Funktionale und objektorientierte Programmierkonzepte

Zusammenfassung

Fabian Damken

9. November 2021

TECHNISCHE UNIVERSITÄT DARMSTADT

Inhaltsverzeichnis

1 HtDP-TL

Die HtDP-TL (How to Design Programs-Teaching Language) ist ein Teil der Sprache Racket, welche einen Dialekt von, welches eine implementierung von Lisp (List Processing) darstellt. Die Syntax der HtDP-TL ist äquivalent zu der Syntax von Racket, da HtDP-TL ausschließlich den Funktionsumfang von Racket einschränkt (das heißt es stehen weniger Funktionen zur Verfügung). Im folgenden werden Racket und HtDP-TL äquivalent verwendet, wobei ausschließlich HtDP-TL gemeint ist.

1.1 Syntax

Da Racket eine vollfunktionale Sprache ist, existieren keine Variablen, sondern der gesamte Quellcode besteht nur aus Funktionsaufrufen.

1.1.1 Funktionsaufrufe

Eine Funktion wird in einem Listenähnlichen konstrukt aufgerufen, wobei der Name der aufzurufenden Funktion am Beginn der Liste steht. Alle Parameter folgen durch ein Leerzeichen getrennt. Der Funktionsaufruf wird mit einer schließenden Klammer abgeschlossen: `(<Funktionsname> <Parameter 1> <Parameter 2> ... <Parameter n>)`

Fun Fact: Es ist irrelevant, ob runde, eckige oder geschweifte Klammern verwendet werden. Eine Variation der Klammern ist nur gut für die Lesbarkeit.

Beispiel `'+'` stellt die Funktion zum Addieren von beliebig vielen Zahlen dar. Möchte man nun die Zahlen 24, 7 und 11 addieren, so wird dies wie folgt geschrieben: `(+ 24 7 11)`

1.1.2 Primitive Datentypen

Beschreibung

String Stellt eine folge von Zeichen dar.

Boolean Stellt einen Wahrheitswert (Wahr/Falsch) dar.

Symbol Stellt ein sogenanntes Symbol dar. Ein Symbol ist prinzipiell ein String, welcher keine Leerzeichen und Klammer (mit Ausnahme von spitzen Klammer) enthalten kann und dynamisch in jeden anderen Typ umgewandelt wird. Konkret bedeuted dies, ein Symbol mit dem Inhalt der syntaktisch korrekt zu einem anderen Datentyp passt (beispielsweise eine Zahl) ist gleichzeitig dieser Datentyp.

Number Stellt eine Zahl dar. Außerdem können alle Zahlen sowohl in Binär, Oktal, Dezimal und Hexadezimal eingelesen werden.

Integers Stellen ganze Zahlen dar.

Floating Point Numbers Stellen Fließkommazahlen dar. Diese können, aufgrund der internen Darstellung, Darstellungsfehler enthalten.

Factions Stellen Brüche dar. Racket beherrscht bruchrechnung, das heißt die Berechnung mit Brüchen ist relativ exakt.

Complex Numbers Stellen komplexe Zahlen mit Real- und Imaginäranteil dar.

Format

String Wird in doppelte Anführungszeich eingefasst, wobei doppelte Anführungszeichen in dem String mit einem Backslash (`\`) maskiert (escaped) werden müssen (d.h. dem Zeichen wird ein Backslash voran gestellt). Soll ein Backslash in dem String enthalten sein, so muss dieses ebenfalls escaped werden (d.h. es müssen zwei Backslashes geschrieben werden).
Beispiel: `"Hello, World!"`, `"\"\\"`

Boolean Ein Boolean hat nur zwei mögliche Werte, welche allerdings unterschiedlich dargestellt werden können: `true` $\iff$ `#t`, `false` $\iff$ `#f`

Symbol Ein Symbol wird eingeleitet durch ein einfaches Anführungszeichen (`'`), gefolgt von seinem Inhalt. Das Symbol wird mit dem ersten Zeichen abgeschlossen, welches ein Symbol nicht enthalten kann (beispielsweise ein Leerzeichen). Da Symbole dynamisch in alle Typen umgewandelt werden, gilt beispielsweise `'123` $\iff$ `123` und `'"HelloWorld"` $\iff$ `HelloWorld`. Falls symbole einen String darstellen (das heißt der Inhalt ist in doppelten Anführungszeichen geschrieben), kann ein Symbol alle Zeichen beinhalten (Beispiel: `'"Hello, World!"` $\iff$ `"Hello, World!"`).

