Bisher

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

1.21 Erinnerung: Lexikalische Bindung

```
((lambda (x[1])
   (+ ((lambda (x[2]) (* x[3] 3)) 3)
      (* x[4] 2))) 14)
```

- Zwei Vorkommen von x, ^{1} und ^{2}, sind *bindend* und *definieren* x.
- Zwei Vorkommen, ^{3} und ^{4}, *verwenden* x.
- Frage: Welche Verwendung bezieht sich auf welche Bindung?
- Es gilt die *lexikalische Bindung*:
  Eine Verwendung bezieht sich **immer** auf das bindende Vorkommen der innersten textlich umschließenden Abstraktion.
- D.h. ^{3} bezieht sich auf ^{2} und ^{4} bezieht sich auf ^{1} (Knopf „Syntaxprüfung“).
- Äquivalenter Ausdruck durch *konsistente Umbenennung* eines bindenden Vorkommens und aller Verwendungen dieser Bindung:

```
((lambda (x[1]) (+ ((lambda (y[2]) (* y[3] 3)) 3) (* x[4] 2))) 14)
```

1.22 Zerlegen in Teilprobleme

1.22.1 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)
    (* (disk-area radius) height)))
; Testfall
(check-within (cylinder-volume 1 1) 3.14159 1e-5)
(check-within (cylinder-volume 2 1) (cylinder-volume 1 4) 1e-5)
```

1.23 Berechnungsprozess zu `cylinder-volume`

```
   (cylinder-volume 5 4)
=> ((lambda (radius height) (* (circle-area radius) height)) 5 4)
=> (* (circle-area 5) 4)
=> (* ((lambda (radius) (* pi (square radius))) 5) 4)
=> (* (* pi (square 5)) 4)
=> (* (* 3.141592653589793 ((lambda (x) (* x x)) 5)) 4)
=> (* (* 3.141592653589793 (* 5 5)) 4)
=> (* (* 3.141592653589793 25) 4)
=> (* 78.539816339744825 4)
=> 314.1592653589793
```

Mantra #3 (Strukturerhaltung)

Versuche, das Programm wie das Problem zu strukturieren.

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.

1.24 Definition durch Fallunterscheidung

Viele Funktionen benötigen eine Fallunterscheidung um ihre Eingabedaten in unterschiedliche Klassen aufzuteilen.

1.24.1 Aufgabe: Preis für Tintenpatronen

Eingabe: Geforderter Preis p für eine Tintenpatrone ($p > 0$)

Ausgabe: „dumping“, „ok“ oder „teuer“ je nachdem, ob die Patrone weniger als 5 Euro, weniger als 15 Euro oder darüber kostet.

Die *Datenanalyse* der Eingabedaten erfordert eine Fallunterscheidung.

$$t(p) = \begin{cases} \text{„dumping“} & \text{falls } p < 5 \\ \text{„ok“} & \text{falls } p < 15 \\ \text{„teuer“} & \text{sonst} \end{cases}$$

Diese Notation heißt *Verzweigung* und sie testet *Bedingungen* wie $p < 15$. Eine Bedingung besitzt einen *Wahrheitswert*.

1.24.2 Konstruktion: Preis für Tintenpatronen

```
; Preis für Tintenpatronen einschätzen
(: check-price (real -> (one-of "dumping" "ok" "teuer")))
(define check-price
  (lambda (price)
    ...))
```

Testfälle

```
(check-expect (check-price 1) "dumping")
(check-expect (check-price 5) "ok")
(check-expect (check-price 15) "teuer")
```

Scheme Notation für die gesuchte Verzweigung:

```
(cond
 ((< price 5)  "dumping")
 ((< price 15) "ok")
 (else         "teuer"))
```

1.24.3 Lösung: Preis für Tintenpatronen

