Sprachebenen und Material zu *Die Macht der Abstraktion*

Version 6.1

August 1, 2014

Note: This is documentation for the teachpacks that go with the German textbook *Die Macht der Abstraktion*.

Das Material in diesem Handbuch ist für die Verwendung mit dem Buch the *Die Macht der Abstraktion* gedacht.

Contents

1 Die Macht der Abstraktion - Anfänger **5**
- 1.1 Definitionen . . . 8
- 1.2 Record-Typ-Definitionen . . . 8
- 1.3 Prozedurapplikation . . . 8
- 1.4 `#t` and `#f` . . . 9
- 1.5 `lambda` . . . 9
- 1.6 Bezeichner . . . 9
- 1.7 `cond` . . . 9
- 1.8 `if` . . . 10
- 1.9 `and` . . . 10
- 1.10 `or` . . . 10
- 1.11 Signaturen . . . 10
  - 1.11.1 `signature` . . . 11
  - 1.11.2 Signaturdeklaration . . . 11
  - 1.11.3 Eingebaute Signaturen . . . 11
  - 1.11.4 `predicate` . . . 12
  - 1.11.5 `one-of` . . . 12
  - 1.11.6 `mixed` . . . 13
  - 1.11.7 Prozedur-Signatur . . . 13
  - 1.11.8 Signatur-Variablen . . . 13
  - 1.11.9 `combined` . . . 13
- 1.12 Testfälle . . . 14
- 1.13 Parametrische Record-Typ-Definitionen . . . 14
- 1.14 Eigenschaften . . . 15
- 1.15 Primitive Operationen . . . 16

2 Die Macht der Abstraktion **27**
- 2.1 Signaturen . . . 30
  - 2.1.1 `list-of` . . . 30
- 2.2 `let`, `letrec` und `let*` . . . 31
- 2.3 Primitive Operationen . . . 31

3 Die Macht der Abstraktion mit Zuweisungen **44**
- 3.1 `define-record-procedures-2` . . . 48
- 3.2 `define-record-procedures-parametric-2` . . . 48
- 3.3 `begin` . . . 48
- 3.4 `set!` . . . 48
- 3.5 Signaturen . . . 48
- 3.6 Definitionen . . . 49
- 3.7 `lambda` . . . 49
- 3.8 Primitive Operationen . . . 49

4 Die Macht der Abstraktion fortgeschritten **61**
4.1 Quote-Literal . . . 65
4.2 Signaturen . . . 65
4.3 Primitive Operationen . . . 65

5 Konstruktionsanleitungen 1 bis 10 **78**
5.1 Konstruktion von Prozeduren . . . 80
5.2 Fallunterscheidung . . . 80
5.3 zusammengesetzte Daten . . . 81
5.4 zusammengesetzte Daten als Argumente . . . 81
5.5 zusammengesetzte Daten als Ausgabe . . . 82
5.6 gemischte Daten . . . 82
5.7 Listen . . . 83
5.8 natürliche Zahlen . . . 83
5.9 Prozeduren mit Akkumulatoren . . . 84
5.10 gekapselter Zustand . . . 86

6 Bilder konstruieren: `"image.ss"` **87**
6.1 Bilder . . . 87
6.2 Modi und Farben . . . 87
6.3 Einfache geometrische Figuren . . . 89
6.4 Eigenschaften von Bildern . . . 90
6.5 Bilder zusammensetzen . . . 91
6.6 Bilder aus Pixeln konstruieren . . . 92

7 Animationen: `"world.ss"` **93**

8 Turtle-Grafik: `"turtle.ss"` **95**
8.1 Tutorial . . . 95
8.1.1 Verbesserungen . . . 98
8.2 Prozeduren . . . 99

9 Abspielen von Audio-Dateien: `"sound.ss"` **101**

10 3D-Liniengraphik: `"line3d.ss"` **102**
10.1 Szenen erzeugen . . . 102
10.2 3D-Vektoren . . . 102
10.3 4D-Vektoren . . . 104
10.4 4x4 Matrizen . . . 105
10.5 3d-Linien . . . 106

11 DMdA: Sprachen als Libraries **108**
11.1 *Die Macht der Abstraktion* - Anfänger . . . 108
11.2 *Die Macht der Abstraktion* . . . 108
11.3 *Die Macht der Abstraktion* mit Zuweisungen . . . 108
11.4 *Die Macht der Abstraktion* - fortgeschritten . . . 108

Index **109**

Index **109**

1 Die Macht der Abstraktion - Anfänger

This is documentation for the language level *Die Macht der Abstraktion - Anfänger* to go with the German textbook *Die Macht der Abstraktion*.

```
     program = def-or-expr ...

 def-or-expr = definition
             | expr
             | test-case

  definition = (define id expr)
             | (define-record-procedures id id id (id ...))
             | (define-record-procedures-parametric (id id ...) id id (id ...))
             | (: id sig)

        expr = (expr expr ...) ; Prozedurapplikation
             | #t
             | #f
             | number
             | string
             | (lambda (id ...) expr)
             | id ; Bezeichner
             | (cond (expr expr) (expr expr) ...)
             | (cond (expr expr) ... (else expr))
             | (if expr expr)
             | (and expr ...)
             | (or expr ...)
             | (let ((id expr) ...) expr)
             | (letrec ((id expr) ...) expr)
             | (let* ((id expr) ...) expr)
             | (begin expr expr ...)
             | (signature sig)
             | (for-all ((id sig) ...) expr)
             | (==> expr expr)

         sig = id
             | (predicate expr)
             | (one-of expr ...)
             | (mixed sig ...)
             | (sig ... -> sig) ; Prozedur-Signatur
             | (list sig)
             | %a %b %c ; Signatur-Variable
             | (combined sig ...)
             | signature
```

```
test-case = (check-expect expr expr)
          | (check-within expr expr expr)
          | (check-member-of expr expr ...)
          | (check-range expr expr expr)
          | (check-error expr expr)
          | (check-property expr)
```

Ein *id* ist eine Folge von Zeichen, die weder Leerzeichen noch eins der folgenden Zeichen enthält:

```
" , ' ` ( ) [ ] { } | ; #
```

Ein *number* ist eine Zahl wie z.B. `123`, `3/2` oder `5.5`.

Ein *string* ist eine Zeichenkette, und durch ein Paar von `"` umschlossen. So sind z.B. `"abcdef"`, `"This is a string"` und `"Dies ist eine Zeichenkette, die \" enthält."` Zeichenketten.

