TECHNISCHE UNIVERSITÄT DARMSTADT

Kapitel 04d:
Funktionales Programmieren: Diskussion

Karsten Weihe

TECHNISCHE UNIVERSITÄT DARMSTADT

Was ist nun typisch funktional?

Wir unterbrechen kurz unseren Rundgang durch Racket für eine allgemeine Diskussion und werden uns nach dieser Diskussion nur noch ein paar wenige Konzepte von Racket ansehen, die unmittelbar mit dieser Diskussion zu tun haben.

Was ist nun „typisch funktional“?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die primären Entitäten
  - In Java hingegen Klassen und ihre Methoden
- Deklaratives Programmieren
  - Referentielle Transparenz
- Rekursion ist die zentrale Kontrollstruktur
- Funktionen höherer Ordnung (lambda)
  - In Java „nur“ eine nachträglich eingebaute Variation von Functional Interfaces
- Generische Funktionen
- Man muss Typen nicht hinschreiben

Der grundlegende Unterschied zwischen objektorientierten und funktionalen Sprachen ist erst einmal, was die zentralen Bausteine in einer Sprache sind. In Java sind das bekanntlich die Klassen, und anstelle von Funktionen haben wir Methoden, die Bestandteile von Klassen sind. In Racket hingegen sind Funktionen die primäre Art von Baustein.

Was ist nun „typisch funktional“?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die primären Entitäten
  - In Java hingegen Klassen und ihre Methoden
- Deklaratives Programmieren
  - Referentielle Transparenz
- Rekursion ist die zentrale Kontrollstruktur
- Funktionen höherer Ordnung (lambda)
  - In Java „nur“ eine nachträglich eingebaute Variation von Functional Interfaces
- Generische Funktionen
- Man muss Typen nicht hinschreiben

Funktionale Sprachen sind deklarative Sprachen, das heißt, man schreibt nicht hin, wie der Computer vorgehen soll, sondern einfach nur, wie das zu berechnende Ergebnis definiert ist. Zeitliche Abläufe sind nicht Teil des Denkmodells.

Konsequenz daraus ist das Prinzip der referentiellen Transparenz: Ein Ausdruck muss überall, wo er vorkommt, denselben Wert haben, denn ohne Zeit kann der Wert eines Ausdrucks auch keinen zeitabhängigen Wert haben.

Was ist nun „typisch funktional“?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die primären Entitäten
  - In Java hingegen Klassen und ihre Methoden
- Deklaratives Programmieren
  - Referentielle Transparenz
- Rekursion ist die zentrale Kontrollstruktur
- Funktionen höherer Ordnung (lambda)
  - In Java „nur“ eine nachträglich eingebaute Variation von Functional Interfaces
- Generische Funktionen
- Man muss Typen nicht hinschreiben

Schleifen sind inkompatibel zum deklarativen Programmierstil. Daher basieren funktionale Sprachen – wie auch andere Arten von deklarativen Sprachen – auf Rekursion.

Was ist nun „typisch funktional“?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die primären Entitäten
  - In Java hingegen Klassen und ihre Methoden
- Deklaratives Programmieren
  - Referentielle Transparenz
- Rekursion ist die zentrale Kontrollstruktur
- Funktionen höherer Ordnung (lambda)
  - In Java „nur“ eine nachträglich eingebaute Variation von Functional Interfaces
- Generische Funktionen
- Man muss Typen nicht hinschreiben

Funktionen höherer Ordnung sind Funktionen, die Parameter von Funktionstypen oder Rückgabe von einem Funktionstyp haben. In typischen funktionalen Srachen ist dieses Konzept in der Sprache eingebaut, in Java wurde es nach diesem Vorbild mit Hilfe von Functional Interfaces nachgebildet.

Was ist nun „typisch funktional“?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die primären Entitäten
  - In Java hingegen Klassen und ihre Methoden
- Deklaratives Programmieren
  - Referentielle Transparenz
- Rekursion ist die zentrale Kontrollstruktur
- Funktionen höherer Ordnung (lambda)
  - In Java „nur“ eine nachträglich eingebaute Variation von Functional Interfaces
- Generische Funktionen
- Man muss Typen nicht hinschreiben

Damit sind Funktionen wie fold, filter und map gemeint, die auf beliebigen Typen X und Y arbeiten können (es kann natürlich mehr als zwei Typen geben). Beim Aufruf einer solchen generischen Funktion muss nur gewährleistet sein, dass die Parameter von passenden Typen sind.

