Informatik I

3. Verzweigungen und Wahrheitswerte

Jan-Georg Smaus

Albert-Ludwigs-Universität Freiburg

26. Oktober 2010

Informatik I

26. Oktober 2010 — 3. Verzweigungen und Wahrheitswerte

3.1 Wiederholung

3.2 Die Sorten `boolean` und `string`

3.3 Verzweigungen

3.1 Wiederholung

Bisher . . .

- Programme, Sequenzen von Formen
- Ausdrücke und ihre Auswertung (Substitutionsmodell)
- Konstruktionsanleitung für Prozeduren
  - Kurzbeschreibung
  - Sorten und Signaturen $\Rightarrow$ Gerüst
  - Testfälle
  - Rumpf ausfüllen
  - Testen

Erinnerung: Lexikalische Bindung

- ```
  ((lambda (x1)
     (+ ((lambda (x2) (* x3 3)) 3)
        (* x4 2))) 14)
  ```
  Die Vorkommen 1 und 2 von x sind bindend, und die Vorkommen, 3 und 4 sind gebunden.
- Es gilt die lexikalische Bindung: Bezug auf das bindende Vorkommen der innersten textlich umschließenden Abstraktion.
- Äquivalenter Ausdruck durch konsistente Umbenennung:

```
((lambda (x1)
   (+ ((lambda (y2) (* y3 3)) 3)
      (* x4 2))) 14)
```


Aufgabe: Rauminhalt eines Zylinders

Eingabe: Radius r und Höhe h eines Zylinders

Ausgabe: Rauminhalt des Zylinders = Grundfläche $*$ Höhe

```
; Rauminhalt eines Zylinders berechnen
(: cylinder-volume (number number -> number))
(define cylinder-volume
  (lambda (radius height)
    (* (circle-area radius) height)))
; Testfall
(check-within (cylinder-volume 1 1) 3.14159 1e-5)
(check-within (cylinder-volume 2 1) 12,56636 1e-4)
(check-within (cylinder-volume 1 4) 12,56636 1e-4)
```


Mantra

Mantra #3 (Strukturerhaltung)

Versuche, das Programm wie das Problem zu strukturieren.

„Untermantras“:

Mantra #4 (Abstraktion)

Schreibe eine Abstraktion für jedes Unterproblem.

Mantra #5 (Namen)

Definiere Namen für häufig benutzte Konstanten und verwende diese Namen anstelle der Konstanten, für die sie stehen.

Wie geht es weiter?

Die bisher definierten Funktionen waren allesamt einfache Kombinationen der eingebauten arithmetischen Operationen.
Es gibt aber noch andere Daten (Sorten) als nur Zahlen, und die einfachen Kombinationen wie bisher reichen auch nicht immer aus.
```

3.2 Die Sorten boolean und string

- Wahrheitswerte
- Strings

Die Sorte boolean (Wahrheitswerte)

Die Sorte boolean enthält zwei Literale:

```
#t      ; wahr
#f      ; falsch
```


Die Sorte boolean: Operationen ...

...mit Signatur `(real real -> boolean)`

`=` `<` `>` `>=` `<=`

...mit Signatur `(real -> boolean)`

`zero?` `positive?` `negative?`

...mit Signatur `(integer -> boolean)`

`odd?` `even?`

...mit Signatur `(boolean ... -> boolean)`

`and` `or`

...mit Signatur `(boolean -> boolean)`

`not`

and, or, and not heißen logische Operationen. and und or haben eine spezielle Auswertungsregel!

Ausdrücke der Sorte boolean

```
   #t      (= 17 4)      (>= 17 4)      (odd? (+ 17 4))
=> #t   => #f         => #t          => (odd? 21)
                                     => #t
```

