Racket Reference Sheet

Jonas Milkovits

Last Edited: 27. April 2020

Inhaltsverzeichnis

1 Einleitung: Funktionales Programmiern **1**

2 Datentypen **2**

3 Funktionen **3**

4 Klassen **3**

5 Konstanten **3**

6 Laufzeitchecks und Fehler **4**

7 Listen **4**

8 Objektmodell **4**

9 Rekursion **5**

10 Syntax **6**

11 Verzweigung, cond **6**

12 Vertrag **6**

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

3 Funktionen

Erzeugung	▷ `(define (name param1 param2) (Ausdruck)) ; Funktion` ◇ z.B. `(define (add x y) (+ x y))` ◇ `define` sagt, dass Konstante oder Funktion d ◇ 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

4 Klassen

5 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`

6 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

7 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
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

8 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	▷ TODO in 4D

9 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

10 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`

11 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

12 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`