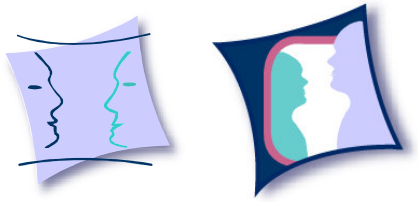
Technische Universität Darmstadt



Telecooperation/RBG

Grundlagen der Informatik I
Thema 2: Strukturierte Datentypen

Prof. Dr. Max Mühlhäuser
Dr. Guido Rößling

Strukturen

- Die Eingabe/Ausgabe einer Prozedur ist häufig nicht ein einzelner atomarer Wert (Zahl, Boolean, Symbol), sondern ein Datenobjekt mit vielen Eigenschaften
  - Bei einer CD etwa Interpret, Titel und Preis
  - Wir brauchen Mechanismen, um Daten zusammenzufassen

- Einer dieser Mechanismen sind **Strukturen**
  - Eine Strukturdefinition hat folgende Form:

```
(define-struct s (field1 … fieldN))
```

  - Beispiel

```
(define-struct point (x y))
```

Strukturdefinition

```
(define-struct s (field1 … fieldN))
```

```
(define-struct point (x y))
```

Diese Definition erzeugt eine Reihe von Prozeduren:

- **`make-s`**
  - Eine **Konstruktor** Prozedur, die *N* Argumente bekommt und einen neuen Struktur-Wert zurückliefert
  - So erzeugt `(define p (make-point 3 4))` einen neuen Punkt mit x = 3, y = 4
- `s?`
  - Eine **Prädikat**-Prozedur, die **`true`** zurückliefert für einen Wert der durch **`make-s`** erzeugt wurde und **`false`** für jeden anderen Wert
  - `(point? p)` → `true`
- `s-field`
  - Für jedes Feld einen **Selektor**, der eine Struktur als Argument bekommt und den Wert des Feldes extrahiert
  - `(point-y p)` → 4

Design von Prozeduren für zusammengesetzte Daten

- Wann brauchen wir Strukturen?
  - Immer dann, wenn die Beschreibung eines Objekts aus mehreren unterschiedlichen Informationen besteht
- Wie ändert sich unser Design-Rezept?
  - Datenanalyse: Problembeschreibung durchsuchen nach Beschreibungen relevanter Objekte, dann entsprechende Datentypen anlegen, Vertrag des Datentyps beschreiben
  - Definition eines Vertrags kann die neu definierten Typnamen benutzen, z.B.
    ;; bafoegberechtigt : Student → bool
  - Template: Header + Body, in dem alle möglichen Selektoren aufgeführt werden
  - Implementation des Bodies: Entwurf eines Ausdrucks, der primitive Ausdrücke, andere Funktionen, Selektor-Ausdrücke und die anderen Parameter enthält

Beispiel

```
;; Data Analysis & Definitions:
(define-struct student (last first teacher))
;; A student is a structure: (make-student l f t)
;; where l, f, and t are symbols.

;; Contract: subst-teacher : student symbol  ->  student
;; Purpose: to create a student structure with a new
;; teacher name if the teacher's name matches 'Fritz

;; Examples:
;;(subst-teacher (make-student 'Fit 'Matthew 'Fritz) 'Elise)
;; = (make-student 'Fit 'Matthew 'Elise)
;;
;;(subst-teacher (make-student 'Smith 'John 'Bill) 'Elise)
;; = (make-student 'Smith 'John 'Bill)
```

Dr. G. Rößling
Prof. Dr. M. Mühlhäuser
RBG / Telekooperation

Beispiel (fortgesetzt)

```
;; Template:
;; (define (subst-teacher a-student a-teacher)
;; ... (student-last a-student) ...
;; ... (student-first a-student) ...
;; ... (student-teacher a-student) ...)
;; Definition:
(define (subst-teacher a-student a-teacher)
  (cond
    [(symbol=? (student-teacher a-student) 'Fritz)
     (make-student (student-last a-student)
                   (student-first a-student)
                   a-teacher)]
    [else a-student]))
;; Test 1:
(subst-teacher (make-student 'Fit 'Matthew 'Fritz) 'Elise)
;; expected value:
(make-student 'Fit 'Matthew 'Elise)
;; Test 2:
(subst-teacher (make-student 'Smith 'John 'Bill) 'Elise)
;; expected value:
(make-student 'Smith 'John 'Bill)
```