***Vorgriff*: In Kapitel 06 werden wir sehen, dass auch dieses Konzept in Java übernommen worden ist.**

Was ist nun „typisch funktional“?

TECHNISCHE UNIVERSITÄT DARMSTADT

- Funktionen sind die primären Entitäten
  - In Java hingegen Klassen und ihre Methoden
- Deklaratives Programmieren
  - Referentielle Transparenz
- Rekursion ist die zentrale Kontrollstruktur
- Funktionen höherer Ordnung (lambda)
  - In Java „nur“ eine nachträglich eingebaute Variation von Functional Interfaces
- Generische Funktionen
- Man muss Typen nicht hinschreiben

Dieser Punkt ist uns immer wieder ins Auge gesprungen, aber dieser Satz kann so kommentarlos nicht einfach stehenbleiben. Auf den nächsten beiden Folien werden wir diesen Satz daher genauer diskutieren.

... Typen nicht hinschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

- In Racket wird erst *zur Laufzeit* geprüft ob die Typen
  - der Operanden zum Operator passen
  - der Parameter einer vordefinierten Funktion zur Funktion passen
  - Begriff: *dynamische* Typisierung
- In Java prüft *der Compiler* die Passung von Typen
  - Begriff: *statische* Typisierung

Zunächst einmal bleiben wir auf dieser Folie bei Java und Racket, bevor wir uns auf der nächsten Folie von diesen beiden konkreten Sprachen etwas lösen.

... Typen nicht hinschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

- In Racket wird erst *zur Laufzeit* geprüft ob die Typen
  - der Operanden zum Operator passen
  - der Parameter einer vordefinierten Funktion zur Funktion passen
  - Begriff: *dynamische* Typisierung
- In Java prüft *der Compiler* die Passung von Typen
  - Begriff: *statische* Typisierung

In Racket schreiben wir nirgendwo die Typen der Werte hin, mit denen wir arbeiten. Damit ist es völlig unmöglich, die Kompatibilität der Typen vor dem Laufenlassen des Programms zu prüfen. Inkompatibilitäten können auftreten bei typspezifischen Operationen und bei vordefinierten Funktionen. Überall sonst werden die Werte einfach blindlings weitergereicht, ohne dass irgendwelche Fehler auftreten könnten.

... Typen nicht hinschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

- In Racket wird erst *zur Laufzeit* geprüft ob die Typen
  - der Operanden zum Operator passen
  - der Parameter einer vordefinierten Funktion zur Funktion passen
  - Begriff: *dynamische* Typisierung
- In Java prüft *der Compiler* die Passung von Typen
  - Begriff: *statische* Typisierung

In Java hingegen muss man überall den Typ hinschreiben. Der Compiler kann auf dieser Basis schon vorab prüfen, ob die Typen miteinander und mit allen Operationen kompatibel sind.

Wir haben gesehen, dass es da in Java eine Lücke gibt: Bei Downcast kann Typkompatibilität unmöglich schon beim Übersetzen geprüft werden, das passiert erst zur Laufzeit, und wenn der Typ nicht passt, bricht das Laufzeitsystem die Ausführung des Programms mit einer Fehlermeldung ab. Beziehungsweise wir können den Typ auch selbst mit Operator instanceof abprüfen.

... Typen nicht hinschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt aber auch funktionale Sprachen mit *statischer* Typisierung
  - Zum Beispiel ML
- Dennoch muss man Typen in ML u.ä. nicht hinschreiben
- Lösung des Rätsels: Programm ist nur korrekt wenn das System den Typ jedes Wertes unzweideutig aus dem Kontext erschließen kann
  - So wie bei der Kurzform von Lambda-Ausdrücken in Java
  - Typinferenz

Der Unterschied zwischen statischer und dynamischer Typisierung ist aber nicht der entscheidende Unterschied, denn es gibt auch funktionale Sprachen mit statischer Typisierung. Es bleibt aber bei der Aussage: Typisch funktional ist, dass man die Typen nicht hinschreiben muss.

