

Racket Reference Sheet

Jonas Milkovits

Last Edited: 25. April 2020

Inhaltsverzeichnis

| | | |
|----|--|---|
| 1 | Einleitung: Funktionales Programmieren | 1 |
| 2 | Datentypen | 2 |
| 3 | Funktionen | 3 |
| 4 | Klassen | 3 |
| 5 | Konstanten | 3 |
| 6 | Laufzeitchecks und Fehler | 4 |
| 7 | Objektmodell | 4 |
| 8 | Rekursion | 4 |
| 9 | Syntax | 5 |
| 10 | Verzweigung, switch | 5 |
| 11 | Vertrag | 5 |

1 Einleitung: Funktionales Programmieren

| | |
|---------------------------------|---|
| Funktionale Programmierkonzepte | <ul style="list-style-type: none">▷ Auch Java enthält auch funktionale Konzepte▷ Unser gewähltes Beispiel: HtDP-TL<ul style="list-style-type: none">◊ Dialekt von Racket◊ Racket Dialekt von Scheme▷ Wir sprechen hier aber der Einfachheit halber von Racket |
| Funktionales Programmieren | <ul style="list-style-type: none">▷ Funktionen sind zentrale Bausteine<ul style="list-style-type: none">◊ $f : D_1 \times D_2 \times \dots \times D_n \rightarrow R$▷ Programmdesign<ul style="list-style-type: none">◊ Zerlegung der zu erstellenden Funktionalität in Funktionen◊ Funktionen rufen andere grundlegende Funktionen auf▷ Funktionen werden variiert durch Parameter, die auch Funktionen sind |
| Deklaratives Programmieren | <ul style="list-style-type: none">▷ Größere Sprachfamilie<ul style="list-style-type: none">◊ Funktionales Programmieren: Untersprache▷ Grundsätzlicher Gedanke des deklarativen Programmierens:<ul style="list-style-type: none">◊ Nur Angabe der Formel für das Ergebnis◊ Nicht Angabe der Befehle, die ausgeführt werden sollen▷ Java: imperativer Programmierstil▷ Konsequenzen:<ul style="list-style-type: none">◊ 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) | <ul style="list-style-type: none">▷ Exakte Zahlen:<ul style="list-style-type: none">◊ ganzzahlig: 123◊ rational: 3/5▷ Nichtexakte Zahlen:<ul style="list-style-type: none">◊ (<code>sqrt 2</code>)- Setzen in Klammern, da Funktionsaufruf"◊ Ergebnisdarstellung mit #i vor Zahl- (<code>sqrt 5</code>) ; #i6.480...▷ Komplexe Zahlen:<ul style="list-style-type: none">◊ 3.14159+3/5i◊ Realteil + Imaginärteil + i |
| Symbole | <ul style="list-style-type: none">▷ Symbol steht für nichts, hat nur für Programmierer eine Bedeutung▷ Erzeugung:<ul style="list-style-type: none">◊ (<code>define last-name 'Spielberg</code>)▷ Funktionen:<ul style="list-style-type: none">◊ (<code>symbol=? 'Hello 'World</code>)- Liefert genau dann #t, falls beide Symbole gleich sind |
| Boolean | <ul style="list-style-type: none">▷ #t für true▷ #f für false▷ Boolesche Verknüpfungsoperatoren:<ul style="list-style-type: none">◊ Veroderung: (<code>or b1 b2 b3</code>)◊ Verundung: (<code>and b1 b2 b3</code>)◊ Negation: (<code>not b1</code>)▷ Vergleichsoperatoren:<ul style="list-style-type: none">◊ (<code>= x1 x2 x3</code>) ; (<code>and (= x1 x2)(= x2 x3)</code>)◊ (<code>< x1 x2 x3</code>) ; (<code>and (< x1 x2)(< x2 x3)</code>)◊ (<code><= x1 x2 x3</code>)▷ Boolesche Funktionen<ul style="list-style-type: none">◊ (<code>integer? value</code>)- Liefert #t zurück, falls value ganzzahlig- z.B. (<code>if (integer? (- x y)) #t #f</code>)◊ (<code>number? value</code>)- #t, falls value eine Zahl ist◊ (<code>real? value</code>)- #t, falls value keine imaginäre Zahl ist◊ (<code>rational? value</code>)- #t, falls value eine rationale Zahl ist◊ (<code>natural? value</code>)- #t, falls value natürliche Zahl◊ (<code>symbol? value</code>)- #t, falls value ein Symbol ist |

