HtDP-TL Stichwortverzeichnis

20. Januar 2020

	Höherer Ordnung	• Funktionen, die Funktionen als Parameter oder Rückgabe haben • Vertrag: z.B.: (number → number) number → number	4C/18
	Aufweichung	• Zeitliche Abläufe in Racket: • #<void> als Wert → Konstante mit leerem Wert • (begin ...) → Block, der mehrere Anweisungen enthalten kann → Letzter Ausdruck = Rückgabewert • (set! my-constant value) → Überschreibt Konstante	4D/56

Funktionales Programmieren	Funktionen	• zentrale Bausteine des funktionalen Programmierens • Zerlegung der zu erstellenden Funktionalität in Funktionen	4A/23
	Deklaratives Programmieren	• Nur Verwendung der Formel des Ergebnis → keine zeitlichen Abläufe → keine Vorstellung vom Computerspeicher	4A/31
	referentielle Transparenz	• Einziger Effekt von Funktionen ist der Rückgabetyp → void Methoden sinnlos	4A/32

Klammern	atomar	• Einzelne Bestandteile müssen nicht geklammert werden (atomar)	4A/69
	Zusammengesetzter Ausdruck	• Klammern sind notwendig	4A/69
	Notwendig	• Keine Punkt vor Strich Rechnung → Klammern notwendig	4A/70

Kommentare	Einzeilig	• ; ...	4A/56
	Funktion	• Festlegung der Eingabeparameter und Ausgabeparameter → Bsp.: ;; Type: number number → number ;; Returns: ... → Vertrag der Funktion • ;; Precondition: ... → siehe Laufzeitchecks • höhere Ordnung: z.B. (number → number) number → number	4A/106
	iff	• gängige Abkürzung für 'if, and only if,..'	4B/75

Konstanten	Häufigkeit	• nur Konstanten in Racket	4A/33
	Syntax	• (define myConstant Wert) • zusammengesetzter Ausdruck → Wert in Klammern	4A/53

Konventionen	Identifier	• Keine Großbuchstaben • Bindestriche zwischen Wörtern → my-identifier	4A/58
	Kommentare	• einzeilig: ;; statt ; (bessere Sichtbarkeit)	

Laufzeitchecks	Funktion	• Unterbrechen des Programms und Testen der Funktion • falls true → Programm wird weitergeführt • falls false → Abbruch des Programms mit Fehlermeldung	4A/112
	check-expect	• (check-expect (divide 15 3) 5) • Test ob 15/3 = 5	4A/112
	check-within	• (check-within (divide pi e) 1.15 0.01) • Test ob pi/e weniger als 0.01 von 1.15 abweicht	4A/113
	check-error	• (check-error (divide 15 0) "/: division by zero") • Überprüfen ob Fehlermeldungen gleich (Dokumentation) • Fehlermeldungen sind Strings	4A/114

	In Funktion	• z.B.: Division (if (= y 0) (error "Divison by 0")(/ x y))	4A/117

Lambda-Ausdrücke	Syntax	• (lambda (x y) (+(* x x)(* y y)))	4C/99
	Beispiel	• ;; Type: (number -> number) (number -> number) ;; -> (number number -> number) • (define (name fct1 fct2) (lambda (x,y) (+(fct1 x)(fct2 y))) → zurückgelieferter Wert auch Funktion	4C/105
	Beispiel Verwendung	• ((name bspfct1 bspfct2) param1 param2)) → Return aus Funktion name"wird danach mit param1 und param2 aufgerufen → Anwendung von bspfct1 auf param1	4C/108
	filter	• Filterung abhängig von Prädikat • Beispiel: (define (my-filter prädikat liste)...) • (my-filter (lambda(x) (< x 10)) liste) • Durch Verwendung von Lambda → variable Filter	4C/110
	map	• Anwendung einer Funktion auf jedes Element der Liste • Beispiel: (define (my-map fct list)...) • (my-map (lambda(x) (* x 10) list) • Durch Verwendung von Lambda → variable Anpassung	4C/125
	fold	• Zusammenrechnen einer Liste nach Vorschrift • foldr von rechts, foldl von links • Beispiel: (define (my-fold init fct list)...) • (my-fold 0 (lambda(x,y) (+ x y) list) • Durch Verwendung von Lambda → variable Verrechnung	4C/139