```
   (define y 1)
   (and (= 5 (+ (* 2 2) y)) (and (<= 0 y) (< y 2)))
=> (and (= 5 (+ 4 y)) (and (<= 0 y) (< y 2)))
=> (and (= 5 (+ 4 1)) (and (<= 0 y) (< y 2)))
=> (and (= 5 5) (and (<= 0 y) (< y 2)))
=> (and #t (and (<= 0 y) (< y 2)))
=> (and (<= 0 y) (< y 2))
=> (and (<= 0 1) (< y 2))
=> (and #t (< y 2))
=> (< y 2)
=> (< 1 2)   => #t
```

Die Sorte string (Zeichenketten)

Die Sorte String haben wir schon einmal kurz erwähnt (Literal "Banane").
I.a. haben die Literale dieser Sorte die Form

"$c_1 c_2 \ldots c_n$"

wobei die c_i beliebige Zeichen („auf der Tastatur“) außer " sein dürfen.
Schreibe \" um " in einer Zeichenkette zu verwenden.
Beispiele:

```
"Der Mond ist aufgegangen."
"Harry schrie: \"Expelliarmus!\""
```

Operationen (mit Signatur `(string string -> boolean)`)

```
string=?   string<?   string>?
```

u.v.a.m., siehe Dokumentation

3.3 Verzweigungen

- Fallunterscheidung anhand der Ausgabesorte
- Fallunterscheidung anhand der Eingabesorte
- Fallunterscheidung ohne Hilfe durch die Sorte
- Konstruktionsanleitung 2: Fallunterscheidung
- Semantik
- Weitere Formen von Verzweigungen
- Verzweigungen mit booleschen Operatoren
- Verbesserte Signaturen

Signatur für Aufzählungen

Betrachten wir folgendes Problem:

Aufgabe: Aggregatzustand von Wasser bestimmen

Eingabe: Wassertemperatur t

Ausgabe: „solid“, „liquid“ oder „gaseous“ je nachdem, ob die Temperatur weniger als 0, zwischen 0 und 100, oder über 100 beträgt.

Erste Beobachtung: Die Ausgabe ist keine Zahl, sondern von einer Sorte, die drei Literale enthält.
Scheme-Syntax zur Definition dieser Sorte:

```
(define phase
  (signature (one-of "solid" "liquid" "gaseous")))
```

`phase` (Aggregatzustand) ist Sorte so wie `real` oder `natural`. Wir nennen eine mittels `one-of` definierte Sorte Aufzählungssorte.

Fallunterscheidungen

Zweite Beobachtung: Es gibt drei mögliche Ausgaben, und welche die richtige ist, muss von irgendwelchen Bedingungen abhängen. Hier benötigen wir eine Fallunterscheidung:

$$p(t) = \begin{cases} \text{„solid“} & \text{falls } \ldots \\ \text{„liquid“} & \text{falls } \ldots \\ \text{„gaseous“} & \text{falls } \ldots \end{cases}$$

Diese mathematische Notation heißt Verzweigung und sie testet Bedingungen. Eine Bedingung besitzt einen Wahrheitswert, d.h. sie hat die Sorte `boolean`.

Scheme-Notation für eine Verzweigung

In Scheme schreibt man diese Verzweigung so:

```
(cond
 (...   "solid")
 (...   "liquid")
 (...   "gaseous"))
```

wobei an Stelle der Ellipsen (...) Bedingungen stehen müssen (dazu später).

Konstruktion: Aggregatzustand

Das Gerüst

Zunächst das Gerüst:

```
; Aggregatzustand von Wasser bestimmen
(: water-phase (real -> phase))
(define water-phase
  (lambda (t)
    (...)))
```

Dann ein paar Testfälle:

```
(check-expect (water-phase -5) "solid")
(check-expect (water-phase 0) "liquid")
(check-expect (water-phase 100) "liquid")
(check-expect (water-phase 120) "gaseous")
```


Konstruktion: Aggregatzustand

Eingabe einsetzen

Dann setzen wir (wie immer) die Eingabe in den Rumpf ein:

```
; Aggregatzustand von Wasser bestimmen
(: water-phase (real -> phase))
(define water-phase
  (lambda (t)
    (... t ...)))
```


Konstruktion: Aggregatzustand

Schablone für Fallunterscheidungen

Als nächstes wenden wir eine Schablone an, die sich aus der Existenz dreier möglicher Ausgaben ergibt:

```
; Aggregatzustand von Wasser bestimmen
(: water-phase (real -> phase))
(define water-phase
  (lambda (t)
    (... t ...)))
    (cond
     (...   "solid")
     (...   "liquid")
     (...   "gaseous"))))
```

Konstruktion: Aggregatzustand

Eingabe in die Bedingungen

Welche Ausgabe die richtige ist, muss von irgendwelchen Bedingungen abhängen, und diese können nur die Eingabe betreffen. Daher setzen wir ein:

```
; Aggregatzustand von Wasser bestimmen
(: water-phase (real -> phase))
(define water-phase
  (lambda (t)
    (... t ...)))
    (cond
     (... t ...  "solid")
     (... t ...  "liquid")
     (... t ...  "gaseous"))))
```


Konstruktion: Aggregatzustand

Vervollständigen der Bedingungen

Und zum Schluss setzen wir noch die Bedingungen ein, die sich direkt aus der Problemformulierung ergeben:

```
; Aggregatzustand von Wasser bestimmen
(: water-phase (real -> phase))
(define water-phase
  (lambda (t)
    (cond
      ((< t 0) "solid")
      ((and (>= t 0) (<= t 100)) "liquid")
      ((> t 100) "gaseous"))))
```


Noch eine Aufzählungssorte

Betrachten wir nun folgendes Problem:

Aufgabe: Inlandsporti der Deutschen Post

Eingabe: „Postkarte“, „Standardbrief“, „Kompaktbrief“, „Grossbrief“, oder „Maxibrief“

Ausgabe: Das entsprechende Inlandsporto der Deutschen Post

Erste Beobachtung: Die Eingabe ist von einer Sorte, die fünf Literale enthält.
Scheme-Syntax zur Definition dieser Aufzählungssorte:

```
(define postsendung
  (signature
   (one-of
    "Postkarte" "Standardbrief" "Kompaktbrief"
    "Grossbrief" "Maxibrief")))
```


Fallunterscheidungen

Zweite Beobachtung: da es nur fünf mögliche Eingaben gibt, gibt es höchstens fünf mögliche Ausgaben. Und natürlich hängt die Ausgabe von der Eingabe ab. Es ergibt sich eine Fallunterscheidung (Nachschauen der Porti auf `www.deutschepost.de`):

$$porto(s) = \begin{cases} 0.45 & \text{falls } s = \text{„Postkarte“} \\ 0.55 & \text{falls } s = \text{„Standardbrief“} \\ 0.90 & \text{falls } s = \text{„Kompaktbrief“} \\ 1.45 & \text{falls } s = \text{„Grossbrief“} \\ 2.20 & \text{falls } s = \text{„Maxibrief“} \end{cases}$$

Konstruktion: Briefporto bestimmen

Das Gerüst

Zunächst das Gerüst:

```
; Briefporto für verschiedene Briefgröß en ausgeben
(: porto (postsendung -> real))
(define porto
  (lambda (s)
    (...)))
```


Konstruktion: Briefporto bestimmen

Eingabe einsetzen

Dann setzen wir (wie immer) die Eingabe in den Rumpf ein:

```
; Briefporto für verschiedene Briefgröß en ausgeben
(: porto (postsendung -> real))
(define porto
  (lambda (s)
    (... s ...)))
```


Konstruktion: Briefporto bestimmen

Schablone für Fallunterscheidungen

Als nächstes wenden wir eine Schablone an, die sich aus der Existenz von fünf möglichen Eingaben ergibt:

```
; Briefporto für verschiedene Briefgröß en ausgeben
;(define porto
  (lambda (s)
    (... s ...)
    (cond
      ((string=? s "Postkarte") ...)
      ((string=? s "Standardbrief") ...)
      ((string=? s "Kompaktbrief") ...)
      ((string=? s "Grossbrief") ...)
      ((string=? s "Maxibrief") ...)
      )
    ))
```


Konstruktion: Briefporto bestimmen

Einfügen der Ausgaben

Und zum Schluss setzen wir noch die richtigen Ausgaben ein:

```
; Briefporto für verschiedene Briefgröß en ausgeben
(define porto
  (lambda (s)
    (cond
      ((string=? s "Postkarte") 0.45)
      ((string=? s "Standardbrief") 0.55)
      ((string=? s "Kompaktbrief") 0.90)
      ((string=? s "Grossbrief") 1.45)
      ((string=? s "Maxibrief") 2.20)
      )
    ))
```

Datenanalyse

Das Beispiel mit den Inlandsporti ist ein besonders klarer Fall von Datenanalyse: Bloßes Betrachten der möglichen Eingaben (genauer: ihrer Sorte) führt zum Anlegen einer Verzweigung mit einer bestimmten Anzahl von Zweigen.

Die Sorte hilft nicht immer

Betrachten wir nun folgendes Problem:

Absolutbetrag

Eingabe: Eine Zahl x

Ausgabe: Der Absolutbetrag von x

Der Absolutbetrag ist durch folgende Fallunterscheidung definiert (Lehrbuch):

$$|x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{falls } x < 0 \end{cases}$$

Konstruktion: Absolutbetrag

Das Gerüst

Definition

$$|x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{falls } x < 0 \end{cases}$$

Zunächst das Gerüst:

```
; Absolutbetrag einer Zahl berechnen
(: absolute (real -> real))
(define absolute
  (lambda (x)
    (... x ...)))
```

Wir sehen: Die Sorte hilft uns nicht weiter.

Konstruktion: Absolutbetrag

Schablone für Verzweigungen

Definition

$$|x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{falls } x < 0 \end{cases}$$

In diesem Fall empfiehlt es sich, zunächst nur folgendes festzustellen: wir haben eine Fallunterscheidung mit zwei Fällen.

```
; Absolutbetrag einer Zahl berechnen
(: absolute (real -> real))
(define absolute
  (lambda (x)
    (cond
     (... ...)
     (... ...))))
```

Dies festzustellen, nennen wir hier Datenanalyse.

Konstruktion: Absolutbetrag

Einsetzen der Bedingungen und Ausgaben

Definition

$$|x| = \begin{cases} x & \text{falls } x \geq 0 \\ -x & \text{falls } x < 0 \end{cases}$$

Danach kann man die Bedingungen und die Ausgaben einsetzen:

```
; Absolutbetrag einer Zahl berechnen
(: absolute (real -> real))
(define absolute
  (lambda (x)
    (cond
     ((>= x 0)  x)
     ((< x 0)   (- x)))))
```


Konstruktionsanleitung 2: Fallunterscheidung

Allgemeine Konstruktionsanleitung

Falls die Datenanalyse für die Ein- oder Ausgabe einer Prozedur eine Fallunterscheidung in n Kategorien enthält, dann steht im Rumpf der Prozedur eine Verzweigung mit n Zweigen.

Schablone zur Verwendung im Prozedurrumpf

```
(cond
 (... ...)
 ...
 (... ...))
```

(n Zweige)

Konstruktionsanleitung 2: Fallunterscheidung

Spezieller: Ausgabesorte

Wenn sich die Fallunterscheidung direkt aus der Ausgabesorte ergibt, da diese aus n Ausdrücken $e_1, \ldots, e_n$ besteht, kann man einen Konstruktionsschritt überspringen und gleich folgende Schablone verwenden:

```
(cond
 (... e1)
 ...
 (... en))
```

Man muss dies nicht tun, aber es wird dadurch vielleicht klarer, woraus sich die Schablone ergibt.

Konstruktionsanleitung 2: Fallunterscheidung

Spezieller: Eingabesorte

Wenn sich die Fallunterscheidung direkt aus der Sorte einer Eingabe ergibt, da diese aus n Ausdrücken $e_1, \ldots, e_n$ besteht, kann man einen Konstruktionsschritt überspringen und gleich folgende Schablone verwenden:

```
(cond
 ((e1?) ...)
 ...
 ((en?) ...))
```

wobei jedes e_i? ein Ausdruck ist, der die o.g. Eingabe auf Gleichheit mit e_i testet.
Auch hier gilt: Man muss dies nicht tun, aber es wird dadurch vielleicht klarer, woraus sich die Schablone ergibt.

Konstruktionsanleitung 2: Fallunterscheidung

Nochmals allgemein

Wenn sich die Fallunterscheidung nicht direkt aus einer Sorte ergibt, sondern allgemein aus der Problembeschreibung (Datenanalyse), dann verwende zunächst die allgemeine Form:

Schablone zur Verwendung im Prozedurrumpf

```
(cond
 (... ...)
 ...
 (... ...))
```

(n Zweige bei n Kategorien)

Bemerkung: da die Bedingungen in jedem Fall die Eingabe und nicht die Ausgabe betreffen, ist es tendentiell vorzuziehen, die Verzweigung von der Eingabe leiten zu lassen.

Mantra

Mantra #6 (Datenanalyse)

Beginne mit einer Datenanalyse und wähle ausgehend davon die passende Konstruktionsanleitung.

Zur Erinnerung: Das Substitutionsmodell

- Ein Literal ist ein Wert.
- Ein Lambda-Ausdruck ist ein Wert.
- Eine freie Variable wird durch ihre `define`-Bindung (einen Wert) ersetzt.
- Zur Berechnung des Wertes einer Applikation

  $(\langle operator \rangle \ \langle operand \rangle_1 \ \ldots \ \langle operand \rangle_n)$

  werden zuerst der Wert v_0 von $\langle operator \rangle$, sowie die Werte $v_1, \ldots, v_n$ der Operanden bestimmt.

  1. Ist v_0 primitiver Operator, so wird er auf $v_1, \ldots, v_n$ angewendet.
  2. Ist $v_0 =$ `(lambda (`$x_1 \ldots x_n$`)` e`)`, so wird in e jedes freie Vorkommen von x_1 durch v_1, x_2 durch v_2 usw. ersetzt und der Wert des entstehenden Ausdrucks ermittelt.
  3. Andernfalls: Laufzeitfehler!

Verzweigungen sind ein Spezialfall!

Die Semantik von `cond` ist auf spezielle Weise definiert. Insbesondere

- ist es uns nicht egal, in welcher Reihenfolge die Teilausdrücke ausgewertet werden;
- ist es nicht so, dass bei einem Zweig

  $(t_i \quad e_i)$

  der „Operator“ t_i auf den „Operanden“ e_i angewendet würde, wie es laut Substitutionsmodell eigentlich sein müsste. Das würde überhaupt keinen Sinn ergeben!

Wir sagen: Die `cond`-Form ist eine Spezialform.
Wie ist es also? ...

Semantik der Verzweigung

Der Wert des Ausdrucks

```
(cond
  (t_1   e_1)
  ⋮
  (t_n   e_n))
```

wird folgendermaßen bestimmt:

- Zuerst wird t_1 ausgewertet und das Ergebnis ist der Wert von e_1, falls der Wert von t_1 = `#t` ist;
- andernfalls wird t_2 ausgewertet und das Ergebnis ist der Wert von e_2, falls der Wert von t_2 = `#t` ist;
- ...
- andernfalls wird t_n ausgewertet und das Ergebnis ist der Wert von e_n, falls der Wert von t_n = `#t` ist;
- andernfalls ist der Wert des Ausdrucks undefiniert.

Zwei Aspekte von Semantik

- Wir haben in unserer Definition der Semantik zwei Aspekte vermischt:
  - Was ist der Wert einer Verzweigung?
  - Wie wird er berechnet?
- Warum ist die Reihenfolge der Auswertung der Zweige wichtig?
  - Effizienz: Wenn t_i zu `#t` auswertet, ist die Auswertung aller weiteren Zweige eine Ressourcenverschwendung.
  - Bedeutung: der Wert einer Verzweigung ergibt sich immer aus dem Wert des textuell ersten Zweiges, für den sich der Test zu `#t` auswertet. Fortgeschrittene Programmierer nutzen diese Tatsache auch aus.

`else`

Bisherige Form