Zahlen

```
* : (number number number ... -> number)
+ : (number number number ... -> number)
- : (number number ... -> number)
/ : (number number number ... -> number)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (number number number ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real real ... -> boolean)
abs : (real -> real)
acos : (number -> number)
angle : (number -> real)
asin : (number -> number)
atan : (number -> number)
ceiling : (real -> integer)
complex? : (any -> boolean)
cos : (number -> number)
current-seconds : (-> natural)
denominator : (rational -> natural)
even? : (integer -> boolean)
exact->inexact : (number -> number)
exact? : (number -> boolean)
exp : (number -> number)
expt : (number number -> number)
floor : (real -> integer)
gcd : (integer integer ... -> natural)
```

```
imag-part : (number -> real)
inexact->exact : (number -> number)
inexact? : (number -> boolean)
integer? : (any -> boolean)
lcm : (integer integer ... -> natural)
log : (number -> number)
magnitude : (number -> real)
make-polar : (real real -> number)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (integer integer -> integer)
natural? : (any -> boolean)
negative? : (number -> boolean)
number->string : (number -> string)
number? : (any -> boolean)
numerator : (rational -> integer)
odd? : (integer -> boolean)
positive? : (number -> boolean)
quotient : (integer integer -> integer)
random : (natural -> natural)
rational? : (any -> boolean)
real-part : (number -> real)
real? : (any -> boolean)
remainder : (integer integer -> integer)
round : (real -> integer)
sin : (number -> number)
sqrt : (number -> number)
string->number : (string -> (mixed number false))
tan : (number -> number)
zero? : (number -> boolean)
```

boolesche Werte

```
boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
true? : (any -> boolean)
```

Listen

```
cons : (%a (list-of %a) -> (list-of %a))
empty : list
empty? : (any -> boolean)
```

Computer

```
computer : signature
computer-hard-drive : (computer -> rational)
computer-processor : (computer -> string)
computer-ram : (computer -> rational)
computer? : (any -> boolean)
```

```
make-computer : (string rational rational -> computer)
```
Schokokekse
```
chocolate-cookie : signature
chocolate-cookie-chocolate : (chocolate-cookie -> number)
chocolate-cookie-cookie : (chocolate-cookie -> number)
chocolate-cookie? : (any -> boolean)
make-chocolate-cookie : (number number -> chocolate-cookie)
```
Zeichenketten
```
string->strings-list : (string -> (list string))
string-append : (string string ... -> string)
string-length : (string -> natural)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (string string string ... -> boolean)
string? : (any -> boolean)
strings-list->string : ((list string) -> string)
```
Symbole

Verschiedenes
```
read : (-> any)
violation : (string -> unspecific)
write-newline : (-> unspecific)
write-string : (string -> unspecific)
```

1.1 Definitionen

```
(define id expr)
```

Diese Form ist eine Definition, und bindet *id* als globalen Namen an den Wert von exp.

1.2 Record-Typ-Definitionen

```
(define-record-procedures t c p (s1 ...))
```

Die define-record-procedures-Form ist eine Definition für einen neuen Record-Typ. Dabei ist *t* der Name der Record-Signatur, *c* der Name des Konstruktors, *p* der Name des Prädikats, und die si sind die Namen der Selektoren.

1.3 Prozedurapplikation

```
(expr expr ...)
```

Dies ist eine Prozeduranwendung oder Applikation. Alle *expr*s werden ausgewertet: Der Operator (also der erste Ausdruck) muß eine Prozedur ergeben, die genauso viele Argumente akzeptieren kann, wie es Operanden, also weitere *expr*s gibt. Die Anwendung wird dann ausgewertet, indem der Rumpf der Applikation ausgewertet wird, nachdem die Parameter der Prozedur durch die Argumente, also die Werte der Operanden ersetzt wurden.

1.4 `#t` and `#f`

`#t` ist das Literal für den booleschen Wert "wahr", `#f` das Literal für den booleschen Wert "falsch".

1.5 `lambda`

```
(lambda (id ...) expr)
```

Ein Lambda-Ausdruck ergibt bei der Auswertung eine neue Prozedur.

1.6 Bezeichner

```
id
```

Eine Variable bezieht sich auf die, von innen nach außen suchend, nächstgelegene Bindung durch `lambda`, `let`, `letrec`, oder `let*`. Falls es keine solche lokale Bindung gibt, muß es eine Definition oder eine eingebaute Bindung mit dem entsprechenden Namen geben. Die Auswertung des Namens ergibt dann den entsprechenden Wert.

1.7 `cond`

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

Ein `cond`-Ausdruck bildet eine Verzweigung, die aus mehreren Zweigen besteht. Jeder Zweig besteht aus einem Test und einem Ausdruck. Bei der Auswertung werden die Zweige nacheinander abgearbeitet. Dabei wird jeweils zunächst der Test ausgewertet, der jeweils einen booleschen Wert ergeben müssen. Beim ersten Test, der `#t` ergibt, wird der Wert des Ausdrucks des Zweigs zum Wert der gesamten Verzweigung. Wenn kein Test `#t` ergibt, wird das Programm mit einer Fehlermeldung abgebrochen.

```
(cond (expr expr) ... (else expr))
```

Die Form des cond-Ausdrucks ist ähnlich zur vorigen, mit der Ausnahme, daß in dem Fall, in dem kein Test `#t` ergibt, der Wert des letzten Ausdruck zum Wert der `cond`-Form wird.

```
else
```

Das Schlüsselwort `else` kann nur in `cond` benutzt werden.

1.8 `if`

```
(if expr expr expr)
```

Eine `if`-Form ist eine binäre Verzweigung. Bei der Auswertung wird zunächst der erste Operand ausgewertet (der Test), der einen booleschen Wert ergeben muß. Ergibt er `#t`, wird der Wert des zweiten Operanden (die Konsequente) zum Wert der `if`-Form, bei `#f` der Wert des dritten Operanden (die Alternative).

1.9 `and`

```
(and expr ...)
```

Bei der Auswertung eines `and`-Ausdrucks werden nacheinander die Operanden (die boolesche Werte ergeben müssen) ausgewertet. Ergibt einer `#f`, ergibt auch der and-Ausdruck `#f`; wenn alle Operanden `#t` ergeben, ergibt auch der `and`-Ausdruck `#t`.