... Typen nicht hinschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt aber auch funktionale Sprachen mit *statischer* Typisierung
  - Zum Beispiel ML
- Dennoch muss man Typen in ML u.ä. nicht hinschreiben
- Lösung des Rätsels: Programm ist nur korrekt wenn das System den Typ jedes Wertes unzweideutig aus dem Kontext erschließen kann
  - So wie bei der Kurzform von Lambda-Ausdrücken in Java
  - Typinferenz

Der Punkt ist: In diversen statisch typisierten funktionalen Sprachen muss man die Typen deshalb nicht hinschreiben, weil das System selbst in der Lage ist, die Typen der einzelnen Werte aus dem Programm heraus für sich abzuleiten und auf dieser Basis vorab eine Typprüfung zu machen.

... Typen nicht hinschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt aber auch funktionale Sprachen mit *statischer* Typisierung
  - Zum Beispiel ML
- Dennoch muss man Typen in ML u.ä. nicht hinschreiben
- Lösung des Rätsels: Programm ist nur korrekt wenn das System den Typ jedes Wertes unzweideutig aus dem Kontext erschließen kann
  - So wie bei der Kurzform von Lambda-Ausdrücken in Java
  - Typinferenz

Ein Beispiel dafür gibt es auch in Java, und es ist kein Zufall, dass dieses Beispiel bei einem funktionalen Konstrukt auftritt: Wenn der Compiler bei einem Lambda-Ausdruck die Typen der Parameter selbst ableiten kann, dann muss man sie nicht hinschreiben.

... Typen nicht hinschreiben

TECHNISCHE UNIVERSITÄT DARMSTADT

- Es gibt aber auch funktionale Sprachen mit *statischer* Typisierung
  - Zum Beispiel ML
- Dennoch muss man Typen in ML u.ä. nicht hinschreiben
- Lösung des Rätsels: Programm ist nur korrekt wenn das System den Typ jedes Wertes unzweideutig aus dem Kontext erschließen kann
  - So wie bei der Kurzform von Lambda-Ausdrücken in Java
  - Typinferenz

Der Fachbegriff dafür, dass die Typen von Variablen und Konstanten automatisch aus dem Kontext heraus bestimmt werden beziehungsweise dass es eine Fehlermeldung gibt, wenn die Typen nicht miteinander und mit dem Kontext zusammenpassen, lautet *Typinferenz*.

Diskussion statische vs. dynamische Typisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Pro statisch:

- **Mehr Fehlersicherheit durch Compilerprüfungen**
- **Geringere Gesamtlaufzeiten da keine Typprüfungen zur Laufzeit**

Pro dynamisch:

- **Quelltext ist in der Regel kürzer und übersichtlicher**
- **Wird oft behauptet: schnellere Entwicklung von ersten Versionen bzw. Prototypen**

Wenn beides seit Jahrzehnten in Programmiersprachen realisiert ist, statische und dynamische Typisierung, dann müssen ja beide Konzepte jeweils so einiges für sich haben.

Diskussion statische vs. dynamische Typisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Pro statisch:

- **Mehr Fehlersicherheit durch Compilerprüfungen**
- **Geringere Gesamtlaufzeiten da keine Typprüfungen zur Laufzeit**

Pro dynamisch:

- **Quelltext ist in der Regel kürzer und übersichtlicher**
- **Wird oft behauptet: schnellere Entwicklung von ersten Versionen bzw. Prototypen**

Fehler, die schon der Compiler findet, können nicht erst im laufenden Betrieb auftreten, das ist ein gewaltiger Vorteil von statischer Typsicherheit.

Diskussion statische vs. dynamische Typisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Pro statisch:

- **Mehr Fehlersicherheit durch Compilerprüfungen**
- **Geringere Gesamtlaufzeiten da keine Typprüfungen zur Laufzeit**

Pro dynamisch:

- **Quelltext ist in der Regel kürzer und übersichtlicher**
- **Wird oft behauptet: schnellere Entwicklung von ersten Versionen bzw. Prototypen**

Ein weiterer Punkt ist, dass die zusätzliche Laufzeit für die Typprüfungen wegfällt. In manchen Anwendungen ist der Unterschied spürbar, in anderen nicht.