Number

Integers Die darzustellende Zahl wird einfach in den Quellcode geschrieben. Vor der Zahl kann ein Minus als Vorzeichen stehen.
Beispiel: `42`

Floating Point Numbers Die Nachkommastellen werden durch einen Punkt von dem ganzzahligen Teil der Zahl getrennt. Vor der Zahl kann ein Minus als Vorzeichen stehen.
Beispiel: `123.45`
Sind keine Nachkommastellen vorhanden, wird der Punkt weg gelassen und die Zahl stellt eine Ganzzahl dar.

Factions Der ganzzahlige Zähler wird mit einem Schrägstrich von dem ganzzahligen nenner getrennt. Dabei darf zwischen dem Schrägstrich und den Zahlen kein Leerzeichen stehen. Vor der Zahl kann ein Minus als Vorzeichen stehen.
Beispiel: `12/34`

Complex Numbers Komplexe Zahlen werden dargestellt durch den Realanteil, gefolgt von dem Imaginäranteil und einem `i`. Dabei muss zwischen dem Real- und Imaginäranteil ein Plus oder ein

Minus stehen. Ebenfalls kann der Realanteil positiv oder negativ sein. Zwischen allen Zeichen darf kein Leerzeichen stehen.
Beispiel: `12+34i`, `-12-34i`

1.1.3 Komplexe Datentypen

Listen

Listen sind in Racket als sogenannte gelinkte Liste implementiert, das heißt jeder Listeneintrag hält das ihm zugehörige Element und einen Zeiger auf den nächsten Listeneintrag, welcher wiederrum einen Zeiger auf das nächste Listenelement hält und so weiter. Das letzte Listenelement hält einen Zeiger auf die leere Liste, um die gesamte Liste abzuschließen. Das Bild ?? visualisiert dieses Verhalten anhand eines Beispiels.

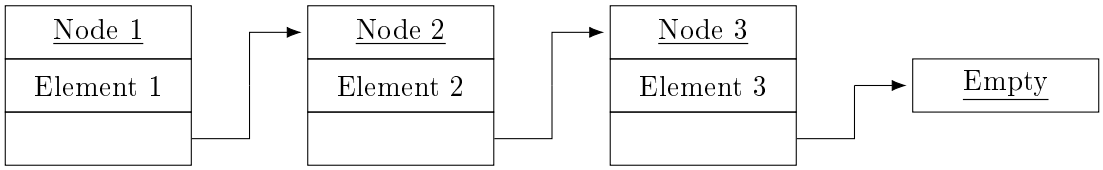

Abbildung 1.1: Gelinkte Liste

Die folgenden Abschnitte fassen kurz die primitiven Operationen auf Listen in Racket zusammen.

Die leere Liste Die leere Liste ist als `empty` vorhanden.

Hinzufügen eines Elementes an eine Liste

Abstraktes Konstrukt `(cons <element> <list>)`

Vertrag `cons :: any list → list`

Beispiel `(cons 'a (cons 'b lst))`

Warnung Dies fügt das Element nicht wirklich zu der Liste hinzu, sondern erstellt eine neue Liste!

Erstellen einer Liste mit bekannten Elementen Eine neue Liste kann mittels `cons` erstellt werden. Beispiel: `(cons 'a (cons 'b empty))`
Da dies relativ viel Schreibarbeit ist, kann auch folgendes Konstrukt mit dem gleichen Effekt verwendet werden:

Abstraktes Konstrukt `(list [element]...)`

Vertrag `list :: any...`

Alternativ ist folgendes Konstrukt verfügbar:

Abstraktes Konstrukt `'([element]...)`

Beispiel `'(a b "Hello, World!")` → `(list 'a 'b '"Hello, World!")`

Warnung Jedes Element der Liste wird zu einem Symbol, bzw. zu dem jeweiligen Typen!