1.10 `or`

```
(or expr ...)
```

Bei der Auswertung eines `or`-Ausdrucks werden nacheinander die Operanden (die boolesche Werte ergeben müssen) ausgewertet. Ergibt einer `#t`, ergibt auch der or-Ausdruck `#t`; wenn alle Operanden `#f` ergeben, ergibt auch der `or`-Ausdruck `#f`.

1.11 Signaturen

Signaturen können statt der Verträge aus dem Buch geschrieben werden: Während Verträge reine Kommentare sind, überprüft DrRacket Signaturen und meldet etwaige Verletzungen.

1.11.1 `signature`

```
(signature sig)
```

Diese Form liefert die Signatur mit der Notation *sig*.

1.11.2 Signaturdeklaration

```
(: id sig)
```

Diese Form erklärt *sig* zur gültigen Signatur für *id*.

1.11.3 Eingebaute Signaturen

```
number
```

Signatur für beliebige Zahlen.

```
real
```

Signatur für reelle Zahlen.

```
rational
```

Signatur für rationale Zahlen.

```
integer
```

Signatur für ganze Zahlen.

```
natural
```

Signatur für ganze, nichtnegative Zahlen.

```
boolean
```

Signatur für boolesche Werte.

```
true
```

Signatur für \scheme[#t].

```
false
```

Signatur für \scheme[#f].

```
string
```

Signatur für Zeichenketten.

```
empty-list
```

Signatur für die leere Liste.

```
any
```

Signatur, die auf alle Werte gültig ist.

```
signature
```

Signatur für Signaturen.

```
property
```

Signatur für Eigenschaften.

1.11.4 `predicate`

```
(predicate expr)
```

Bei dieser Signatur muß *expr* als Wert ein Prädikat haben, also eine Prozedur, die einen beliebigen Wert akzeptiert und entweder `#t` oder `#f` zurückgibt. Die Signatur ist dann für einen Wert gültig, wenn das Prädikat, darauf angewendet, `#t` ergibt.

1.11.5 `one-of`

```
(one-of expr ...)
```

Diese Signatur ist für einen Wert gültig, wenn er gleich dem Wert eines der *expr* ist.

1.11.6 `mixed`

```
(mixed sig ...)
```

Diese Signatur ist für einen Wert gültig, wenn er für eine der Signaturen *sig* gültig ist.

1.11.7 Prozedur-Signatur

```
->
```

```
(sig ... -> sig)
```

Diese Signatur ist dann für einen Wert gültig, wenn dieser eine Prozedur ist. Er erklärt außerdem, daß die Signaturen vor dem `->` für die Argumente der Prozedur gelten und die Signatur nach dem `->` für den Rückgabewert. }

1.11.8 Signatur-Variablen

```
%a
```

```
%b
```

```
%c
```

```
...
```

Dies ist eine Signaturvariable: sie steht für eine Signatur, die für jeden Wert gültig ist.

1.11.9 `combined`

```
(combined sig ...)
```

Diese Signatur ist für einen Wert gültig, wenn sie für alle der Signaturen *sig* gültig ist.

1.12 Testfälle

```
(check-expect expr expr)
```

Dieser Testfall überprüft, ob der erste *expr* den gleichen Wert hat wie der zweite *expr*, wobei das zweite *expr* meist ein Literal ist.

```
(check-within expr expr expr)
```

Wie `check-expect`, aber mit einem weiteren Ausdruck, der als Wert eine Zahl *delta* hat. Der Testfall überprüft, daß jede Zahl im Resultat des ersten *expr* maximal um *delta* von der entsprechenden Zahl im zweiten *expr* abweicht.

```
(check-member-of expr expr ...)
```

Ähnlich wie `check-expect`: Der Testfall überprüft, daß das Resultat des ersten Operanden gleich dem Wert eines der folgenden Operanden ist.

```
(check-range expr expr expr)
```

Ähnlich wie `check-expect`: Alle drei Operanden müssen Zahlen sein. Der Testfall überprüft, ob die erste Zahl zwischen der zweiten und der dritten liegt (inklusive).

```
(check-error expr expr)
```

Dieser Testfall überprüft, ob der erste *expr* einen Fehler produziert, wobei die Fehlermeldung der Zeichenkette entspricht, die der Wert des zweiten *expr* ist.

```
(check-property expr)
```

Dieser Testfall überprüft experimentell, ob die Eigenschaft *expr* erfüllt ist. Dazu werden zufällige Werte für die mit `for-all` quantifizierten Variablen eingesetzt: Damit wird überprüft, ob die Bedingung gilt.

Wichtig: `check-property` funktioniert nur für Eigenschaften, bei denen aus den Signaturen sinnvoll Werte generiert werden können. Dies ist für die meisten eingebauten Signaturen der Fall, aber nicht für Signaturvariablen und Signaturen, die mit `predicate` oder `define-record-procedures` definiert wurden - wohl aber für Signaturen, die mit dem durch `define-record-procedures-parametric` definierten Signaturkonstruktor erzeugt wurden.

1.13 Parametrische Record-Typ-Definitionen

```
(define-record-procedures-parametric t cc c p (s1 ...))
```

Die `define-record-procedures-parametric` ist wie `define-record-procedures`. Zusäzlich wird der Bezeichner *cc* an einen Signaturkonstruktor gebunden: Dieser akzeptiert für jedes Feld eine Feld-Signatur und liefert eine Signatur, die nur Records des Record-Typs *t* erfüllen, bei dem die Feldinhalte die Feld-Signaturen erfüllen.

Beispiel:

```
(define-record-procedures-parametric pare pare-of
  make-pare pare?
  (pare-one pare-two))
```

Dann ist `(pare-of integer string)` die Signatur für `pare`-Records, bei dem die Feldinhalte die Signaturen `integer` bzw. `string` erfüllen müssen.

Die Signaturen für die Feldinhalte werden erst überprüft, wenn ein Selektor aufgerufen wird.

1.14 Eigenschaften

Eine *Eigenschaft* definiert eine Aussage über einen Scheme-Ausdruck, die experimentell überprüft werden kann. Der einfachste Fall einer Eigenschaft ist ein boolescher Ausdruck. Die folgende Eigenschaft gilt immer:

```
(= 1 1)
```

Es ist auch möglich, in einer Eigenschaft Variablen zu verwenden, für die verschiedene Werte eingesetzt werden. Dafür müssen die Variablen gebunden und *quantifiziert* werden, d.h. es muß festgelegt werden, welche Signatur die Werte der Variable erfüllen sollen. Eigenschaften mit Variablen werden mit der `for-all`-Form erzeugt:

```
(for-all ((id sig) ...) expr)
```

Dies bindet die Variablen *id* in der Eigenschaft *expr*. Zu jeder Variable gehört eine Signatur *sig*, der von den Werten der Variable erfüllt werden muß.