Die Bedeutung von Strukturen im Substitutionsmodell (1/2)

- Wie funktioniert define-struct im Substitutionsmodell?

```
(define-struct c (f1 … fn))
```

- Diese Struktur erzeugt folgende Operationen:
  - **`make-c`** : einen Konstruktor
  - **`c-f1`** … **`c-fn`**: eine Serie von Selektoren
  - **`c?`** : ein Prädikat

Die Bedeutung von Strukturen im Substitutionsmodell (2/2)

- Es wird wie bei jeder Kombination vorgegangen:
  - Auswertung des Operators und der Operanden
  - Der Wert von `(make-c v1 … vn)` ist `(make-c v1 … vn)`
    - Konstruktoren sind also selbstauswertend!
  - Die Auswertung von `(c-fi v)` ergibt
    - `vi` falls v = `(make-c v1 …vi … vn)`
    - einen Fehler in allen anderen Fällen
  - Die Auswertung von `(c? v)` ergibt
    - `true`, falls v = `(make-c v1 … vn)`
    - `false`, sonst

- Probieren Sie es mit dem DrScheme Stepper aus!

Datenabstraktion

- Wir haben für Prozeduren
  - primitive Ausdrücke (`+`, `-`, `and`, `or`, …)
  - Kombinationsmittel (Prozeduranwendung)
  - Abstraktionsmittel (prozedurale Abstraktion)

- Das gleiche gibt es für Daten:
  - primitive Daten (Zahlen, Boolean, Symbole)
  - zusammengesetzte Daten (z.B. Strukturen)
  - **Datenabstraktion**

Warum brauchen wir Datenabstraktion?

- Beispiel: Implementieren einer Operation zum Addieren von rationalen Zahlen:
  - Rationale Zahlen bestehen aus Zähler und Nenner, z.B. 1/2 oder 7/9.
  - Die Addition zweier rationaler Zahlen hat zwei Ergebnisse: Den resultierenden Zähler und den resultierenden Nenner.
  - Eine Prozedur kann aber nur einen Wert zurückgeben.
  - Wir brauchen also zwei Prozeduren: Eine gibt den resultierenden Zähler, die andere den Nenner zurück.
  - Wir müssten uns merken, welcher Zähler zu welchem Nenner gehört.
- Datenabstraktion ist eine Methode, die mehrere Datenobjekte kombiniert, so dass sie wie eines gehandhabt werden können. Wie das geschieht, versteckt die Datenabstraktion.

Warum brauchen wir Datenabstraktion?

- Die neuen Datenobjekte sind abstrakte Daten:
  - Sie werden benutzt, ohne Annahmen über ihre Implementierung zu machen.
- Datenabstraktion hilft dabei, …
  - das konzeptuelle Level zu erhöhen, auf dem wir Programme entwerfen,
  - die Modularität der Designs zu erhöhen und
  - die Ausdrucksstärke der Programmiersprache zu verbessern.
- Eine konkrete Repräsentation der Daten ist unabhängig von Programmen definiert, die die Daten verwenden.
- Die Schnittstelle zwischen der Repräsentation und den Programmen, die die abstrakten Daten verwenden, sind ein paar Prozeduren, die Selektoren und Konstruktoren genannt werden.

Spracherweiterungen für die Handhabung von abstrakten Daten

- Konstruktor: Prozedur, die Instanzen der abstrakten Daten aus Werten erzeugt, die ihr übergeben werden
- Selektor: Prozedur, die einen der Werte zurückgibt, aus denen das abstrakte Datenobjekt zusammengesetzt ist
- Die Komponentenwerte, die von einem Selektor zurückgegeben werden, können
  - der Wert einer internen Variablen sein oder
  - berechnet worden sein.
- Die von **`define-struct`** erzeugten Konstruktoren/Selektoren sind ein Spezialfall
  - Der Rückgabewert eines Selektors ist immer einer der Werte, die beim zugehörigen Konstruktoraufruf übergeben wurden (werden dabei nicht berechnet)

Beispiel: Rationale Zahlen

- Mathematisch dargestellt durch ein Paar von Ganzzahlen: 1/2, 7/9, 56/874, 78/23, etc.