Prüfen, ob X eine Liste ist

Abstraktes Konstrukt `(list? <arg>)`

Vetrag `list? :: any → boolean`

Prüfen, ob X leer ist

Abstraktes Konstrukt `(empty? <arg>)`

Vertrag `empty? :: any → boolean`

1.1.4 Kontrollstrukturen

Notation

Für jede Kontrollstruktur wird der Name, das Abstrakte Konstrukt, der Vertrag der Kontrollstruktur, eine Beschreibung und ein Beispiel angegeben.

If-Statement

Name If-Statement

Abstraktes Konstrukt `(if <condition> <then> <else>)`

Vertrag `if :: boolean any any → any`

Beschreibung Liefert `<condition>` `true`, so wird das Ergebnis von `<then>` zurück gegeben. Liefert `<condition>` `false`, so wird das Ergebnis von `<else>` zurück gegeben.

Beispiel `(if (> x y) x y)` Dies stellt eine implementierung der Max-Funktion dar, welche die größere Zahl von zweien liefert.

Cond-Statement

Name Cond-Statement

Abstraktes Konstrukt `(cond (<condition> <statement>)... [(else <else-statement>)])`

Vertrag `cond :: (boolean any)... (else any)? → any`

Beschreibung Liefert `<condition>` eines condition-statement-tupels `true`, so wird das Ergebnis des dazugehörigen `<statement>` zurück geliefert. Die Paare werden dabei von links nach rechts abgefragt, wobei das Ergebnis des erste Paares dessen `<condition>` `true` liefert, zurück gegeben wird. Existiert kein solches Paar, so wird das Ergebnis von `<else-statement>` zurück gegeben. Da dieses optional ist wird, sollte kein else-statement existieren, ein Fehler ausgelöst.

Beispiel `(cond [(> x 2) 'x_greater_2] [(> x 1) 'x_greater_1] [else 'x_smaller_1])` Dies stellt eine Funktion da, die für x mit `x > 2` das Symbol `'x_greater_2`, für x mit `x > 1` das Symbol `'x_greater_1` und ansonsten das Symbol `'x_smaller_1` liefert. Hier ist auch zu sehen, wie die Variation von Klammertypen dazu eingesetzt werden kann, den Code übersichtlicher zu gestalten.

1.1.5 Überblick über die Funktionen

Name	Vertrag	Beschreibung
Addition	`+ :: number... → number`	Summiert die gegebenen Parameter auf.
Subtraktion	`- :: number number → number`	Subtrahiert den zweiten von dem ersten Parameter.
Multiplikation	`* :: number... → number`	Multipliziert die gegebenen Parameter miteinander.
Quadrat	`sqr :: number → number`	Errechnet das Quadrat der gegebenen Zahl.
Quadratwurzel	`sqrt :: number → number`	Errechnet die Quadratwurzel der gegebenen Zahl.
Potenz	`expt :: number number → number`	Berechnet die erste Zahl hoch den zweiten.
Null	`zero? :: number → boolean`	Prüft, ob die gegebene Zahl gleich Null ist.
Gerade	`even? :: number → boolean`	Prüft, ob die gegebene Zahl gerade ist.
Ungerade	`odd? :: number → boolean`	Prüft, ob die gegebene Zahl ungerade ist.
Division	`/ :: number number → number`	Dividiert den ersten durch den zweiten Parameter.
Minimaler Wert	`min :: number... → number`	Gibt das kleinste Element zurück.
Maximaler Wert	`max :: number... → number`	Gibt das gröSSte Element zurück.
Absoluter Wert	`abs :: number → number`	Gibt den absoluten Wert der gegebenen Zahl zurück.
Modulo	`modulo :: number number → number`	Errechnet den Divisionsrest.
GröSSer	`> :: number number → boolean`	Prüft, ob die erste Zahl gröSSer als die zweite ist.
Kleiner	`> :: number number → boolean`	Prüft, ob die erste Zahl kleiner als die zweite ist.
GröSSer-Gleich	`> :: number number → boolean`	Prüft, ob die erste Zahl gröSSer-gleich die zweite ist.
Kleiner-Gleich	`> :: number number → boolean`	Prüft, ob die erste Zahl kleiner-gleich die zweite ist.
Gleichheit	`equal? :: any any → boolean`	Prüft, ob die gegebenen Parameter gleich sind.
Erstes Element	`first :: (listof X) → X`	Gibt das erste Element der Liste zurück.
Zweites Element	`second :: (listof X) → X`	Gibt das zweite Element der Liste zurück.
⋮	⋮	⋮
Achtes Element	`eighth :: (listof X) → X`	Gibt das achte Element der Liste zurück.
Rest der Liste	`rest :: (listof X) → (listof X)`	Gibt den Rest der Liste zurück.
Länge der Liste	`length :: (listof X) → number`	Gibt die Länge der Liste zurück.
If-Statement		Siehe 1.1.4.
Cond-Statement		Siehe 1.1.4.