Beispiel:

```
(for-all ((x integer))
  (= x (/ (* x 2) 2)))
```

```
(expect expr expr)
```

Ein `expect`-Ausdruck ergibt eine Eigenschaft, die dann gilt, wenn die Werte von *expr* und *expr* gleich sind, im gleichen Sinne wie bei `check-expect`.

```
(expect-within expr expr expr)
```

Wie `expect`, aber entsprechend `check-within` mit einem weiteren Ausdruck, der als Wert eine Zahl *`delta`* hat. Die resultierende Eigenschaft gilt, wenn jede Zahl im Resultat des ersten *`expr`* maximal um *`delta`* von der entsprechenden Zahl im zweiten *`expr`* abweicht.

```
(expect-member-of expr expr ...)
```

Wie `expect`, aber entsprechend `check-member-of` mit weiteren Ausdrücken, die mit dem ersten verglichen werden. Die resultierende Eigenschaft gilt, wenn das erste Argument gleich einem der anderen Argumente ist.

```
(expect-range expr expr expr)
```

Wie `expect`, aber entsprechend `check-range`: Die Argumente müssen Zahlen sein. Die Eigenschaft gilt, wenn die erste Zahl zwischen der zweiten und dritten Zahl liegt (inklusive).

```
(==> expr expr)
```

Der erste Operand ist ein boolescher Ausdruck, der zweite Operand eine Eigenschaft: `(==> c p)` legt fest, daß die Eigenschaft `p` nur erfüllt sein muß, wenn `c` (die *Bedingung*) `#t` ergibt, also erfüllt ist.

```
(for-all ((x integer))
  (==> (even? x)
       (= x (* 2 (/ x 2)))))
```

1.15 Primitive Operationen

```
* : (number number number ... -> number)
```

Produkt berechnen

```
+ : (number number number ... -> number)
```

Summe berechnen

```
- : (number number ... -> number)
```

bei mehr als einem Argument Differenz zwischen der ersten und der Summe aller weiteren Argumente berechnen; bei einem Argument Zahl negieren

```
/ : (number number number ... -> number)
```

das erste Argument durch das Produkt aller weiteren Argumente berechnen

```
< : (real real real ... -> boolean)
```

Zahlen auf kleiner-als testen

```
<= : (real real real ... -> boolean)
```

Zahlen auf kleiner-gleich testen

```
= : (number number number ... -> boolean)
```

Zahlen auf Gleichheit testen

```
> : (real real real ... -> boolean)
```

Zahlen auf größer-als testen

```
>= : (real real real ... -> boolean)
```

Zahlen auf größer-gleich testen

```
abs : (real -> real)
```

Absolutwert berechnen

```
acos : (number -> number)
```

Arcuscosinus berechnen (in Radian)

```
angle : (number -> real)
```

Winkel einer komplexen Zahl berechnen

```
asin : (number -> number)
```

Arcussinus berechnen (in Radian)

```
atan : (number -> number)
```

Arcustangens berechnen (in Radian)

```
ceiling : (real -> integer)
```

nächste ganze Zahl oberhalb einer rellen Zahlen berechnen

```
complex? : (any -> boolean)
```

feststellen, ob ein Wert eine komplexe Zahl ist

```
cos : (number -> number)
```

Cosinus berechnen (Argument in Radian)

```
current-seconds : (-> natural)
```

aktuelle Zeit in Sekunden seit einem unspezifizierten Startzeitpunkt berechnen

```
denominator : (rational -> natural)
```

Nenner eines Bruchs berechnen

```
even? : (integer -> boolean)
```

feststellen, ob eine Zahl gerade ist

```
exact->inexact : (number -> number)
```

eine Zahl durch eine inexakte Zahl annähern

```
exact? : (number -> boolean)
```

feststellen, ob eine Zahl exakt ist

```
exp : (number -> number)
```

Exponentialfunktion berechnen (e hoch Argument)

```
expt : (number number -> number)
```

Potenz berechnen (erstes Argument hoch zweites Argument)

```
floor : (real -> integer)
```

nächste ganze Zahl unterhalb einer rellen Zahlen berechnen

```
gcd : (integer integer ... -> natural)
```

größten gemeinsamen Teiler berechnen

```
imag-part : (number -> real)
```

imaginären Anteil einer komplexen Zahl extrahieren

```
inexact->exact : (number -> number)
```

eine Zahl durch eine exakte Zahl annähern

```
inexact? : (number -> boolean)
```

feststellen, ob eine Zahl inexakt ist

```
integer? : (any -> boolean)
```

feststellen, ob ein Wert eine ganze Zahl ist

```
lcm : (integer integer ... -> natural)
```

kleinstes gemeinsames Vielfaches berechnen

```
log : (number -> number)
```

natürlichen Logarithmus (Basis e) berechnen

```
magnitude : (number -> real)
```

Abstand zum Ursprung einer komplexen Zahl berechnen

```
make-polar : (real real -> number)
```

komplexe Zahl aus Abstand zum Ursprung und Winkel berechnen

```
max : (real real ... -> real)
```

Maximum berechnen

```
min : (real real ... -> real)
```

Minimum berechnen

```
modulo : (integer integer -> integer)
```

Divisionsmodulo berechnen

```
natural? : (any -> boolean)
```

feststellen, ob ein Wert eine natürliche Zahl (inkl. 0) ist

```
negative? : (number -> boolean)
```

feststellen, ob eine Zahl negativ ist

```
number->string : (number -> string)
```

Zahl in Zeichenkette umwandeln

```
number? : (any -> boolean)
```

feststellen, ob ein Wert eine Zahl ist

```
numerator : (rational -> integer)
```

Zähler eines Bruchs berechnen

```
odd? : (integer -> boolean)
```

feststellen, ob eine Zahl ungerade ist

```
positive? : (number -> boolean)
```

feststellen, ob eine Zahl positiv ist

```
quotient : (integer integer -> integer)
```

ganzzahlig dividieren

```
random : (natural -> natural)
```

eine natürliche Zufallszahl berechnen, die kleiner als das Argument ist

```
rational? : (any -> boolean)
```

feststellen, ob eine Zahl rational ist

```
real-part : (number -> real)
```

reellen Anteil einer komplexen Zahl extrahieren

```
real? : (any -> boolean)
```

feststellen, ob ein Wert eine reelle Zahl ist

```
remainder : (integer integer -> integer)
```