- Konstruktor:

```
(make-rat numerator denominator)
```

- Selektor:

```
(numer rn)
(denom rn)
```

- Das ist alles, was der Benutzer wissen muss!
  - Für den Programmierer und DrScheme langt es noch nicht ganz, da wir die Struktur „rat" nicht definiert haben – das passiert in ein paar Folien.

Benutzerdefinierte Operationen für rationale Zahlen

- Multiplikation von $x = n_x/d_x$ und $y = n_y/d_y$

  $(n_x/d_x) * (n_y/d_y) = (n_x*n_y) / (d_x*d_y)$

```
;; mul-rat: rat rat -> rat
;; Multiplies two rational numbers
;; Example: (mul-rat (make-rat 1 2) (make-rat 2 3)
;;           = (make-rat 2 6)
(define (mul-rat x y)
  (make-rat (* (numer x)
               (numer y))
            (* (denom x)
               (denom y))))
```

Eine weitere Operation

- Addition von $x = n_x/d_x$ und $y = n_y/d_y$

  $n_x/d_x + n_y/d_y = (n_x*d_y + n_y*d_x) / (d_x*d_y)$

```
;; add-rat: rat rat -> rat
(define (add-rat x y)
  (make-rat (+ (* (numer x)
                  (denom y))
               (* (numer y)
                  (denom x)))
            (* (denom x)
               (denom y)))))
```

Subtraktion und Division werden ähnlich wie Addition und Multiplikation implementiert.

Eine Abfrage

- Gleichheit:

  $n_x/d_x = n_y/d_y$

  gdw.

  $n_x*d_y = n_y*d_x$

gdw steht für:
„genau dann, wenn“

```
;; equal-rat: rat rat -> boolean
(define (equal-rat? x y)
  (= (* (numer x) (denom y))
     (* (numer y) (denom x))))
```

Eine Ausgabeoperation

Um rationale Zahlen auszugeben, definieren wir eine Ausgabeprozedur, die auf der Datenabstraktion aufbaut.

```
;; print-rat: rat -> String
(define (print-rat x)
  (string-append
    (number->string (numer x))
    "/"
    (number->string (denom x))))
```

Dies ist unser erstes Beispiel mit Stringmanipulation!
`string-append` setzt mehrere Strings zusammen.
`number->string` wandelt eine Zahl in einen String um.
Das wäre mit Symbolen nicht möglich.

Unterhalb der Datenabstraktion

- Wir haben die Operationen **`add-rat`**, **`mul-rat`** und **`equal-rat`** implementiert, indem wir **`make-rat`**, **`denom`**, **`numer`** verwendet haben.
  - Ohne dass make-rat, denom, numer implementiert wurden!
  - Auch ohne zu wissen, wie diese implementiert werden.

- Wir müssen jetzt noch **`make-rat`**, **`numer`**, **`denom`** definieren.
  - Dafür müssen wir Zähler und Nenner zusammenkleben.
  - Hierzu legen wir eine Scheme-Struktur für das Speichern von Paaren an:

```
(define-struct xy (x y))
```

Darstellung von rationalen Zahlen

Wir können den Konstruktor und die Selektoren mit Hilfe der xy Struktur definieren.