Tabelle 1.1: Racket: Funktionsüberblick 1

Name	Vertrag	Beschreibung
Datenmapping	`map :: (X → Y) (listof X) → (listof Y)`	Führt die übergebene Funktion für jedes Element der Liste aus und erstellt eine neue Liste, welche die zurückgegebenen Werte enthält.
Datenfilterung	`filter :: (X → boolean) (listof X) → (listof X)`	Erstellt eine neue Liste, welche alle Werte aus der alten Liste enthält, für die das übergebene Prädikat `true` geliefert hat.
Akkumulation der Daten	`foldl :: (X Y → Y) Y (listof X) → Y`	Fasst die Daten mit Hilfe der übergebenen Funktion von links zusammen.
Leere Liste	`empty? :: (listof X) → boolean`	Prüft, ob die Liste leer ist.
Definition von Konstanten	`(define <name> <value>)`	Definiert eine Konstante mit dem gegebenen Namen und dem gegebenen Wert. Dieser kann ein Ausdruck sein.
Definition von Funktionen	`(define (<name> <arg1> <arg2> ... <argN>) <expr>)`	Definiert eine Funktion mit dem gegebenen Namen und den gegebenen Parametern. In dem Ausdruck `<expr>` können diese genutzt werden.
Konjunktion	`and :: boolean... → boolean`	Stellt ein logisches UND dar.
Disjunktion	`or :: boolean... → boolean`	Stellt ein logisches ODER dar.
Negation	`not :: boolean → boolean`	Stellt ein logisches NICHT dar.
Test auf Gleichheit	`(check-expect <actual: any> <expected: any>)`	Testet, ob `<actual>` dem Wert `<expected>` entspricht.
Test auf Ähnlichkeit	`(check-within <actual: number> <expected: number> <delta: number>)`	Testet, ob `<actual>` und `<expected>` sich um maximal `<delta>` unterscheiden.
Feld von Struktur selektieren `(<struct>-<field> <value>)`	Gibt den Wert des Feldes `<field>` der Struktur `<struct>` und dem Wert `<value>` zurück.	
Lambda	`(lambda (<param1> <param2> ... <paramN>) <expr>)`	Erstellt eine anonyme Funktion mit den Parametern `<param1>`, `<param2>`, ..., `<paramN>`, die den Ausdruck `<expr>` darstellt.
Lexikalischer Scope	`(local (<define>...) <expr>)`	Erstellt einen neuen lexikalischen Scope, in dessen Audruck `<expr>` alle Definition aus dem `define`-Block verfügbar sind.

Tabelle 1.2: Racket: Funktionsüberblick 2

2 Java

2.0.1 Streams und Lambdas

Java 8 Streams (`java.util.stream`) bringen funktionale Programmierung in Java ein. Durch Lambdas wird diese kürzer.

Die besten Wege, um an einen `java.util.stream.Stream<T>` zu kommen, sind die folgenden:

- Mittels einer `Collection` (`List` oder `Set`): `java.util.Collection#stream()`
- Mittels eines Arrays: `java.util.Arrays#stream(T[])`

2.0.2 Generics

Merksatz

Greift man lesend auf generische Typen zu, so sollte man `extends` verwenden.
Greift man schreibend auf generische Typen zu, so sollte man `super` verwenden.

2.0.3 JUnit

Generell müssen alle Methoden in JUnit-Testklassen `public` sein mit dem Rückgabetyp `void`. Alle Annotation müssen an solchen Methoden stehen. Ferner kann jede Methode jeden beliebigen Fehler deklarieren (`throws`). Es kann Sinnvoll sein, einfach `Exception` zu werfen.

Annotationen

`Assert`

2.0.4 Überblick über die Standardbibliothek

`Collection<E>`

Alle funktionalen Interfaces sind in 2.0.4 zusammengefasst.

