

Bisher

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

2.21 Erinnerung: Lexikalische Bindung

$((\text{lambda } (x^1) (+ ((\text{lambda } (x^2) (* x^3 3)) 3) (* x^4 2))) 14)$

- Vier Vorkommen von x
- Die ersten beiden, 1 und 2 , sind *bindend*.
- Die restlichen beiden, 3 und 4 , sind *gebunden*.
- Es gilt die *lexikalische Bindung*:
Ein gebundenes Vorkommen 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:

$((\text{lambda } (x^1) (+ ((\text{lambda } (y^2) (* y^3 3)) 3) (* x^4 2))) 14)$

2.22 Definition durch Fallunterscheidung

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

2.22.1 Aufgabe: Preis für Tintenpatronen

Eingabe: Geforderter Preis p für eine Tintenpatrone

Ausgabe: „Betrug“, „kaufen“ 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.

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

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

2.22.2 Konstruktion: Preis für Tintenpatronen

```
; Preis für Tintenpatronen einschätzen
; check-price : number -> {"Betrug", "kaufen", "teuer"}
(define check-price
  (lambda (price)
    ...))
```

Testfälle

```
; (check-price 1) => "Betrug"
; (check-price 5) => "kaufen"
; (check-price 15) => "teuer"
```

Scheme Notation für die gesuchte Verzweigung:

```
(cond
  ((< price 5) "Betrug")
  ((< price 15) "kaufen")
  (else "teuer"))
```

2.22.3 Lösung: Preis für Tintenpatronen

```
; Preis für Tintenpatronen einschätzen
; check-price : number -> {"Betrug", "kaufen", "teuer"}
(define check-price
  (lambda (price)
    (cond
      ((< price 5) "Betrug")
      ((< price 15) "kaufen")
      (else "teuer"))))
```

Testfälle

```
; (check-price 1) => "Betrug"
; (check-price 5) => "kaufen"
; (check-price 15) => "teuer"
```

2.23 Der Datentyp `boolean` (Wahrheitswerte)

Literale:

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

Operationen:

- mit Vertrag `number number -> boolean`
`=` `<` `>` `>=` `<=`
- numerische Prädikate mit Vertrag `number -> boolean`
`zero?` `positive?` `negative?` `odd?` `even?`
- logische Operationen
`not` `and` `or`
(`and` und `or` haben eine spezielle Auswertungsregel!)

2.24 Ausdrücke vom Typ boolean

```
#t (define y 1)
=> #t (and (= 5 (+ (* 2 2) y)) (and (<= 0 y) (< y 2)))
    => (and (= 5 (+ 4 y)) (and (<= 0 y) (< y 2)))
    (= 17 4)
    => (and (= 5 (+ 4 1)) (and (<= 0 y) (< y 2)))
=> #f => (and (= 5 5) (and (<= 0 y) (< y 2)))
    => (and #t (and (<= 0 y) (< y 2)))
    (>= 17 4)
=> #t => (and (<= 0 y) (< y 2))
    => (and (<= 0 1) (< y 2))
    => (and #t (< y 2))
    (odd? (+ 17 4))
=> (odd? 21)
=> #t => (< y 2)
    => (< 1 2)
    => #t
```

2.25 Der Datentyp string (Zeichenketten)

Eine *Zeichenkette* repräsentiert ein Stück Text.

Literale:

" $c_1c_2 \dots c_n$ "

Dabei darf c_i jedes Zeichen sein, außer " selbst.

Schreibe \" um " in einer Zeichenkette zu verwenden.

Beispiel:

"Ein Neger mit Gazelle zagt im Regen nie."

"Harry schrie: \"Expelliarmo!\""

Operationen:

- mit Vertrag `number number -> boolean`
`string=?` `string<?` `string>?`
- uvam

2.26 Semantik der Verzweigung

(Erweiterung des Substitutionsmodells)

Falls der auszuwertende Ausdruck die Form einer Verzweigung hat:

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

dann ist das Ergebnis 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 $t_1, \dots, t_{i-1} = \#f$ ist und der Wert von $t_i = \#t$ ist;
- e_n , falls der Wert von $t_1, \dots, t_{n-1} = \#f$ ist.

2.27 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, \dots, t_n müssen Tests sein, 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

Mantra #6 (Datenanalyse)

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

2.28 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 : number -> number
```

```
(define absolute
```

```
  (lambda (x)
```

```
    (cond
```

```
      ((>= x 0) x)
```

```
      (else (- x))))))
```

2.28.1 Binäre Verzweigung: If

Spezialsyntax für binäre Verzweigungen

```
(if <test> <konsequenz> <alternative>)
```

<test>, <konsequenz> und <alternative> sind jeweils Ausdrücke. <test> sollte ein Ergebnis der Sorte boolean liefern.

Beispiel damit

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

2.28.2 Bedingte 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
```

2.28.3 Simulation von cond

- cond läßt sich durch verschachtelte ifs simulieren.
- Der Ausdruck

```
(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)))
```

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

2.29 Weitere boolesche Operatoren

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

; feststellen, ob Temperatur mild ist

; `temperature-mild? : number -> 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
- Konvention: ? als Suffix für boolesche Prozeduren

2.29.1 Milde Temperatur mit drei Kategorien

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

- unschön
- Nächster Versuch: Zweimal Aufteilung in zwei Kategorien

2.29.2 Milde Temperatur mit geschachteltem if

```
; feststellen, ob Temperatur mild ist
; temperature-mild? : number -> 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.

2.29.3 Milde Temperatur mit geschachteltem if, II

```
; feststellen, ob Temperatur mild ist
; temperature-mild? : number -> 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

2.29.4 Milde Temperatur mit logischem Und

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

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

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

2.29.5 Beispiel zum logischen Oder

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

```
; feststellen, ob Temperatur unangenehm
```

```
; temperature-uncomfortable? : number -> boolean
```

```
(define temperature-uncomfortable?
```

```
  (lambda (t)
```

```
    (or (< t -10) (> t 40))))
```

- Konstruktion nach gleichem Prinzip wie temperature-mild?

2.29.6 Logische Operatoren

Logische Operatoren werden **nicht** nach der Funktionsanwendungsregel ausgewertet, sondern von links nach rechts.

Logisches Und

$$(\text{and } t_1 t_2 \dots t_n)$$

ist äquivalent zu

$$(\text{if } t_1 (\text{if } t_2 (\dots (\text{if } t_{n-1} t_n \text{ #f})) \text{ #f}) \text{ #f})$$

Logisches Oder

$$(\text{or } t_1 t_2 \dots t_n)$$

ist äquivalent zu

$$(\text{if } t_1 \text{ #t } (\text{if } t_2 \text{ #t } (\dots (\text{if } t_{n-1} \text{ #t } t_n))))$$

Elegantere Formulierung

Logisches Und

`(and)` \equiv `#t`

`(and t1 t2 ...)` \equiv `(if t1 (and t2 ...) #f)`

Logisches Oder

`(or)` \equiv `#f`

`(or t1 t2 ...)` \equiv `(if t1 #t (or t2 ...))`

2.30 Zusammenfassung

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