Divisionsrest berechnen

```
round : (real -> integer)
```

relle Zahl auf eine ganze Zahl runden

```
sin : (number -> number)
```

Sinus berechnen (Argument in Radian)

```
sqrt : (number -> number)
```

Quadratwurzel berechnen

```
string->number : (string -> (mixed number false))
```

Zeichenkette in Zahl umwandeln, falls möglich

```
tan : (number -> number)
```

Tangens berechnen (Argument in Radian)

```
zero? : (number -> boolean)
```

feststellen, ob eine Zahl Null ist

```
boolean=? : (boolean boolean -> boolean)
```

Booleans auf Gleichheit testen

```
boolean? : (any -> boolean)
```

feststellen, ob ein Wert ein boolescher Wert ist

```
false? : (any -> boolean)
```

feststellen, ob ein Wert #f ist

```
not : (boolean -> boolean)
```

booleschen Wert negieren

```
true? : (any -> boolean)
```

feststellen, ob ein Wert #t ist

```
cons : (%a (list-of %a) -> (list-of %a))
```

erzeuge ein Paar aus Element und Liste

```
empty : list
```

die leere Liste

```
empty? : (any -> boolean)
```

feststellen, ob ein Wert die leere Liste ist

```
computer : signature
```

Signatur für Computer

```
computer-hard-drive : (computer -> rational)
```

Festplattenkapazität aus Computer extrahieren

```
computer-processor : (computer -> string)
```

Prozessorname aus Computer extrahieren

```
computer-ram : (computer -> rational)
```

Arbeitsspeicher aus Computer extrahieren

```
computer? : (any -> boolean)
```

feststellen, ob Wert ein Computer ist

```
make-computer : (string rational rational -> computer)
```

Computer aus Prozessorname, Arbeitsspeicher und Festplattenkapazität konstruieren

```
chocolate-cookie : signature
```

Signatur für Schokokekse

```
chocolate-cookie-chocolate : (chocolate-cookie -> number)
```

Schoko-Anteil eines Schokokekses extrahieren

```
chocolate-cookie-cookie : (chocolate-cookie -> number)
```

Keks-Anteil eines Schokokekses extrahieren

```
chocolate-cookie? : (any -> boolean)
```

feststellen, ob ein Wert ein Schokokeks ist

```
make-chocolate-cookie : (number number -> chocolate-cookie)
```

Schokokeks aus Schoko- und Keks-Anteil konstruieren

```
string->strings-list : (string -> (list string))
```

Eine Zeichenkette in eine Liste von Zeichenketten mit einzelnen Zeichen umwandeln

```
string-append : (string string ... -> string)
```

Hängt Zeichenketten zu einer Zeichenkette zusammen

```
string-length : (string -> natural)
```

Liefert Länge einer Zeichenkette

```
string<=? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf kleiner-gleich testen

```
string<? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf kleiner-als testen

```
string=? : (string string string ... -> boolean)
```

Zeichenketten auf Gleichheit testen

```
string>=? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf größer-gleich testen

```
string>? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf größer-als testen

```
string? : (any -> boolean)
```

feststellen, ob ein Wert eine Zeichenkette ist

`strings-list->string : ((list string) -> string)`

Eine Liste von Zeichenketten in eine Zeichenkette umwandeln

`read : (-> any)`

Externe Repräsentation eines Werts in der REPL einlesen und den zugehörigen Wert liefern

`violation : (string -> unspecific)`

Programmm mit Fehlermeldung abbrechen

`write-newline : (-> unspecific)`

Zeilenumbruch ausgeben

`write-string : (string -> unspecific)`

Zeichenkette in REPL ausgeben

2 Die Macht der Abstraktion

This is documentation for the language level *Die Macht der Abstraktion* to go with the German textbook *Die Macht der Abstraktion*.

```
     program = def-or-expr ...

 def-or-expr = definition
             | expr
             | test-case

  definition = (define id expr)
             | (define-record-procedures id id id (id ...))
             | (define-record-procedures-parametric (id id ...) id id (id ...))
             | (: id sig)

        expr = (expr expr ...) ; Prozedurapplikation
             | #t
             | #f
             | number
             | string
             | (lambda (id ...) expr)
             | id ; Bezeichner
             | (cond (expr expr) (expr expr) ...)
             | (cond (expr expr) ... (else expr))
             | (if expr expr)
             | (and expr ...)
             | (or expr ...)
             | (let ((id expr) ...) expr)
             | (letrec ((id expr) ...) expr)
             | (let* ((id expr) ...) expr)
             | (begin expr expr ...)
             | (signature sig)
             | (for-all ((id sig) ...) expr)
             | (==> expr expr)

         sig = id
             | (predicate expr)
             | (one-of expr ...)
             | (mixed sig ...)
             | (sig ... -> sig) ; Prozedur-Signatur
             | (list sig)
             | %a %b %c ; Signatur-Variable
             | (combined sig ...)
             | signature
```

```
test-case = (check-expect expr expr)
          | (check-within expr expr expr)
          | (check-member-of expr expr ...)
          | (check-range expr expr expr)
          | (check-error expr expr)
          | (check-property expr)
```

Ein *id* ist eine Folge von Zeichen, die weder Leerzeichen noch eins der folgenden Zeichen enthält:

```
" , ' ` ( ) [ ] { } | ; #
```

Ein *number* ist eine Zahl wie z.B. `123`, `3/2` oder `5.5`.

Ein *string* ist eine Zeichenkette, und durch ein Paar von `"` umschlossen. So sind z.B. `"abcdef"`, `"This is a string"` und `"Dies ist eine Zeichenkette, die \" enthält."` Zeichenketten.