`Stream<T>`

Alle funktionalen Interfaces sind in 2.0.4 zusammengefasst.

Annotation	Parameter	Beschreibung
`@BeforeClass`	-	Die Methode wird vor dem initialisieren der Klasse ausgeführt. Die Methode muss `static` sein!
`@AfterClass`	-	Die Methode wird nach dem Ausführen der gesamten Klasse ausgeführt. Die Methode muss `static` sein!
`@Before`	-	Die Methode wird vor jedem Test ausgeführt.
`@After`	-	Die Methode wird nach jedem Test ausgeführt.
`@Test`	`expected: Class<? extends Exception>` - Der erwartete Fehler. `timeout: long` - Eine Zeit, in Millisekunden, nach der der Test abgebrochen werden soll.	Markiert eine Methode als Test.

Tabelle 2.1: Java: JUnit: Annotationen

Methode	Beschreibung
`void assertTrue(boolean actual)`	Schlägt fehl, wenn `actual` `false` ist.
`void assertFalse(boolean actual)`	Schlägt fehl, wenn `actual` `true` ist.
`<T> void assertArrayEquals(T[] expected, T[] actual)`	Schlägt fehl, wenn `actual` sich von `expected` unterscheidet.
`void assertArrayEquals(double[] expected, double[] actual, double delta)`	Schlägt fehl, wenn `actual` sich mehr als `delta` von `expected` unterscheidet.
`<T> void assertEquals(T expected, T actual)`	Schlägt fehl, wenn `actual` sich von `expected` unterscheidet.
`void assertEquals(double expected, double actual, double delta)`	Schlägt fehl, wenn `actual` sich mehr als `delta` von `expected` unterscheidet.
`void fail()`	Lässt den Test fehlschlagen.

Tabelle 2.2: Java: JUnit: `Assert`

Methode	Beschreibung	List	Set
`add(E element)`	Fügt das gegebene Element hinzu.	×	×
`addAll(Collection<? extends E> elements)`	Fügt alle gegebenen Elemente hinzu.	×	×
`clear()`	Leert die Liste/Menge.	×	×
`boolean contains(Object o)`	Prüft, ob das gegebene Objekt vorhanden ist.	×	×
`boolean containsAll(Collection<?> c)`	Prüft, ob alle gegebenen Objekte vorhanden sind.	×	×
`boolean isEmpty()`	Prüft, ob die Liste/Menge leer ist.	×	×
`remove(Object o)`	Entfernt das gegebene Objekt.	×	×
`removeAll(Collection<?> c)`	Entfernt alle gegebenen Objekte.	×	×
`Stream<E> stream()`	Wandelt die Liste/Menge in einen Stream um.	×	×
`<T> T[] toArray(T[] a)`	Wandelt die Liste/Menge in ein Array des gegebenen Types um.	×	×
`add(int index, E element)`	Fügt das gegebene Element in die Liste an der gegebenen Position ein.	×	
`addAll(int index, Collection<? extends E> elements)`	Fügt alle gegebenen Elemente in die Liste an der gegebenen Position ein.	×	
`E get(int index)`	Gibt das Element an der gegebenen Position zurück.	×	
`int indexOf(Object o)`	Gibt den Index des ersten Eintrages des gegebenen Objektes zurück.	×	
`int lastIndexOf(Object o)`	Gibt den Index des letzten Eintrages des gegebenen Objektes zurück.	×	
`remove(int index)`	Entfernt das Element an der gegebenen Position.	×	
`set(int index, E element)`	Setzt das Element an der gegebenen Position.	×	