```
; Preis für Tintenpatronen einschätzen
(: check-price (real -> (one-of "dumping" "ok" "teuer")))
(define check-price
  (lambda (price)
    (cond
     ((< price 5)  "dumping")
     ((< price 15) "ok")
     (else         "teuer"))))
```

Testfälle

```
(check-expect (check-price 1) "dumping")
(check-expect (check-price 5) "ok")
(check-expect (check-price 15) "teuer")
```

1.25 Der Datentyp `boolean` (Wahrheitswerte)

Literale:

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

Operationen:

- mit Vertrag `(real real -> boolean)`

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

- numerische Prädikate mit Vertrag `(number -> boolean)`

  `zero?` `positive?` `negative?` `odd?` `even?`

- logische Operationen

  `not` `and` `or`

  (`and` und `or` haben eine spezielle Auswertungsregel!)

  Vertrag für `and`, `or`: `(boolean ... -> boolean)`

1.25.1 Ausdrücke vom Typ `boolean`

```
   #t
=> #t

   (= 17 4)
=> #f

   (>= 17 4)
=> #t

   (odd? (+ 17 4))
=> (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
```

1.25.2 Verbesserter Vertrag für Tintenpatronen

```
; Preis für Tintenpatronen einschätzen
(: check-price ((predicate positive?) -> (one-of "dumping" "ok" "teuer")))
(define check-price
  (lambda (price)
    (cond
     ((< price 5)  "dumping")
     ((< price 15) "ok")
     (else         "teuer"))))
; Testfälle
(check-expect (check-price -5) "dumping") ; Vertragsverletzung!
(check-expect (check-price 1) "dumping")
(check-expect (check-price 5) "ok")
(check-expect (check-price 15) "teuer")
```

1.26 Der Datentyp `string` (Zeichenketten)

Eine *Zeichenkette* repräsentiert ein Stück Text.

Literale:

"$c_1 c_2 \ldots c_n$"

Dabei darf c_i ein beliebiges Zeichen außer " sein.
Schreibe \" um " in einer Zeichenkette zu verwenden.

Beispiel:

```
"Ein Neger mit Gazelle zagt im Regen nie."
"Harry schrie: \"Expelliarmo!\""
```

Operationen:

- mit Vertrag `(string string -> boolean)`

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

- uvam, siehe Dokumentation

1.27 Semantik der Verzweigung (Substitutionsmodell)

Falls der auszuwertende Ausdruck die Form einer Verzweigung hat:

```
(cond
  (t_1     e_1)
  (t_2     e_2)
  ⋮
  (t_{n-1}   e_{n-1})
  (else e_n))
```

so wird zuerst t_1 ausgewertet und das Ergebnis ist der Wert von

- e_1, falls der Wert von $t_1 =$ `#t` ist;
- e_2, falls der Wert von $t_1 =$ `#f` ist und der Wert von $t_2 =$ `#t` ist;
- e_i, falls der Wert von $t_1, \ldots, t_{i-1} =$ `#f` ist und der Wert von $t_i =$ `#t` ist;
- e_n, falls der Wert von $t_1, \ldots, t_{n-1} =$ `#f` ist.

1.28 Konstruktionsanleitung: Fallunterscheidung

Falls die Datenanalyse für ein Argument einer Prozedur eine Fallunterscheidung in mehrere Kategorien beinhaltet, dann steht im Rumpf der Prozedur eine Verzweigung. Die Verzweigung hat für jede Kategorie einen Zweig.

Schablone zur Verwendung im Prozedurrumpf einer Prozedur mit Argument a einer Sorte mit n Kategorien:

```
(cond
 (t_1 ...)
 ...
 (t_n ...))
```

Die $t_1, \ldots, t_n$ sind Tests, die a betreffen und die einzelnen Kategorien der Sorte erkennen. Der letzte Zweig darf ein `else`-Zweig sein, falls die vorangegangenen Tests alles außer der letzten Kategorie abdecken.

Mantra #6 (Datenanalyse)

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

1.29 Binäre Verzweigung

Verzweigung mit genau zwei Kategorien: Spezialfall mit einfacherer Scheme-Syntax.

Beispiel:

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

Nach Anleitung:

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

1.29.1 Binäre Verzweigung: If

Spezialsyntax für binäre Verzweigungen

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

⟨*test*⟩, ⟨*konsequenz*⟩ und ⟨*alternative*⟩ sind jeweils Ausdrücke.

⟨*test*⟩ muss ein Ergebnis der Sorte `boolean` liefern.

Beispiel damit

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

1.29.2 Binäre Verzweigung: Auswertungsregel

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

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

```
(if (not (zero? x)) (/ 1 x) 0)
```

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
```

1.29.3 Simulation von `cond`

- `cond` läßt sich durch verschachtelte `if`s simulieren.
- Der Ausdruck

  (`cond` (t_1 e_1)
  (t_2 e_2)
  $\vdots$
  (t_{n-1} e_{n-1})
  (`else` e_n))

  ist äquivalent zu

  (`if` t_1 e_1
  (`if` t_2 e_2
  ...
  (`if` t_{n-1} e_{n-1} e_n)))

- `cond` ist eine *abgeleitete Form*, da `cond` durch `if` simulierbar ist. Anderer Name für eine abgeleitete Form: *Syntaktischer Zucker*.

1.30 Weitere boolesche Operatoren

Aufgabe: Eine Temperatur ist mild (im Winter), falls sie zwischen 4° und 12° liegt.

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

- Beobachtung: es liegen zwei Kategorien vor, wovon eine (Temperatur unter 4° oder über 12°) nicht direkt durch einen Test abgedeckt werden kann.
- Notbehelf: drei Kategorien bilden, $t < 4$, $4 \leq t \leq 12$, $t > 12$
- Konvention: `?` als Suffix für boolesche Prozeduren

1.30.1 Milde Temperatur mit drei Kategorien

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

- unschön
- Nächster Versuch: Zwei geschachtelte Aufteilungen in zwei Kategorien

1.30.2 Milde Temperatur mit geschachteltem `if`

```
; feststellen, ob Temperatur mild ist
(: temperature-mild? (real -> boolean))
(define temperature-mild?
  (lambda (t)
    (if (>= t 4)
        ...
        #f)))
```

Bei `...` ist klar, dass die Temperatur nicht zu kalt ist, also muss dort eine weitere binäre Verzweigung eingebaut werden.

1.30.3 Milde Temperatur mit geschachteltem `if`, II

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

- unschön
- Nächster Versuch: Verwende den Operator *logisches Und* um die Bedingungen zusammenzufassen

1.30.4 Milde Temperatur mit logischem Und

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

1.30.5 Milde Temperatur mit logischem Und, II

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))))
```

1.30.6 Beispiel zum logischen Oder

Eine Temperatur ist unangenehm, falls sie unter -10° *oder* über 40° liegt.

```
; 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?`

1.30.7 Logische Operatoren

Logische Operatoren werden **nicht** nach der Funktionsanwendungsregel 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))))

Elegantere Formulierung

Logisches Und

$$\texttt{(and)} \equiv \texttt{\#t}$$
$$\texttt{(and } t_1\ t_2\ \texttt{...)} \equiv \texttt{(if } t_1\ \texttt{(and } t_2\ \texttt{...) \#f)}$$

Logisches Oder

$$\texttt{(or)} \equiv \texttt{\#f}$$
$$\texttt{(or } t_1\ t_2\ \texttt{...)} \equiv \texttt{(if } t_1\ \texttt{\#t (or } t_2\ \texttt{...))}$$

1.31 Zusammenfassung

- Konstruktionsanleitung für Fallunterscheidungen
- Datentypen `boolean` und `string`
- Vergleichsoperatoren
- Verzweigungen: `cond` und `if`
- Logische Operatoren und ihre Auswertung: `and`, `or`, `not`