Zahlen

```
* : (number number number ... -> number)
+ : (number number number ... -> number)
- : (number number ... -> number)
/ : (number number number ... -> number)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (number number number ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real real ... -> boolean)
abs : (real -> real)
acos : (number -> number)
angle : (number -> real)
asin : (number -> number)
atan : (number -> number)
ceiling : (real -> integer)
complex? : (any -> boolean)
cos : (number -> number)
current-seconds : (-> natural)
denominator : (rational -> natural)
even? : (integer -> boolean)
exact->inexact : (number -> number)
exact? : (number -> boolean)
exp : (number -> number)
expt : (number number -> number)
floor : (real -> integer)
gcd : (integer integer ... -> natural)
```

```
imag-part : (number -> real)
inexact->exact : (number -> number)
inexact? : (number -> boolean)
integer? : (any -> boolean)
lcm : (integer integer ... -> natural)
log : (number -> number)
magnitude : (number -> real)
make-polar : (real real -> number)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (integer integer -> integer)
natural? : (any -> boolean)
negative? : (number -> boolean)
number->string : (number -> string)
number? : (any -> boolean)
numerator : (rational -> integer)
odd? : (integer -> boolean)
positive? : (number -> boolean)
quotient : (integer integer -> integer)
random : (natural -> natural)
rational? : (any -> boolean)
real-part : (number -> real)
real? : (any -> boolean)
remainder : (integer integer -> integer)
round : (real -> integer)
sin : (number -> number)
sqrt : (number -> number)
string->number : (string -> (mixed number false))
tan : (number -> number)
zero? : (number -> boolean)
```

boolesche Werte

```
boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
true? : (any -> boolean)
```

Listen

```
append : ((list-of %a) ... -> (list-of %a))
cons : (%a (list-of %a) -> (list-of %a))
cons? : (any -> boolean)
empty : list
empty? : (any -> boolean)
first : ((list-of %a) -> %a)
fold : (%b (%a %b -> %b) (list-of %a) -> %b)
length : ((list-of %a) -> natural)
list : (%a ... -> (list-of %a))
```

```
list-ref : ((list-of %a) natural -> %a)
make-pair : (%a (list-of %a) -> (list-of %a))
pair? : (any -> boolean)
rest : ((list-of %a) -> (list-of %a))
reverse : ((list-of %a) -> (list-of %a))
```

Computer

```
computer : signature
computer-hard-drive : (computer -> rational)
computer-processor : (computer -> string)
computer-ram : (computer -> rational)
computer? : (any -> boolean)
make-computer : (string rational rational -> computer)
```

Schokokekse

```
chocolate-cookie : signature
chocolate-cookie-chocolate : (chocolate-cookie -> number)
chocolate-cookie-cookie : (chocolate-cookie -> number)
chocolate-cookie? : (any -> boolean)
make-chocolate-cookie : (number number -> chocolate-cookie)
```

Zeichenketten

```
string->strings-list : (string -> (list string))
string-append : (string string ... -> string)
string-length : (string -> natural)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (string string string ... -> boolean)
string? : (any -> boolean)
strings-list->string : ((list string) -> string)
```

Symbole

Verschiedenes

```
for-each : ((%a -> %b) (list %a) -> unspecific)
map : ((%a -> %b) (list %a) -> (list %b))
read : (-> any)
violation : (string -> unspecific)
write-newline : (-> unspecific)
write-string : (string -> unspecific)
```

2.1 Signaturen

2.1.1 `list-of`

```
(list-of sig)
```

Diese Signatur ist dann für einen Wert gültig, wenn dieser eine Liste ist, für dessen Elemente *sig* gültig ist.

2.2 `let`, `letrec` und `let*`

```
(let ((id expr) ...) expr)
```

Bei einem `let`-Ausdruck werden zunächst die *expr*s aus den `(id expr)`-Paaren ausgewertet. Ihre Werte werden dann im Rumpf-*expr* für die Namen *id* eingesetzt. Dabei können sich die Ausdrücke nicht auf die Namen beziehen.

```
(define a 3)
(let ((a 16)
      (b a))
  (+ b a))
=> 19
```

Das Vorkommen von `a` in der Bindung von `b` bezieht sich also auf das `a` aus der Definition, nicht das `a` aus dem `let`-Ausdruck.

```
(letrec ((id expr) ...) expr)
```

Ein `letrec`-Ausdruck ist ähnlich zum entsprechenden `let`-Ausdruck, mit dem Unterschied, daß sich die *expr*s aus den Bindungen auf die gebundenen Namen beziehen dürfen.

```
(let* ((id expr) ...) expr)
```

Ein `let*`-Ausdruck ist ähnlich zum entsprechenden `let`-Ausdruck, mit dem Unterschied, daß sich die *expr*s aus den Bindungen auf die Namen beziehen dürfen, die jeweils vor dem *expr* gebunden wurden. Beispiel:

```
(define a 3)
(let* ((a 16)
       (b a))
  (+ b a))
=> 32
```

Das Vorkommen von `a` in der Bindung von `b` bezieht sich also auf das `a` aus dem `let*`-Ausdruck, nicht das `a` aus der globalen Definition.

2.3 Primitive Operationen

```
* : (number number number ... -> number)
```

Produkt berechnen

```
+ : (number number number ... -> number)
```

Summe berechnen

```
- : (number number ... -> number)
```

bei mehr als einem Argument Differenz zwischen der ersten und der Summe aller weiteren Argumente berechnen; bei einem Argument Zahl negieren

```
/ : (number number number ... -> number)
```

das erste Argument durch das Produkt aller weiteren Argumente berechnen

```
< : (real real real ... -> boolean)
```

Zahlen auf kleiner-als testen

```
<= : (real real real ... -> boolean)
```

Zahlen auf kleiner-gleich testen

```
= : (number number number ... -> boolean)
```

Zahlen auf Gleichheit testen

```
> : (real real real ... -> boolean)
```

Zahlen auf größer-als testen

```
>= : (real real real ... -> boolean)
```

Zahlen auf größer-gleich testen

```
abs : (real -> real)
```

Absolutwert berechnen

```
acos : (number -> number)
```

Arcuscosinus berechnen (in Radian)

```
angle : (number -> real)
```

Winkel einer komplexen Zahl berechnen

```
asin : (number -> number)
```

Arcussinus berechnen (in Radian)

```
atan : (number -> number)
```

Arcustangens berechnen (in Radian)

```
ceiling : (real -> integer)
```

nächste ganze Zahl oberhalb einer rellen Zahlen berechnen

```
complex? : (any -> boolean)
```

feststellen, ob ein Wert eine komplexe Zahl ist

```
cos : (number -> number)
```

Cosinus berechnen (Argument in Radian)

```
current-seconds : (-> natural)
```

aktuelle Zeit in Sekunden seit einem unspezifizierten Startzeitpunkt berechnen

```
denominator : (rational -> natural)
```