Tabelle 2.3: Java: Funktionsübersicht: `Collection<E>`

Methode	Beschreibung
`boolean allMatch(Predicate<? super T> pred)`	Prüft, ob das übergebene Prädikat für alle Elemente `true` liefert.
`boolean anyMatch(Predicate<? super T> pred)`	Prüft, ob das übergebene Prädikat für mindestens ein Elemente `true` liefert.
`boolean noneMatch(Predicate<? super T>)`	Prüft, ob das übergebene Prädikat für kein Element `true` liegert.
`<R, A> R collect(Collector<? super T, A, R>)`	Führt den übergebenen `Collector` auf der Liste aus. Hierbei wird meist die Klasse `Collectors` verwendet, welche einige Standard-Collectors zur verfügung stellt. Siehe 2.0.4.
`long count()`	Zählt die im Stream enthaltenen Elemente.
`Optional<T> findAny()`	Gibt eines der Elemente zurück. Siehe 2.0.4.
`Optional<T> findFirst()`	Gibt das erste Element zurück. Siehe 2.0.4.
`void forEach(Consumer<? super T> cons`	Führt den übergebenen `Consumer` auf jedem Element aus.
`Optional<T> max()`	Gibt das gröSSte Element zurück. Siehe 2.0.4.
`Optional<T> max(Comparator<? super T>)`	Gibt das gröSSte Element auf Basis des übergebenen `Comparator`s zurück. Siehe 2.0.4.
`Optional<T> min()`	Gibt das kleine Element zurück. Siehe 2.0.4.
`Optional<T> min(Comparator<? super T>)`	Gibt das kleinste Element auf Basis des übergebenen `Comparator`s zurück. Siehe 2.0.4.
`<U> reduce(U init, BiFunction<U, ? super T, U> accumulator, BinaryOperator<U> combiner)`	Führt eine reduce-Funktion auf dem Stream aus. Dies entspricht dem `fold` in Racket, wobei der `combiner` zwei unabhängige Ergebnisse zusammen führt (Sichwort: parallele Programmierung).
`Stream<T> distinct()`	Erstellt einen Stream mit einzigartigen Elementen zurück.
`Stream<T> filter(Predicate<? super T> pred)`	Erstellt einen neuen Stream, der alle Elemente enthält, für die das übergebene Prädikat `true` liefert.
`Stream<T> limit(long limit)`	Erstellt einen neuen Stream mit maximal `limit` Elementen.
`<R> Stream<R> map(Function<? super T, ? extends R> mapper)`	Erstellt einen neuen Stream, wobei für jedes Element die übergebene Funktion ausgeführt wird und das Ergebnis in den neuen Stream inkludiert wird.
`XXXStream mapToXXX(ToXXXFunction<? super T>) mapper`	Das selbe wir `map(…)`, nur dass hierbei auf einen primitiven Typ (`XXX`) projeziert wird.
`Stream<T> sorted()`	Sortiert den Stream.
`Stream<T> sorted(Comparator<? super T> comp)`	Sortiert den Stream auf Basis des übergebenen `Comparator`s.

Tabelle 2.4: Java: Funktionsübersicht: `Stream<E>`

Methode	Beschreibung
`static Collector<CharSequence, ?, String> joining(CharSequence delimiter)`	Erstellt einen `Collector`, der alle Elemente zu einem String zusammenführt, von `delimiter` getrennt.
`static <T> Collector<T, ?, List<T>> toList()`	Erstellt einen `Collector`, der alle Elemente in eine Liste einfügt.
`static <T> Collector<T, ?, Set<T>> toSet()`	Erstellt einen `Collector`, der alle Elemente in ein Set einfügt.

Tabelle 2.5: Java: Funktionsübersicht: `Collectors`

Methode	Beschreibung
`T get()`	Gibt den Wert zurück oder wirft einen Fehler, falls keiner existiert.
`boolean isPresent()`	Prüft, ob ein Wert vorhanden ist.
`T orElse(T other)`	Gibt den gespeicherten Wert zurück oder, falls keiner vorhanden ist, den übergebenen.

Tabelle 2.6: Java: Funktionsübersicht: `Optional<T>`

`Collectors`

`Optional<T>`

Funktionale Interfaces

Interface	Methode
`java.util.function.Consumer<T>`	`void accept(T)`
`java.util.function.Function<T, R>`	`R apply(T)`
`java.util.function.Predicate<T>`	`boolean test(T)`
`java.util.function.Supplier<T>`	`T get()`
`java.util.function.UnaryOperator<T, T>`	`T apply(T)`
`java.util.function.BiConsumer<T, U>`	`void accept(T, U)`
`java.util.function.BiFunction<T, R, U>`	`R apply(T, U)`
`java.util.function.BinaryOperator<T>`	`T apply(T, T)`
`java.util.function.BiPredicate<T, U>`	`boolean test(T, U)`

Tabelle 2.7: Java: Funktionsübersicht: Funktionale Interfaces