Diskussion statische vs. dynamische Typisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Pro statisch:

- **Mehr Fehlersicherheit durch Compilerprüfungen**
- **Geringere Gesamtlaufzeiten da keine Typprüfungen zur Laufzeit**

Pro dynamisch:

- **Quelltext ist in der Regel kürzer und übersichtlicher**
- **Wird oft behauptet: schnellere Entwicklung von ersten Versionen bzw. Prototypen**

Wenn man nicht überall die Typen hinschreiben muss, verkürzt sich das Programm natürlich entsprechend.

Diskussion statische vs. dynamische Typisierung

TECHNISCHE UNIVERSITÄT DARMSTADT

Pro statisch:

- **Mehr Fehlersicherheit durch Compilerprüfungen**
- **Geringere Gesamtlaufzeiten da keine Typprüfungen zur Laufzeit**

Pro dynamisch:

- **Quelltext ist in der Regel kürzer und übersichtlicher**
- **Wird oft behauptet: schnellere Entwicklung von ersten Versionen bzw. Prototypen**

Bei diesem Punkt bin ich mir nicht so sicher, ob er wirklich fundiert durch aussagekräftige Studien belegt ist. Plausibel wäre es zumindest. Aber es ist schon fraglich, ab welcher Größenordnung und ab welchem Entwicklungsgrad des Programms der Nachteil der fehlenden statischen Typprüfung sich negativ auf die Entwicklungszeit auswirkt, weil man laufend Fehler macht, die die statische Typprüfung gefunden hätte, aber bei dynamischer Typprüfung eine Menge Zeit bei der Fehlersuche kosten.

TECHNISCHE UNIVERSITÄT DARMSTADT

Programme aus Funktionen: Prinzip schrittweise Verfeinerung

Typisch funktional ist, die Funktionalität eines Programms in viele kleine Funktionen zu zerlegen, die sich gegenseitig aufrufen. Bewährt hat sich ein Vorgehen top-down, also zuerst die obersten Funktionen zu definieren, dann die dafür benötigten Funktionen und so weiter. Das sehen wir uns an einem einzelnen, leidlich realistischen Beispiel an.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-debt person )
    ( − ( assessed-tax person ) ( pre-payment person ) ) )
```

Das Beispiel ist: Berechnung der Steuernachforderung an eine gegebene Person nach Abgabe der Steuererklärung. Aus Zeitgründen und da es ja nur ums Prinzip geht, legen wir eine extrem einfache Steuergesetzgebung zugrunde, die mit der Situation in Deutschland natürlich allenfalls ansatzweise vergleichbar ist.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-debt person )
    ( − ( assessed-tax person ) ( pre-payment person ) ) )
```

Dieser Wert soll durch eine Funktion berechnet werden, die eine Person als Parameter hat. Diese Person soll ein Struct sein, in dem alle notwendigen Informationen als Attribute gespeichert sind. Wir machen uns in diesem Moment noch keine Gedanken darüber, welche Informationen so alles benötigt werden, das wird sich im Laufe der Entwicklung des Gesamtprogramms ergeben.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-debt person )
   ( − ( assessed-tax person ) ( pre-payment person ) ) )
```

Das ist das Prinzip der schrittweisen Verfeinerung: Wir überlegen uns bei der Realisierung der Funktion tax-debt natürlich, wie die Steuernachforderung berechnet werden soll, aber nur im Grundsätzlichen, das heißt, wir überlegen uns die Funktionen, in die die detaillierten Berechnungsschritte ausgelagert werden können.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-debt person )
    ( − ( assessed-tax person ) ( pre-payment person ) ) )
```

Um das Gesamtergebnis, also die Steuernachforderung zu berechnen, brauchen wir erst einmal die Gesamtsteuerforderung für diese Person. Die berechnen wir jetzt nicht gleich in allen Details in tax-debt, sondern rufen einfach eine Funktion auf, die das leisten soll. Diese Funktion müssen wir dann natürlich hinterher noch definieren.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-debt person )
    ( − ( assessed-tax person ) ( pre-payment person ) ) )
```

Von der Gesamtsteuerforderung müssen die Vorauszahlungen abgezogen werden, die die Person im Laufe des Steuerjahres geleistet hat. Damit ist die Funktion tax-debt auch schon fertig.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( assessed-tax person )
    ( * ( income person ) ( tax-rate ( income person ) ) ) )
```