```
(cond
  (t_1   e_1)
  ⋮
  (t_n-1   e_n-1)
  (t_n   e_n))
```

Form mit `else`

```
(cond
  (t_1   e_1)
  ⋮
  (t_n-1   e_n-1)
  (else   e_n))
```

Für unsere Konstruktionsanleitung 2 bedeutet dies:
Der letzte Zweig darf ein `else`-Zweig sein, falls die vorangegangenen Tests alles außer der letzten Kategorie abdecken.

Semantik mit `else`

Der Wert des Ausdrucks

```
(cond
  (t_1   e_1)
  ⋮
  (t_n-1   e_n-1)
  (else   e_n))
```

wird folgendermaßen bestimmt:

- ...
- andernfalls wird t_{n-1} ausgewertet und das Ergebnis ist der Wert von e_{n-1}, falls der Wert von t_{n-1} = `#t` ist;
- andernfalls ist der Wert des Ausdrucks der Wert von e_n.

Binäre Verzweigungen

Betrachten wir wieder das Programm für den Absolutbetrag (hier mit `else` geschrieben):

```
(: absolute (real -> real))
(define absolute
  (lambda (x)
    (cond
     ((>= x 0) x)
     (else     (- x)))))
```

Hier haben wir eine Verzweigung mit genau zwei Zweigen. Hierfür gibt es eine Spezialsyntax:

```
(define absolute
  (lambda (x)
    (if (>= x 0)
        x
        (- x))))
```


Binäre Verzweigungen allgemein

Im Allgemeinen sieht eine binäre Verzweigung folgendermaßen aus:

`(if` ⟨*test*⟩ ⟨*konsequenz*⟩ ⟨*alternative*⟩`)`

wobei ⟨*test*⟩, ⟨*konsequenz*⟩ und ⟨*alternative*⟩ jeweils Ausdrücke sind, und ⟨*test*⟩ ein Ergebnis der Sorte `boolean` liefern muss.
Auswertungsregel:

- Zuerst Bedingung ⟨*test*⟩ auswerten.
- Ergebnis:
  - Wert von ⟨*konsequenz*⟩, falls Bedingung `#t`
  - Wert von ⟨*alternative*⟩, falls Bedingung `#f`

Binäre Verzweigung: Beispiel für Auswertung

```
   (define x 7)
   (if (not (zero? x)) (/ 1 x) 0)
=> (if (not (zero? 7)) (/ 1 x) 0)
=> (if (not #f) (/ 1 x) 0)
=> (if #t (/ 1 x) 0)
=> (/ 1 x)
=> (/ 1 7)
=> 1/7
```


Simulation von `cond`

Die `cond`-Form ist scheinbar allgemeiner als das `if`. Allerdings lässt sich ein `cond` durch verschachtelte `if`s simulieren:

```
(cond (t1 e1)
      (t2 e2)
      ⋮
      (tn−1 en−1)
      (else en))
```

ist äquivalent zu

```
(if t1 e1
    (if t2 e2
        ...
        (if tn−1 en−1 en)))
```

Wir nennen die `cond`-Form daher eine abgeleitete Form (syntaktischer Zucker).

Milde Temperaturen

Betrachten wir folgendes Problem:

Aufgabe: Milde einer Temperatur feststellen

Eingabe: Lufttemperatur t

Ausgabe: `#t`, falls t zwischen 4 °C und 12 °C liegt, `#f` sonst.

Das Gerüst:

```
; feststellen, ob Temperatur mild ist
(: temperature-mild? (real -> boolean))
```

Bemerkung: Per Konvention hat eine Prozedur mit Ausgabesorte `bool` (boolesche Prozedur) das Suffix `?`, da sie gleichsam eine ja/nein-Frage beantwortet.

Milde Temperaturen: Konstruktion

Aus der Sorte der Ausgabe folgt gemäß Konstruktionsanleitung 2 die Schablone

```
(lambda (t)
  (if ...)
      #t
      #f)))
```

Doch wie sieht der Test aus?
Verwende den Operator logisches Und um „zwischen 4 °C und 12 °C“ auszudrücken.