Listen	Syntax	• (define my-list (list 1 2 3))	4B/3
	cons	• Anhängen an Liste • z.B.: (define my-list2 (cons 0 list1)) → 0 1 2 3	4B/6
	empty	• leere Liste (z.B.: (... (cons 0 empty))	4B/8
	Funktionen	• (first my-list) → erstes Element • (rest my-list) → Alle Elemente außer Erstes	4B/9
	Vertrag	• ;; Type: (list of ...) → ... • ;; Type: (list of ANY) → ... , falls Typ beliebig • ;; Type: X → X , Eingabetyp beliebig, Ausgabe vom selben Typ • ;; Type: ANY → ANY , Eingabetyp beliebig, Ausgabetyp auch	4B/16
	Rekursion	• Beispiel Summe: → (if (empty? list) 0 (+ (first list) (sum (rest list)))) • Beispiel Liste als Ausgabe: → .. (if (empty? list) empty (cons (sqrt (first list) (sqrts (rest list)))))	4B/41
	Filter	• Aussortieren von gewissen Elementenn • (define (filter list) (if (Sortierbedingung) → (cons (first list) (filter list)) , falls #t → (filter list))) , falls #f	4B/42
	homogen heterogen	• Listen auch mit unterschiedlichen Typen möglich → heterogen → ;; Type: (list of ANY)...	4B/107

Objektmodell	nur Konstanten	→ Werte werden immer kopiert	4A/126
	Idealisiert	• Abweichung vom Modell im Hintergrund zur Optimierung	4A/129

Rekursion	Zweck	• grundlegendes Konzept zur Steuerung des Programmablaufs	4A/161

Streams	Eigenschaften	• nicht in DrRacket (HtDP-TL), sondern nur in Racket • Variante von Listen, ziemlich ähnlich • Einziger Unterschied: Elemente müssen nicht unbedingt physisch existieren	8/122
	Funktionen	• (stream-cons x str): Hängt x vorne an str • (stream-first str): Gibt erstes Element von str zurück • (stream-rest str): Stream ohne erstes Element • (stream-empty? str): Abfrage ob Stream leer ist • Map, Filter und fold analog zu Listen	8/129

Struct	Zweck	• Zusammenfassung von Elementen → Vergleichbar mit Klasse mit public Attributen aber ohne Methoden	4B/94
	Syntax	• (define-struct my-struct (attribut1 attribut2)) → Attribute werden auch 'Felder' genannt	4B/99
	'Konstruktor'	• (define my-struct-name (make-my-struct wert1 wert2))	4B/101
	Attributzugriff	• .. (my-struct-attribut1 my-struct-name) ..	4B/104
	mystruct?	• #t falls Element von Typ my-struct	4B/112
	Funktionen als Attribute	• (define-struct functions (fct1 fct2)) • z.B.: (define a (make-functions + *)) → fct1 von a ist Summe, fct2 von a ist Multiplikation	4C/11

Symbole	Syntax	• (define my-symbol 'helloworld)	4A/83
	Verwendung	• Symbole stehen nur für sich selbst → keine Bedeutung für Racket	4A/83
	Gleichheit	• (if (symbol=? my-symbol1 my-symbol2) (falls #t) (falls #f))	4A/84

Zahlen	Exakte Zahlen	• ganzzahlig (z.B.: 5) • rational (z.B.: 3/5 → wird auch als Bruch gespeichert)	4A/61
	Nichtexakte Zahlen	• irrationale Zahlen (z.B.: (sqrt 2)	4A/62
	Komplexe Zahlen	• komplexe Zahlen • Realteil + Imaginärteil (i nach Imaginärteil)	4A/63

		•	
		•	
		•	
		•	
		•	