Jetzt müssen wir die Funktionen definieren, die wir in tax-debt aufgerufen haben. Wir nehmen uns nur assessed-tax vor, das reicht für unsere Zwecke; Sie können sich pre-payment übungsweise selbst überlegen.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( assessed-tax person )
    ( * ( income person ) ( tax-rate ( income person ) ) ) )
```

Die Gesamtsteuerforderung berechnet sich aus dem Einkommen der Person und ihrem Steuersatz. Der Steuersatz ist eine Art Faktor, also Multiplikation.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( assessed-tax person )
    ( * ( income person ) ( tax-rate ( income person ) ) ) )
```

Wobei der Steuersatz in einem Steuersystem, das nicht einfach eine flat tax rate hat, wiederum durch das Einkommen der Person determiniert ist.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( income person )
    ( + ( annual-salary person )
        ( additional-income person ) ) )
```

Das Einkommen einer Person besteht in unserer einfach gehaltenen Steuerwelt aus einem Gehalt und zusätzlichen Einkünften aus Kapital und ähnlichem.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-rate inc )
    ( cond
       [ ( < inc tax-excempt-amount ) 0  ]
       [ ( < inc threshold ) 0.1 ]
       [ else 0.2 ] ) )

( define tax-excempt-amount 10000 )
( define threshold 20000 )
```

Wie gesagt, ist die hier zugrunde gelegte Steuergesetzgebung extrem einfach, so auch die Berechnung des Steuersatzes.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-rate inc )
    ( cond
       [ ( < inc tax-excempt-amount ) 0  ]
       [ ( < inc threshold ) 0.1 ]
       [ else 0.2 ] ) )

( define tax-excempt-amount 10000 )
( define threshold 20000 )
```

Bis zu einem gewissen Freibetrag sollen gar keine Steuern bezahlt werden.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-rate inc )
    ( cond
       [ ( < inc tax-excempt-amount ) 0  ]
       [ ( < inc threshold ) 0.1 ]
       [ else 0.2 ] ) )

( define tax-excempt-amount 10000 )
( define threshold 20000 )
```

Bis zu einer zweiten Grenze soll der Steuersatz 10% betragen.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-rate inc )
    ( cond
       [ ( < inc tax-excempt-amount ) 0  ]
       [ ( < inc threshold ) 0.1 ]
       [ else 0.2 ] ) )

( define tax-excempt-amount 10000 )
( define threshold 20000 )
```

Oberhalb dieser Grenze soll der Steuersatz dann 20% sein. In einem realen Steuerprogramm, zumindest im deutschen Steuerrecht, würde man diesen einfachen cond-Ausdruck durch eine Auswertung der Steuertabelle ersetzen müssen, aber das Prinzip bleibt dasselbe.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( tax-rate inc )
    ( cond
       [ ( < inc tax-excempt-amount ) 0  ]
       [ ( < inc threshold ) 0.1 ]
       [ else 0.2 ] ) )

( define tax-excempt-amount 10000 )
( define threshold 20000 )
```

Einfache Konstanten wie diese beiden Grenzen können wir natürlich sofort definieren, die brauchen nicht schrittweise verfeinert zu werden.

Wichtig für die Lesbarkeit und Wartbarkeit eines Programms ist aber immer, dass solche Konstanten tatsächlich als Konstanten mit sprechenden Namen definiert und in den einzelnen Funktionen dann mit diesen Namen verwendet werden.

Es wäre fatal, statt dessen die Zahlenwerte direkt an die Stellen zu schreiben, an denen jetzt die Namen der Konstanten stehen. Denken Sie an das Beispiel mit den vier Jahreszeiten und den vier Musketieren zurück!

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( annual-salary person )
    ( * 12 ( monthly-salary ( level person ) ) ) )

( define ( additional-income person )
    ( + ( real-estate-income person )
        ( investment-income person ) ) )
```

Diese beiden Funktionen sind für die Funktion income verwendet, aber bisher noch nicht definiert worden.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( annual-salary person )
    ( * 12 ( monthly-salary ( level person ) ) ) )

( define ( additional-income person )
    ( + ( real-estate-income person )
        ( investment-income person ) ) )
```

Wir gehen in unserer einfachen Steuerwelt davon aus, dass das Gehalt monatlich gegeben ist und ausschließlich davon abhängt, welche Stufe die Person in ihrer Firma erreicht hat.