3 Funktionen

| | |
|---------------------------|--|
| Erzeugung | <ul style="list-style-type: none"> ▷ <code>(define (name param1 param2) (Ausdruck)) ; Funktion</code> ◊ z.B. <code>(define (add x y) (+ x y))</code> ◊ <code>define</code> sagt, dass Konstante oder Funktion d ◊ Konstante: <code>(define name value)</code> ◊ kein <code>return</code> notwendig |
| Aufruf | <ul style="list-style-type: none"> ▷ <code>(name param1 param2)</code> ◊ z.B. <code>(add 2.71 3.14)</code> ◊ Ergebnis wird ins Ausgabefenster des Bildschirms geschrieben |
| Arithmetische Operationen | <ul style="list-style-type: none"> ▷ <code>(+ 2 3) ; 5</code> ▷ <code>(- -2 3 ; -5)</code> ▷ <code>(/ 37 30) ; 1.23..</code> ▷ <code>(modulo 20 3) ; 2</code> ▷ Verkettung: <ul style="list-style-type: none"> ◊ <code>(* (+ 2 3) 4) ; 20</code> ▷ Auch mehrere Operanden <ul style="list-style-type: none"> ◊ <code>(+ 3 4 5)</code> ◊ <code>(- 1 2 3) ; 1 - (2 + 3)</code> ◊ <code>(/ 1 2 3) ; 1 / (2 * 3)</code> |
| Mathematische Funktionen | <ul style="list-style-type: none"> ▷ <code>(floor 3.14) ; 3</code> <ul style="list-style-type: none"> ◊ Abrunden des übergebenen Wertes ▷ <code>(ceiling 3.14) ; 4</code> <ul style="list-style-type: none"> ◊ Aufrunden des übergebenen Wertes ▷ <code>(gcd 357 753 573)</code> <ul style="list-style-type: none"> ◊ Größter gemeinsamer Teiler ◊ <code>greatest common denominator</code> ▷ <code>(modulo 753 357)</code> <ul style="list-style-type: none"> ◊ Rest der ganzzahigen Division |
| Typ einer Funktion | <ul style="list-style-type: none"> ▷ Prüfung erst zur Laufzeit, ob Typen der Operanden zur Operation passen ▷ Typenzusicherung deswegen über Verträge (siehe Vertrag) |
| Definitionen verstecken | <ul style="list-style-type: none"> ▷ Zugriff auf definierte Funktionen nur innerhalb des <code>local</code>-Blocks ▷ Verwendung von <code>(local ...)</code> <pre> 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 </pre> <ul style="list-style-type: none"> ◊ <code>local</code> enthält in sich einen Block für lokale Definitionen ◊ Zeile 5: Die letzte Zeile stellt den Wert des <code>local</code>-Ausdrucks dar |

4 Klassen

5 Konstanten

| | |
|---------------------|--|
| Allgemein | ▷ In Racket stellt jeder Wert, der definiert wird, eine Konstante dar |
| Erzeugung | <ul style="list-style-type: none"> ▷ <code>(define name ausdruck)</code> ◊ z.B. <code>(define my-pi 3.14159)</code> ◊ <code>(define my-pi (+ 3 0.14159))</code> |
| Wichtige Konstanten | <ul style="list-style-type: none"> ▷ <code>pi</code> ▷ <code>e</code> |