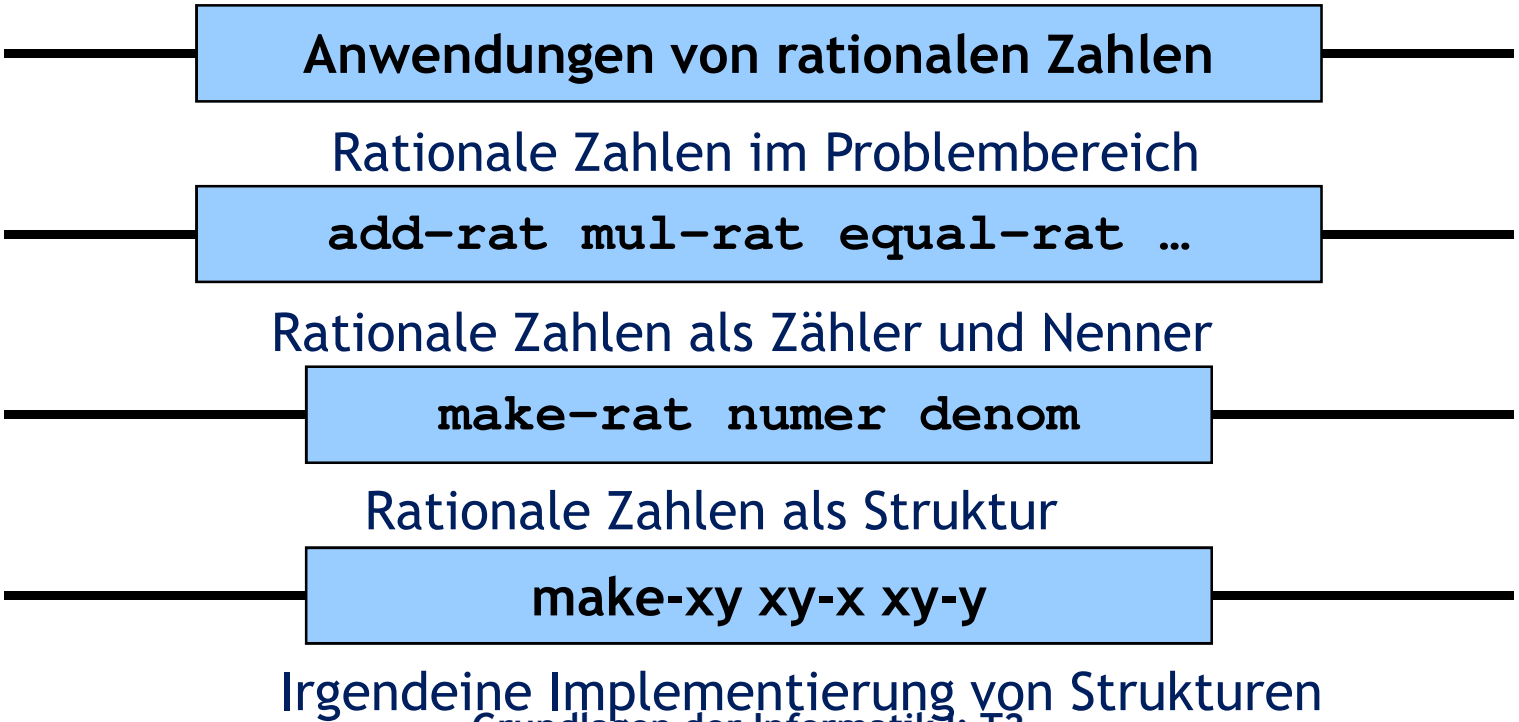
```
(define (make-rat n d) (make-xy n d))

(define (numer r) (xy-x r))

(define (denom r) (xy-y r))
```

Benutzen der Operationen für rationale Zahlen

```
(define one-third (make-rat 1 3))

(define four-fifths (make-rat 4 5))

(print-rat one-third)
“1/3“

(print-rat (mul-rat one-third four-fifths))
“4/15“

(print-rat (add-rat four-fifths four-fifths))
“40/25“
```

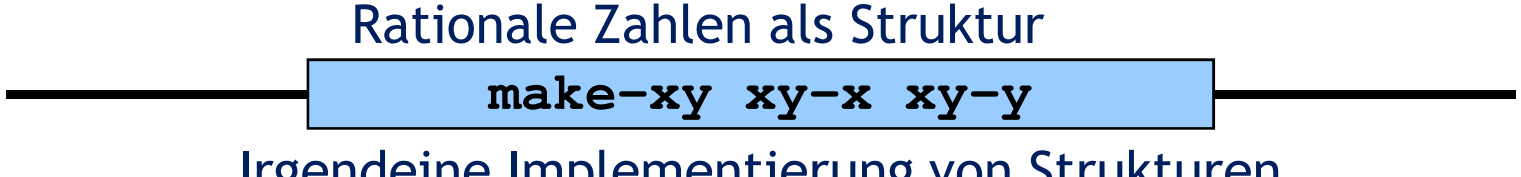
Abstraktionsebenen

- Programme werden als Ebenen von Spracherweiterungen aufgebaut.
- Jede Ebene ist eine Abstraktionsebene.
- Jede Abstraktion versteckt einige Implementierungsdetails.
- In unserem Beispiel gibt es vier Abstraktionsebenen.