***Nebenbemerkung*: Scheint wohl ein Land des (nicht mehr) real existierenden Sozialismus zu sein, oder ein Land, in dem nur Mitglieder des öffentlichen Dienstes Steuern bezahlen.**

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( annual-salary person )
    ( * 12 ( monthly-salary ( level person ) ) ) )

( define ( additional-income person )
    ( + ( real-estate-income person )
        ( investment-income person ) ) )
```

Wir unterscheiden in unserer einfachen Steuerwelt nur zwischen zusätzlichen Einkünften aus Immobilienbesitz und zusätzlichen Einkünften aus Kapital. Also doch nicht Sozialismus, jedenfalls kein reiner.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( monthly-salary level )
    ( cond
       [ ( = level 1 ) 3000 ]
       [ ( = level 2 ) 4000 ]
       [ ( = level 3 ) 5000 ]
       [ ( = level 4 ) 6000 ]
       [ else 7000 ] ) )
```

Beim monatlichen Gehalt nehmen wir fiktive Gehaltsstufen an, die jeweils ein ebenso fiktives Monatsgehalt ergeben.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( level person ) ( person-data-level person ) )

( define-struct person-data ( ... level ... ) )
```

Bei der Definition der Funktion level sind wir zum ersten Mal an einer Stelle, an der wir auf die Daten zu einer Person zugreifen, denn die Stufe einer Person ist einfach ein fester Wert für jede Person und muss nicht durch eine Funktion aus anderen Daten berechnet werden.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( level person ) ( person-data-level person ) )

( define-struct person-data ( ... level ... ) )
```

Wie haben uns hier dafür entschieden, einen Struct-Typ namens person-data einzurichten und ihm ein Attribut namens level mitzugeben, in dem die Stufe der Person als positive ganze Zahl gespeichert ist.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( level person ) ( person-data-level person ) )

( define-struct person-data ( ... level ... ) )
```

Auf weitere Attribute des Struct-Typs person-data kommen wir gleich zu sprechen, aber nicht auf alle. Zum Beispiel Daten zur Identifikation der Person wie etwa die Steuernummer werden wir für unser kleines Beispiel nicht benötigen und daher auch nicht besprechen.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( real-estate-income person )
    ( income-from-objects
        ( person-data-real-estate-objects person ) ) )

( define-struct person-data ( ... real-estate-objects ... ) )
```

Wir müssen noch die beiden Funktionen real-estate-income und investment-income definieren, als erste real-estate-income.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( real-estate-income person )
    ( income-from-objects
        ( person-data-real-estate-objects person ) ) )

( define-struct person-data ( ... real-estate-objects ... ) )
```

Dazu fügen wir ein weiteres Attribut namens real-estate-objects in den Struct person-data ein. Da eine Person mehr als eine Immobilie haben kann, aus der sie Einkünfte erzielt, soll real-estate-objects eine Liste von einzelnen Immobilien sein.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( real-estate-income person )
    ( income-from-objects
        ( person-data-real-estate-objects person ) ) )

( define-struct person-data ( ... real-estate-objects ... ) )
```

Die Aufgabe selbst delegieren wir wieder an eine eigene Funktion income-from-objects, die durch die Liste durchgeht.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( income-from-objects object-list )
  ( if ( empty? object-list )
     0
     ( + ( real-estate-object-income ( first object-list ) )
        ( income-from-objects ( rest object-list ) ) ) ) )

( define-struct real-estate-object ( ... income ... ) )
```

Nun die Funktion income-from-objects, die durch die Liste der Immobilien geht und die Einkünfte aus den einzelnen Immobilien aufsummiert.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( income-from-objects object-list )
  ( if ( empty? object-list )
     0
     ( + ( real-estate-object-income ( first object-list ) )
        ( income-from-objects ( rest object-list ) ) ) ) )

( define-struct real-estate-object ( ... income ... ) )
```