6 Laufzeitchecks und Fehler

| | |
|----------------------|---|
| Allgemein | <ul style="list-style-type: none"> ▷ Möglichkeit des Testens von Funktionen zur Laufzeit |
| Verwendung | <ul style="list-style-type: none"> ▷ <code>(check-expect param1 param2)</code> <ul style="list-style-type: none"> ◊ Abbruch mit Fehlermeldung, falls inkorrekt ◊ z.B. <code>(check-expect (divide 15 3) 5) ; #t</code> ▷ <code>(check-within param1 param2 param3)</code> <ul style="list-style-type: none"> ◊ Test, ob Werte ausreichend nahe beieinander liegen ◊ <code>param3</code> ist dieser maximale Abstand ◊ z.B. <code>(check-within (divide pi e) 1.15 0.01)</code> ▷ <code>(check-error (divide 15 0) "/: division by zero")</code> <ul style="list-style-type: none"> ◊ Test, ob Fehler im Fehlerfall wirklich geworfen wird ◊ Fehlermeldung des 1. Parameters muss dem 2. Parameter entsprechen ◊ <code>""</code> geben hier einen String an ◊ Nachgucken der entsprechenden Fehlermeldung in Racket Dokumentation ▷ Wichtig: Abprüfung aller Randfälle |
| Werfen eines Fehlers | <ul style="list-style-type: none"> ▷ Laufzeittests können auch innerhalb einer Methode ausgeführt werden ▷ Bei falschem Parameter kann man selbst einen Error werfen ▷ <code>(if (= y 0) (error "Division by 0") (/x y))</code> <ul style="list-style-type: none"> ◊ error führt zum Programmabbruch und Ausgabe der Fehlermeldung |

7 Objektmodell

| | |
|-------------------------------|--|
| Allgemein | <ul style="list-style-type: none"> ▷ Es gibt keine Objekte, nur Werte <ul style="list-style-type: none"> ◊ Werte sind immer Konstante, nie Variable ◊ Werte werden immer kopiert <ul style="list-style-type: none"> - Formaler Parameter innerhalb Funktion ist Kopie des aktuellen Parameters ▷ Laufzeitsystem kann intern zur Optimierung von Grundlogik abweichen |
| Aufweichung des Objektmodells | <ul style="list-style-type: none"> ▷ TODO in 4D |

8 Rekursion

| | |
|--------------------------------|---|
| Allgemein | <ul style="list-style-type: none"> ▷ Grundlegendes Konzept zur Steuerung des Programmablaufs in Funktion <ul style="list-style-type: none"> ◊ Verwendung anstatt von Schleifen wie in z.B. Java ◊ Schleifen widersprechen funktionaler Programmierung |
| Beispiel Normale Berechnung | <ul style="list-style-type: none"> ▷ z.B. <code>(define (factorial n) (if (= n 0) 1 (* n (factorial (- n 1)))))</code> <ul style="list-style-type: none"> ◊ Zurückliefern von 1, falls <code>n</code> gleich 0 ist <ul style="list-style-type: none"> - Ermöglicht Multiplizieren mit 1 auf niedrigster Rekursionsstufe - Verändert damit den Rückgabewert nicht und beendet Rekursion ◊ Beispiel für <code>factorial 2</code> <pre>1 (factorial 2) 1 (* 2 (factorial 1)) 1 (* 2 (* 1 (factorial 0))) 1 (* 2 (* 1 1)) ; Ergebnis: 2</pre> |

9 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: ; |
| Identifizier | ▷ Keine Zahlen ▷ Keine Whitespaces ▷ Konventionen: ◊ Keine Großbuchstaben ◊ Bindestriche zwischen den einzelnen Wörtern - z.B. <code>this-identifizier-conforms-to-all-conventions</code> |

10 Verzweigung, switch

| | |
|--------------|---|
| if-Anweisung | ▷ Boolesche Funktion mit drei Parametern ▷ (<code>if(bedingung) anweisung-if-true anweisung-if-false</code>) ◊ Muss wieder jeder andere Funktion in Klammern stehen ◊ Liefert ersten Parameter zurück falls <code>true</code> ◊ z.B. <code>(define (my-abs x) (if (< 0 x) -x x))</code> ▷ Verschachtelung von if-Anweisungen auch möglich |
|--------------|---|

11 Vertrag

| | |
|------------------|---|
| Allgemein | ▷ Warum? ◊ Typprüfung erst zur Laufzeit ◊ Fehlervermeidung ▷ "Vertrag": ◊ Nutzer erfüllt seinen Teil des Vertrags (Precondition) ◊ Dann erfüllt Funktion ihren Teil des Vertrags |
| Aufbau | <code>;; Type: number number -> number</code> <code>;;</code> <code>;; Returns: the sum of two parameters</code> ▷ Type: Aufzählung der Parameter nach Reihenfolge des Auftretens ▷ ->: Angabe des Rückgabetyps nach dem Pfeil ▷ Returns: Kurze Beschreibung des Rückgabewertes ▷ Nutzung von <code>;;</code> statt <code>;</code> ist hier Konvention |
| Weitere Elemente | ▷ <code>;; Precondition:</code> Angabe für Parameterrichtlinien |