Unterste Ebene

- Ebene der Paare
- Die Prozeduren **`make-xy`**, **`xy-x`** und **`xy-y`** werden bereits vom Interpreter aufgrund der Strukturdefinition erzeugt.
- Die tatsächliche Implementierung von Paaren bzw. Strukturen wird versteckt.



Zweite Ebene

- Ebene der rationalen Zahlen als Datenobjekte
- Die Prozeduren **`make-rat`**, **`numer`** und **`denom`** werden auf dieser Ebene definiert.
- Die tatsächliche Implementierung der rationalen Zahlen wird versteckt.

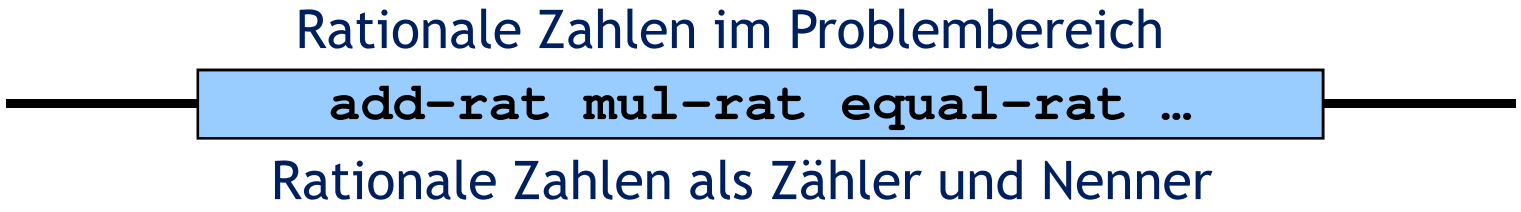
Rationale Zahlen als Zähler und Nenner

`make-rat numer denom`

Rationale Zahlen als Struktur

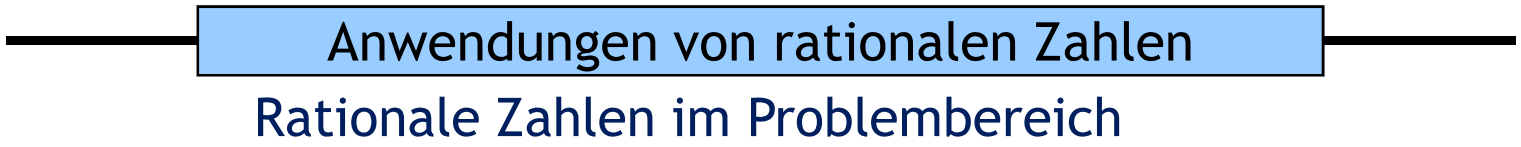
Dritte Ebene

- Ebene der Dienst-Prozeduren für rationale Zahlen
- Die Prozeduren **`add-rat, mul-rat, equal-rat`**, etc. sind auf dieser Ebene definiert.
- Die Implementierung dieser Prozeduren wird versteckt.



Oberste Ebene

- Programmebene
- Rationale Zahlen werden in Berechnungen verwendet (analog zu normalen Zahlen).



Abstraktionsbarrieren

- Jede Ebene ist so entworfen, dass sie Implementierungsdetails vor höheren Ebenen versteckt.
- Diese Ebenen sind Abstraktionsbarrieren.

Vorteile von Datenabstraktion

- Die Abstraktionsebenen eines Programms können nacheinander entworfen werden.
- Dadurch unterstützt Datenabstraktion top-down Design.
  - Wir können graduell entscheiden, wie Daten dargestellt werden und wie Konstruktoren, Selektoren und Dienst-Prozeduren implementiert sein müssen.
- Wir müssen uns keine Gedanken über Implementierungsdetails von tieferen Ebenen machen.
- Eine Implementierung kann später einfach verändert werden, ohne dass Prozeduren, die auf einer höheren Ebene geschrieben wurden, verändert werden müssen.