Jede Immobilie erzeugt Einkünfte.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( income-from-objects object-list )
  ( if ( empty? object-list )
     0
     ( + ( real-estate-object-income ( first object-list ) )
        ( income-from-objects ( rest object-list ) ) ) ) )

( define-struct real-estate-object ( ... income ... ) )
```

Natürlich hat eine Immobilie noch viele andere Attribute, die wir hier nicht besprechen, deshalb wieder ein Struct-Typ für einzelne Immobilien.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( income-from-objects object-list )
  ( if ( empty? object-list )
     0
     ( + ( real-estate-object-income ( first object-list ) )
          ( income-from-objects ( rest object-list ) ) ) ) )

( define-struct real-estate-object ( ... income ... ) )
```

Die Aufsummierung der einzelnen Einkünfte folgt exakt dem Schema, das wir schon bei der Aufsummierung der Elemente einer Liste von Zahlen gesehen haben.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( income-from-objects object-list )
  ( if ( empty? object-list )
     0
     ( + ( real-estate-object-income ( first object-list ) )
        ( income-from-objects ( rest object-list ) ) ) ) )

( define-struct real-estate-object ( ... income ... ) )
```

Der einzige Unterschied ist der, dass nicht einfach das erste Element hergenommen und aufsummiert wird, denn die Elemente sind ja keine Zahlen. Statt dessen muss aus jedem Element, das ja vom Struct-Typ real-estate-object ist, das Attribut income extrahiert und addiert werden.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( investment-income person )
    ( income-from-investments
        ( person-data-investments person ) ) )

( define-struct person-data ( ... investments ... ) )
```

Jetzt fehlt endgültig nur noch die Funktion investment-income. Diese Funktion ist völlig analog zur Funktion real-estate-income.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( investment-income person )
    ( income-from-investments
        ( person-data-investments person ) ) )

( define-struct person-data ( ... investments ... ) )
```

Der Unterschied ist, dass jetzt nicht auf ein Attribut real-estate-objects, sondern auf ein weiteres Attribut zugegriffen wird, das wir investments nennen.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( income-from-investments investment-list )
  ( if ( empty? investment-list )
       0
       ( + ( investment-income ( first investment-list ) )
          ( income-from-investments
                                ( rest investment-list ) ) ) ) )

( define-struct investment ( ... income ... ) )
```

Auch die Funktion zur Aufsummierung der Einkünfte aus Investments ist völlig analog zur Funktion income-from-objects, die wir eben definiert hatten.

Programme aus Funktionen

TECHNISCHE UNIVERSITÄT DARMSTADT

```
( define ( income-from-investments investment-list )
  ( if ( empty? investment-list )
       0
       ( + ( investment-income ( first investment-list ) )
           ( income-from-investments
                                ( rest investment-list ) ) ) ) )

( define-struct investment ( ... income ... ) )
```

Auch hier der einzige Unterschied, wie der Zahlenwert aus jedem einzelnen Listenelement extrahiert wird.

Nun fehlt nur noch die Funktion pre-payment. Wie schon gesagt, lassen wir die hier aus, das Prinzip sollte auch so klar geworden sein.

TECHNISCHE UNIVERSITÄT DARMSTADT

Aufweichung der reinen funktionalen Lehre in Racket

Nun kommen wir zu den angekündigten weiteren Konzepten von Racket, von denen wir gesagt haben, dass sie unmittelbar mit der Diskussion, was typisch funktional ist, zu tun haben.

Keine ref. Transparenz in Java

TECHNISCHE UNIVERSITÄT DARMSTADT

```
public class X {
   private static int m = 1;
   public static int foo ( int n ) {
      m++;
      return n + m;
   }
}                    if ( 2 * X.foo(1) != X.foo(1) + X.foo(1) )   //  true!
```