Nenner eines Bruchs berechnen

```
even? : (integer -> boolean)
```

feststellen, ob eine Zahl gerade ist

```
exact->inexact : (number -> number)
```

eine Zahl durch eine inexakte Zahl annähern

```
exact? : (number -> boolean)
```

feststellen, ob eine Zahl exakt ist

```
exp : (number -> number)
```

Exponentialfunktion berechnen (e hoch Argument)

```
expt : (number number -> number)
```

Potenz berechnen (erstes Argument hoch zweites Argument)

```
floor : (real -> integer)
```

nächste ganze Zahl unterhalb einer rellen Zahlen berechnen

```
gcd : (integer integer ... -> natural)
```

größten gemeinsamen Teiler berechnen

```
imag-part : (number -> real)
```

imaginären Anteil einer komplexen Zahl extrahieren

```
inexact->exact : (number -> number)
```

eine Zahl durch eine exakte Zahl annähern

```
inexact? : (number -> boolean)
```

feststellen, ob eine Zahl inexakt ist

```
integer? : (any -> boolean)
```

feststellen, ob ein Wert eine ganze Zahl ist

```
lcm : (integer integer ... -> natural)
```

kleinstes gemeinsames Vielfaches berechnen

```
log : (number -> number)
```

natürlichen Logarithmus (Basis e) berechnen

```
magnitude : (number -> real)
```

Abstand zum Ursprung einer komplexen Zahl berechnen

```
make-polar : (real real -> number)
```

komplexe Zahl aus Abstand zum Ursprung und Winkel berechnen

```
max : (real real ... -> real)
```

Maximum berechnen

```
min : (real real ... -> real)
```

Minimum berechnen

```
modulo : (integer integer -> integer)
```

Divisionsmodulo berechnen

```
natural? : (any -> boolean)
```

feststellen, ob ein Wert eine natürliche Zahl (inkl. 0) ist

```
negative? : (number -> boolean)
```

feststellen, ob eine Zahl negativ ist

```
number->string : (number -> string)
```

Zahl in Zeichenkette umwandeln

```
number? : (any -> boolean)
```

feststellen, ob ein Wert eine Zahl ist

```
numerator : (rational -> integer)
```

Zähler eines Bruchs berechnen

```
odd? : (integer -> boolean)
```

feststellen, ob eine Zahl ungerade ist

```
positive? : (number -> boolean)
```

feststellen, ob eine Zahl positiv ist

```
quotient : (integer integer -> integer)
```

ganzzahlig dividieren

```
random : (natural -> natural)
```

eine natürliche Zufallszahl berechnen, die kleiner als das Argument ist

```
rational? : (any -> boolean)
```

feststellen, ob eine Zahl rational ist

```
real-part : (number -> real)
```

reellen Anteil einer komplexen Zahl extrahieren

```
real? : (any -> boolean)
```

feststellen, ob ein Wert eine reelle Zahl ist

```
remainder : (integer integer -> integer)
```

Divisionsrest berechnen

```
round : (real -> integer)
```

relle Zahl auf eine ganze Zahl runden

```
sin : (number -> number)
```

Sinus berechnen (Argument in Radian)

```
sqrt : (number -> number)
```

Quadratwurzel berechnen

```
string->number : (string -> (mixed number false))
```

Zeichenkette in Zahl umwandeln, falls möglich

```
tan : (number -> number)
```

Tangens berechnen (Argument in Radian)

```
zero? : (number -> boolean)
```

feststellen, ob eine Zahl Null ist

```
boolean=? : (boolean boolean -> boolean)
```

Booleans auf Gleichheit testen

```
boolean? : (any -> boolean)
```

feststellen, ob ein Wert ein boolescher Wert ist

```
false? : (any -> boolean)
```

feststellen, ob ein Wert #f ist

```
not : (boolean -> boolean)
```

booleschen Wert negieren

```
true? : (any -> boolean)
```

feststellen, ob ein Wert #t ist

```
append : ((list-of %a) ... -> (list-of %a))
```

mehrere Listen aneinanderhängen

```
cons : (%a (list-of %a) -> (list-of %a))
```

erzeuge ein Paar aus Element und Liste

```
cons? : (any -> boolean)
```

feststellen, ob ein Wert ein Paar ist

```
empty : list
```

die leere Liste

```
empty? : (any -> boolean)
```

feststellen, ob ein Wert die leere Liste ist

```
first : ((list-of %a) -> %a)
```

erstes Element eines Paars extrahieren

```
fold : (%b (%a %b -> %b) (list-of %a) -> %b)
```

Liste einfalten.

```
length : ((list-of %a) -> natural)
```

Länge einer Liste berechnen

```
list : (%a ... -> (list-of %a))
```

Liste aus den Argumenten konstruieren

```
list-ref : ((list-of %a) natural -> %a)
```

das Listenelement an der gegebenen Position extrahieren

```
make-pair : (%a (list-of %a) -> (list-of %a))
```

erzeuge ein Paar aus Element und Liste

```
pair? : (any -> boolean)
```

feststellen, ob ein Wert ein Paar ist

```
rest : ((list-of %a) -> (list-of %a))
```

Rest eines Paars extrahieren

```
reverse : ((list-of %a) -> (list-of %a))
```

Liste in umgekehrte Reihenfolge bringen

```
computer : signature
```

Signatur für Computer

```
computer-hard-drive : (computer -> rational)
```

Festplattenkapazität aus Computer extrahieren

```
computer-processor : (computer -> string)
```

Prozessorname aus Computer extrahieren

```
computer-ram : (computer -> rational)
```

Arbeitsspeicher aus Computer extrahieren

```
computer? : (any -> boolean)
```

feststellen, ob Wert ein Computer ist

```
make-computer : (string rational rational -> computer)
```

Computer aus Prozessorname, Arbeitsspeicher und Festplattenkapazität konstruieren

```
chocolate-cookie : signature
```

Signatur für Schokokekse

```
chocolate-cookie-chocolate : (chocolate-cookie -> number)
```

Schoko-Anteil eines Schokokekses extrahieren

```
chocolate-cookie-cookie : (chocolate-cookie -> number)
```

Keks-Anteil eines Schokokekses extrahieren

```
chocolate-cookie? : (any -> boolean)
```

feststellen, ob ein Wert ein Schokokeks ist

```
make-chocolate-cookie : (number number -> chocolate-cookie)
```

Schokokeks aus Schoko- und Keks-Anteil konstruieren

```
string->strings-list : (string -> (list string))
```