Ändern der Darstellung von Daten

- Vor ein paar Folien haben wir gesehen:

```
(print-rat (add-rat four-fifths four-fifths))
“40/25“
```

- Unsere rationalen Zahlen sind nicht immer reduziert.
- Wir entscheiden uns dafür, die rationalen Zahlen immer reduziert darzustellen.
  - 40/25 und 8/5 sind dieselben Zahlen.
  - Wegen der Datenabstraktion ist es für die Dienst-Prozeduren egal, ob die rationalen Zahlen reduziert sind oder nicht.
  - Die Prozeduren wie **`add-rat`** oder **`equal-rat`** funktionieren in beiden Fällen korrekt.

Ändern der Darstellung von Daten

- Wir können den Konstruktor ändern…

```
(define (make-rat n d)
    (make-xy
      (/ n (gcd n d))
      (/ d (gcd n d))))
```

- …oder die Selektoren

```
(define (numer r)
    (/ (xy-x r) (gcd (xy-x r) (xy-y r))))
(define (denom r)
    (/ (xy-y r) (gcd (xy-x r) (xy-y r))))
```

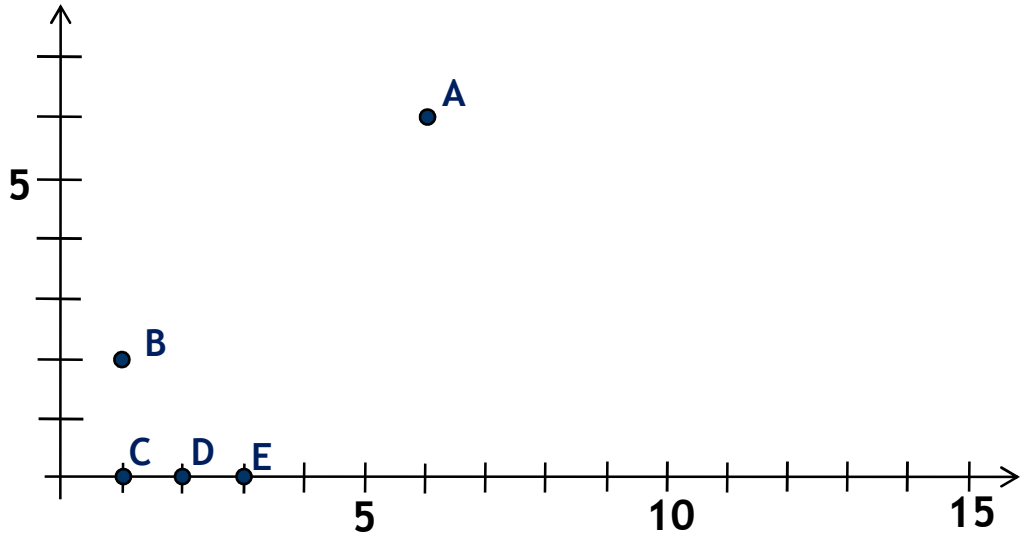
`gcd` ist eine eingebaute Prozedur zum Berechnen des größten gemeinsamen Teilers zweier natürlicher Zahlen

Mit unterschiedlichen Daten umgehen

- Bis jetzt haben unsere Prozeduren immer nur einen Typ von Daten verarbeitet
  - Zahlen
  - Booleans
  - Symbole
  - Exemplare einer bestimmten Struktur

- Häufig möchten wir jedoch, dass Prozeduren mit unterschiedlichen Arten von Daten funktionieren
- Außerdem werden wir lernen, wie man Prozeduren vor falscher Benutzung schützt

Beispiel



- Gegeben sei `(define-struct point (x y))` für Punkte
- Viele Punkte liegen auf der x-Achse
- Für diesen Fall möchten wir Punkte nur durch eine Zahl darstellen
  - A = (make-point 6 6), B= (make-point 1 2)
    C = 1, D = 2, E = 3

Mit unterschiedlichen Daten umgehen

- Um unsere Repräsentation von Punkten zu dokumentieren, machen wir folgende informelle Datentypdefinition

```
;; a pixel-2 is either
;;  1. a number
;;  2. a point-structure
```

- Der Vertrag, die Beschreibung und der Header einer Prozedur distance-to-0 ist nun einfach:

```
;; distance-to-0 : pixel-2  ->  number
;; to compute the distance of a-pixel to the origin
(define (distance-to-0 a-pixel) ...)
```

- Wie können wir zwischen den Datentypen unterscheiden?
  - Mit Hilfe der Prädikate: number?, point? etc.