- Derselbe Aufruf kann jedes Mal völlig unterschiedliche Ergebnisse haben

In Java ist referentielle Transparenz möglich, wird aber nicht unterstützt und schon gar nicht erzwungen. Das sehen Sie an diesem kleinen Beispiel: Referentielle Transparenz würde fordern, dass jeder Aufruf einer Methode mit demselben aktualen Parameter zum selben Ergebnis führen würde. Das ist in Java offenkundig nicht vorgesehen.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Allerdings ist die rein deklarative Lehre extrem einengend, und so ist es nicht überraschend, dass gängige funktionale und andere deklarative Sprachen Konstrukte bieten, die beim besten Willen nicht mehr der rein funktionalen Lehre entsprechen. Wir betrachten hier nur ein Beispiel dafür, dass es Hintertüren gibt, um in Racket doch wieder zeitliche Abläufe einzuführen.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Zunächst einmal etwas rein Funktionales: die Definition einer Konstante.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Mit einer kleinen Besonderheit, die wir bisher noch nicht gesehen haben: Mit dieser Syntax kann man angeben, dass die Konstante last-called einen leeren Wert hat. Das kann man sich vielleicht ähnlich zu null in Java vorstellen.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Jetzt eine ganz normale Funktion, die wir schon gesehen haben: Die Fakultät einer gegebenen Zahl n wird rekursiv berechnet.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Dieses Konstrukt ist neu: Es besagt, dass bis zur schließenden Klammer mehrere Ausdrücke nacheinander kommen werden, und der Wert des begin-Ausdrucks ist der Wert des letzten dieser Ausdrücke. In diesem Beispiel folgen zwei Ausdrücke, und der Wert des begin-Ausdrucks ist daher der Wert des zweiten Ausdrucks.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Das ist der erste der beiden Ausdrücke, die im begin-Ausdruck zusammengefasst sind. Den schauen wir uns gleich genauer an.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Und das ist der zweite Ausdruck. Damit liefert factorial den Wert zurück, den die Funktion auch in der früheren Version ohne begin schon zurückgeliefert hat und ja auch zurückliefern soll.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Jetzt schauen wir uns den ersten Ausdruck im begin-Ausdruck an. Welchen Sinn hat ein Ausdruck, dessen Rückgabewert nicht weiterverwendet wird? Noch schlimmer: Ausdrücke mit set! haben überhaupt keinen Rückgabewert.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Zeitliche Abläufe: set! und begin

```
( define last-called #<void> )

( define ( factorial n )
    ( begin
        ( set! last-called ´factorial )
        ( if ( = 0 n ) 1 ( * ( n ( factorial ( - n 1 ) ) ) ) ) ) )
```

Der Punkt ist, dass ein Ausdruck mit set! einen Seiteneffekt hat: Er überschreibt den Wert der Konstanten, die als erster Parameter aufgeführt wird, mit dem Wert des zweiten Parameters. Durch Aufruf der Funktion factorial wird also die Konstante last-called wie eine Variable mit dem Symbol ´factorial überschrieben. Im Grunde ist ein Ausdruck mit set! daher nichts anderes als eine ganz normale Zuweisung, wie wir sie auch aus Java kennen.

Dieses Beispiel hat durchaus seinen Sinn: Wenn man ein solches Konstrukt in verschiedene Funktionen setzt, die jeweils last-called wie hier mit dem Namen der jeweiligen Funktion überschreiben, dann steht in last-called immer die Information, welche dieser Funktionen als Letztes aufgerufen wird.

Das ist natürlich nur ein sehr einfaches, illustratives Beispiel für den Sinn und Zweck von begin und set!. Als Übung können Sie versuchen, mit begin und set! eine Schleife in Racket zu realisieren.

Aufweichung in Racket

TECHNISCHE UNIVERSITÄT DARMSTADT

Objektidentität in Racket:

- **(eq? a b) testet ob a und b dasselbe Objekt sind**
- ***Aber Achtung*: System hat freie Entscheidung ob zwei wertgleiche Objekte als *ein* oder *zwei* Objekte realisiert sind**
- **Daher kann es systemabhängige Überraschungen geben**
- **Besser also nicht verwenden, wenn es nicht absolut unumgänglich ist**

Auch Objektidentität wird in Racket gegen die reine funktionale Lehre eingeführt. Man kann mit einem eigenen booleschen Operator tatsächlich testen, ob zwei Objekte dasselbe Objekt sind oder nicht. Wie schon gesagt, ist das System frei darin, wertgleiche Objekte als eines oder als mehrere zu realisieren, was sich natürlich auf das Testergebnis auswirkt. Dieser Test ist nur in sehr speziellen Fällen sinnvoll, man sollte ihn generell vermeiden.

Damit ist unsere Einführung in die funktionale Programmierung beendet.