (sum (rest list)))
```

▷ z.B.: Liste der Quadratwurzeln einer Liste
- ◇ Kerngedanke: Sukzessiver Aufbau einer neuen Liste der Quadratwurzeln
- ◇ Alle Wurzeln müssen durch die Rekursion "geschleift" werden
- ◇ Hinzufügen von `empty`, falls die Liste leer ist
- ◇ Folgendes Beispiel im Rahmen einer Methode `sqrts`

```
1  (if (empty? list)
2    empty
3    (cons (sqrt (first list)) (sqrts (rest list))))
```

▷ z.B.: Filterung einer Liste
- ◇ Selbes Konzept wie bei den Quadratwurzeln einer Liste
- ◇ Meist mit zwei `if`-Anweisungen (Rekursionsanker + Filter)
- ◇ Falls die Bedingung nicht erfüllt ist, "Überspringen" des Elements
  - Aufruf der rekursiven Methode ohne `cons` davor

```
1  (define (filter-fct list x)
2    (if (empty? list)
3      empty
4      (if (< (first list) x)
5        (cons (first list) (filter-fct (rest list) x))
6        (filter-fct (rest list) x))))
```

Objektmodell

▷ Liste ist eine Folge von **Werten**, nicht von **Objekten**

▷ Eine gefilterte Liste enthält **Kopien** von Werten

Randfälle bei Listen

▷ Ausgabeliste leer, trotz nicht leerer Eingabeliste

▷ Alle Elemente der Eingabeliste in Ausgabeliste

▷ Test auf Vorzeichen bei Filterungen

▷ Eingabeliste leer

Akkumulatoren	▷ Wird meist durch lokale Hilfsmethode implementiert
	▷ Beispielsmethode: Akkumulation einer Liste
	`1 (define (list-of-sums input-list)`
	`2 (local`
	`3 ((define (list-of-sum-with-accu list accu))`
	`4 (cond`
	`5 [(empty? list) (list accu)]`
	`6 [else (cons accu (list-of-sums-with-accu (rest list)`
	`7 (+ (first list) accu)))]))`
	`8 (list-of-sums-with-accu (rest input-list) (first input-list))))`
	◇ Zeile 2: Einrichtung einer lokalen Umgebung
	◇ Zeile 3: Definitionsstart der lokalen Methode mit Akkumulator
	◇ Zeile 8: Letzter Ausdruck des `local`-Ausdrucks und deswegen Rückgabewert
	◇ Zeile 5: Rekursionsanker: Falls Liste leer, Rückgabe des `accu` als Liste
	◇ Beispielaufruf: `(list-of-sums (10 18 22 30))`
	- Ausgabe: `(list 10 28 50 80)`

13 Structs

Allgemein	▷ Zusammenfassung von Elementen, vergleichbar mit Klasse ohne Methoden
Erzeugung	▷ `(define-struct name (attribute1 attribute2 attribute3))` ◇ Attribute werden in `Racket` Felder genannt
"Objekterzeugung"	▷ z.B. `(define test (make-name 'Hallo 2 (list 1 2 3)))` ▷ "Konstruktor" wird automatisch erzeugt ▷ Initialisierungen der Felder werden in selber Reihenfolge übergeben
Zugriff	▷ `(name-attribute1 object-name)` ◇ z.B. `student-last-name 247852` ◇ Abfrage des Nachnamens des `student`-Objektes mit den Namen 247852

14 Syntax

Präfixnotation	▷ Zuerst der Operand, danach die Operanden ◇ (+ 1 2)
Klammersetzung	▷ Jede Einheit, die nicht atomar ist, wird in Klammern gesetzt ◇ Zusammengesetzte Ausdrücke ◇ Funktionen allgemein ▷ Keine unterschiedlichen Bindungsstärken, immer Setzen aller Klammern
Kommentare	▷ Einzelne Zeile: `;`
Identifier	▷ Keine Zahlen ▷ Keine Whitespaces ▷ Konventionen: ◇ Keine Großbuchstaben ◇ Bindestriche zwischen den einzelnen Wörtern - z.B. `this-identifier-conforms-to-all-conventions`

15 Verzweigung, cond

if-Anweisung	▷ Boolsche Funktion mit drei Parametern ▷ `( if(bedindung) anweisung-if-true anweisung-if-false)` ◇ Muss wiede jeder andere Funktion in Klammern stehen ◇ Liefert ersten Parameter zurück falls `true` ◇ z.B. `(define (my-abs x) (if (< 0 x) -x x))` ▷ Verschachtelung von `if`-Anweisungen auch möglich

cond Funktion	▷ Bei mehreren `if`-Anweisungen meist der bessere Ersatz
	▷ Stark an `switch`-Anweisung aus `Java` angelegt
	▷ Wird bei Rekursion z.B. für Randfälle oder Rekursionsanker verwendet
	▷ Aufbau:
	`1 (cond`
	`2 [(empty? list) 2]`
	`3 [(number? a) 0]) 3 [else 1])`
	▷ `cond`-Funktion hat eine variable Anzahl von Anweisungen
	▷ Jede Anweisung wird in `[]` gefasst und bildet einen Fall ab
	▷ Aufbau eines Falls: `[(bedingung) anweisung]`
	▷ Überprüfung aller Fälle der Reihe nach
	◇ Falls ein Fall eintritt, ist die Anweisung dort der Rückgabewert
	▷ `else` deckt den Fall ab, falls keiner der vorangehenden eintritt

16 Vertrag

Allgemein	▷ Warum? ◇ Typprüfung erst zur Laufzeit ◇ Fehlervermeidung ▷ "Vertrag": ◇ Nutzer erfüllt seinen Teil des Vertrags (Precondition) ◇ Dann erfüllt Funktion ihren Teil des Vertrags

Aufbau	`;; Type: number number -> number`
	`;;`
	`;; Returns: the sum of two parameters`
	▷ `Type:` Aufzählung der Paramater nach Reihenfolge des Auftretens
	▷ `->`: Angabe des Rückgabetyps nach dem Pfeil
	▷ `Returns`: Kurze Beschreibung des Rückgabewertes
	▷ Nutzung von `;;` statt `;` ist hier Konvention

Weitere Elemente	▷ `;; Precondition`: Angabe für Parameterrichtlinien ▷ `(list of number)` im `Type` für Listen ◇ z.B. `(list of number) -> number` ▷ Unterschiedliche Typen in Liste: `(list of ANY)` ◇ Verwendung mehrerer `ANY`: Keine Abhängigkeit voneinander ▷ Feste Typen in Liste: `(list of student)` ▷ `X`: Platzhalter für beliebigen Datentyp ◇ Bei Verwendung von mehreren `X`: `X` sind vom gleichen Typ ◇ Falls diese unterschiedlich sein sollen: Verwendung von `Y,Z`
Funktionen höherer Ordnung	▷ z.B. `(define (add fct1 x fct2 y) (+ (fct1 x) (fct2 y)))` ◇ `(number -> number) number (number -> number) number -> number` ◇ Beschreibung des Funktionstyps an sich ▷ Vertrag für Lambda-Funktionen vom selben Aufbau ◇ z.B. `... -> (number number -> number)` ◇ Prädikat: `(X -> boolean) ... -> ...` ◇ Vertrag für jede Lambda-Funktion notwendig ◇ sowohl innerhalb, als auch im oberen Vertrag