Mit unterschiedlichen Daten umgehen

- Basisstruktur: Prozedurbody mit **cond**-Ausdruck, der nach Typ unterscheidet

```
(define (distance-to-0 a-pixel)
  (cond
    [(number? a-pixel) ...]
    [(point? a-pixel) ...]))
```

- Im zweiten Fall wissen wir, dass der Input aus zwei Koordinaten besteht…

```
(define (distance-to-0 a-pixel)
  (cond
    [(number? a-pixel) ...]
    [(point? a-pixel)
            …(point-x a-pixel)…(point-y a-pixel)…]))
```

Mit unterschiedlichen Daten umgehen

- Die Fertigstellung der Funktion ist nun einfach…

```
(define (distance-to-0 a-pixel)
  (cond
    [(number? a-pixel) a-pixel]
    [(point? a-pixel)
             (sqrt (+ (sqr (point-x a-pixel))
                      (sqr (point-y a-pixel))))]))
```

Eingebaute Prozeduren:
- (sqr x) : x zum Quadrat
- (sqrt x) : Quadratwurzel von x

Mit unterschiedlichen Daten umgehen

- Weiteres Beispiel: Grafische Objekte (shapes)
  - Varianten: squares, circles,…
  - Prozeduren: Umfang berechnen, zeichnen, …

```
;; A shape is either
;;    a circle structure:
;;       (make-circle p s)
;;       where p is a point describing the center
;;       and s is a number describing the radius; or
;;    a square structure:
;;       (make-square nw s)
;;        where nw is the north-west corner point
;;        and s is a number describing the side length.

(define-struct circle (center radius))
(define-struct square (nw length))

;; Examples:(make-circle (make-point 5 9) 87)
;;          (make-square (make-point 20 5) 5)
```

Mit unterschiedlichen Daten umgehen

- Umfang (perimeter) berechnen:
  - Anwendung unseres Design-Rezepts…

```
;; perimeter : shape ->  number
;; to compute the perimeter of a-shape
(define (perimeter a-shape)
  (cond
    [(square? a-shape) ... ]
    [(circle? a-shape) ... ]))
```

  - Anwendung unseres Design-Rezepts… (fortgesetzt)

```
;; perimeter : shape->  number
;; to compute the perimeter of a-shape
(define (perimeter a-shape)
 (cond
  [(square? a-shape)
   ..(square-nw a-shape)..(square-length a-shape)..]
  [(circle? a-shape)
   ..(circle-center a-shape)..(circle-radius a-shape)..]))
```

Mit unterschiedlichen Daten umgehen

- Umfang (perimeter) berechnen:
  - Endresultat

```
;; perimeter : shape ->  number
;; to compute the perimeter of a-shape

(define (perimeter a-shape)
  (cond
    [(square? a-shape) (* (square-length a-shape) 4)]
    [(circle? a-shape) (* (* 2 (circle-radius a-shape)) pi)]))
```

Programmdesign und heterogene Daten

- Datenanalyse wird sehr viel wichtiger
  - Welche Klassen von Objekten gibt es und was sind ihre Attribute?
  - Welche sinnvollen Gruppierungen von Klassen gibt es?
    - Sogenannte “Subklassenbildung”
  - Im Allgemeinen entsteht eine Hierarchie von Datendefinitionen
- Templates
  - 1. Schritt: **`cond`**-Ausdruck, der die unterschiedlichen Typen innerhalb einer Gruppe unterscheidet
  - 2. Schritt: zu jedem Zweig entsprechende Selektoren einfügen
  - Alternativ: in jedem Zweig Datentyp-spezifische Prozedur aufrufen (z.B.: Prozeduren **`perimeter-circle`**, **`perimeter-square`**)
- Programmkörper
  - In jedem Zweig der Fallunterscheidung separat die verfügbaren Informationen je nach Anwendung kombinieren
  - Alternativ: Datentyp-spezifische Prozeduren nacheinander implementieren, nach bewährtem Designrezept vorgehen
- Übersicht über neuen Designprozess siehe HtDP Fig. 18

Get your data structures correct first,
and the rest of the program will write itself.

David Jones