```
(define temperature-mild?
  (lambda (t)
    (if (and (>= t 4) (<= t 12))
        #t
        #f)))
```


Milde Temperatur ohne `if`

Beobachtung: Der Ausdruck `(and (>= t 4) (<= t 12))` liefert schon das gewünschte Ergebnis:

```
; feststellen, ob Temperatur mild ist
(: temperature-mild? (real -> boolean))
(define temperature-mild?
  (lambda (t)
    (and (>= t 4) (<= t 12))))
```

Das ist allgemein so:

`(if` ⟨*test*⟩ `#t #f)`

ist äquivalent zu ⟨*test*⟩. D.h. für boolesche Prozeduren braucht man gar kein `cond` oder `if`.

Unangenehme Temperaturen

Betrachten wir folgendes Problem:

Aufgabe: Unannehmlichkeit einer Temperatur feststellen

Eingabe: Lufttemperatur t

Ausgabe: `#t`, falls t unter -10 ° oder über 40 ° liegt, `#f` sonst.

Das Gerüst:

```
; feststellen, ob Temperatur unangenehm ist
(: temperature-uncomfortable? (real -> boolean))

; feststellen, ob Temperatur unangenehm
(: temperature-uncomfortable? (real -> boolean))
(define temperature-uncomfortable?
  (lambda (t)
    (or (< t -10) (> t 40))))
```

Konstruktion nach gleichem Prinzip wie `temperature-mild?`

Auswertung logischer Operatoren

Logische Operatoren werden nicht nach der Funktionsanwendungsregel (Substitutionsmodell) ausgewertet, sondern von links nach rechts.

Logisches Und

(and t_1 t_2 ... t_n)

ist äquivalent zu

(if t_1 (if t_2 (... (if t_{n-1} t_n #f)) #f) #f)

Logisches Oder

(or t_1 t_2 ... t_n)

ist äquivalent zu

(if t_1 #t (if t_2 #t (... (if t_{n-1} #t t_n))))

Verbesserte Signatur zu Aggregatzustand

Ein Nachtrag

Erinnern wir uns an das Programm zum Aggregatzustand:

```
; Aggregatzustand von Wasser bestimmen
(: water-phase (real -> phase))
(define water-phase
  (lambda (t)
    (cond
      ((< t 0) "solid")
      ((and (>= t 0) (<= t 100)) "liquid")
      ((> t 100) "gaseous"))))
```


Geändertes/ergänztes Programm

```
; eine Temperatur unter -273.18 Grad ist unmöglich
(: possible-temp? (real -> boolean))
(define possible-temp?
  (lambda (t) (>= t -273.18)))

(: water-phase ((predicate possible-temp?) -> phase))
; weiterer Testfall
(check-expect (water-phase -300.5) "solid")
                          ; Signaturverletzung

(define water-phase
   ...; wie gehabt
```

Solche Signaturverletzungen werden von DrRacket (unter bestimmten Umständen) erkannt.

Signaturverletzungen

- `(water-phase -300.5)` ist nicht etwa verboten/undefiniert, sondern hat den Wert `"solid"`.
- Der Testfall wird demnach auch bestanden, aber es wird wegen der Signaturverletzung eine Warnung ausgegeben.
- Die Erkennung von Signaturverletzungen in DrRacket ist nicht perfekt.
- Doch selbst wenn sie perfekt wäre, was hat man davon?
  Man wird gewarnt, wenn eine Prozedur mit unsinnigen Argumenten aufgerufen wird und demnach möglicherweise unsinnige Ergebnisse produziert. Etwa wenn ein physikalischer Ignorant unser Programm um

  ```
  (define rock-hard
    (water-phase -290))
  ```

  erweitert.

Zusammenfassung

- Die Sorten `boolean` und `string`
- Verzweigungen: `cond` und `if`
- Konstruktionen für Fallunterscheidungen
  - Anhand der Ausgabesorte
  - Anhand der Eingabesorte
  - Ohne Hilfe der Sorte
  - Semantik von `cond` und `if`
- Logische Operatoren und ihre Auswertung: `and`, `or`, `not`