Eine Zeichenkette in eine Liste von Zeichenketten mit einzelnen Zeichen umwandeln

```
string-append : (string string ... -> string)
```

Hängt Zeichenketten zu einer Zeichenkette zusammen

```
string-length : (string -> natural)
```

Liefert Länge einer Zeichenkette

```
string<=? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf kleiner-gleich testen

```
string<? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf kleiner-als testen

```
string=? : (string string string ... -> boolean)
```

Zeichenketten auf Gleichheit testen

```
string>=? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf größer-gleich testen

```
string>? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf größer-als testen

```
string? : (any -> boolean)
```

feststellen, ob ein Wert eine Zeichenkette ist

```
strings-list->string : ((list string) -> string)
```

Eine Liste von Zeichenketten in eine Zeichenkette umwandeln

```
for-each : ((%a -> %b) (list %a) -> unspecific)
```

Prozedur von vorn nach hinten auf alle Elemente einer Liste anwenden

```
map : ((%a -> %b) (list %a) -> (list %b))
```

Prozedur auf alle Elemente einer Liste anwenden, Liste der Resultate berechnen

```
read : (-> any)
```

Externe Repräsentation eines Werts in der REPL einlesen und den zugehörigen Wert liefern

```
violation : (string -> unspecific)
```

Programmm mit Fehlermeldung abbrechen

```
write-newline : (-> unspecific)
```

Zeilenumbruch ausgeben

```
write-string : (string -> unspecific)
```

Zeichenkette in REPL ausgeben

3 Die Macht der Abstraktion mit Zuweisungen

This is documentation for the language level *Die Macht der Abstraktion mit Zuweisungen* to go with the German textbook *Die Macht der Abstraktion*.

```
     program = def-or-expr ...

 def-or-expr = definition
             | expr
             | test-case

  definition = (define id expr)
             | (define-record-procedures id id id (id ...))
             | (define-record-procedures-parametric (id id ...) id id (id ...))
             | (: id sig)
             | (define-record-procedures-2 id id id (field-spec ...))
             | (define-record-procedures-parametric-2 id id id id id (field-spec ...))

  field-spec = id
             | (id id)

        expr = (expr expr ...) ; Prozedurapplikation
             | #t
             | #f
             | number
             | string
             | (lambda (id ...) expr)
             | id ; Bezeichner
             | (cond (expr expr) (expr expr) ...)
             | (cond (expr expr) ... (else expr))
             | (if expr expr)
             | (and expr ...)
             | (or expr ...)
             | (let ((id expr) ...) expr)
             | (letrec ((id expr) ...) expr)
             | (let* ((id expr) ...) expr)
             | (begin expr expr ...)
             | (signature sig)
             | (for-all ((id sig) ...) expr)
             | (==> expr expr)
             | (set! id expr)

         sig = id
             | (predicate expr)
             | (one-of expr ...)
```

```
            | (mixed sig ...)
            | (sig ... -> sig) ; Prozedur-Signatur
            | (list sig)
            | %a %b %c ; Signatur-Variable
            | (combined sig ...)
            | signature

test-case = (check-expect expr expr)
            | (check-within expr expr expr)
            | (check-member-of expr expr ...)
            | (check-range expr expr expr)
            | (check-error expr expr)
            | (check-property expr)
```

Ein *id* ist eine Folge von Zeichen, die weder Leerzeichen noch eins der folgenden Zeichen enthält:

`" , ' ` ( ) [ ] { } | ; #`

Ein *number* ist eine Zahl wie z.B. `123`, `3/2` oder `5.5`.

Ein *string* ist eine Zeichenkette, und durch ein Paar von `"` umschlossen. So sind z.B. `"abcdef"`, `"This is a string"` und `"Dies ist eine Zeichenkette, die \" enthält."` Zeichenketten.

Zahlen

```
* : (number number number ... -> number)
+ : (number number number ... -> number)
- : (number number ... -> number)
/ : (number number number ... -> number)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (number number number ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real real ... -> boolean)
abs : (real -> real)
acos : (number -> number)
angle : (number -> real)
asin : (number -> number)
atan : (number -> number)
ceiling : (real -> integer)
complex? : (any -> boolean)
cos : (number -> number)
current-seconds : (-> natural)
denominator : (rational -> natural)
even? : (integer -> boolean)
```

```
exact->inexact : (number -> number)
exact? : (number -> boolean)
exp : (number -> number)
expt : (number number -> number)
floor : (real -> integer)
gcd : (integer integer ... -> natural)
imag-part : (number -> real)
inexact->exact : (number -> number)
inexact? : (number -> boolean)
integer? : (any -> boolean)
lcm : (integer integer ... -> natural)
log : (number -> number)
magnitude : (number -> real)
make-polar : (real real -> number)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (integer integer -> integer)
natural? : (any -> boolean)
negative? : (number -> boolean)
number->string : (number -> string)
number? : (any -> boolean)
numerator : (rational -> integer)
odd? : (integer -> boolean)
positive? : (number -> boolean)
quotient : (integer integer -> integer)
random : (natural -> natural)
rational? : (any -> boolean)
real-part : (number -> real)
real? : (any -> boolean)
remainder : (integer integer -> integer)
round : (real -> integer)
sin : (number -> number)
sqrt : (number -> number)
string->number : (string -> (mixed number false))
tan : (number -> number)
zero? : (number -> boolean)
```

boolesche Werte

```
boolean=? : (boolean boolean -> boolean)
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
true? : (any -> boolean)
```

Listen

```
append : ((list-of %a) ... -> (list-of %a))
cons : (%a (list-of %a) -> (list-of %a))
cons? : (any -> boolean)
```

```
empty : list
empty? : (any -> boolean)
first : ((list-of %a) -> %a)
fold : (%b (%a %b -> %b) (list-of %a) -> %b)
length : ((list-of %a) -> natural)
list : (%a ... -> (list-of %a))
list-ref : ((list-of %a) natural -> %a)
make-pair : (%a (list-of %a) -> (list-of %a))
pair? : (any -> boolean)
rest : ((list-of %a) -> (list-of %a))
reverse : ((list-of %a) -> (list-of %a))
```

Computer

```
computer : signature
computer-hard-drive : (computer -> rational)
computer-processor : (computer -> string)
computer-ram : (computer -> rational)
computer? : (any -> boolean)
make-computer : (string rational rational -> computer)
```

Schokokekse

```
chocolate-cookie : signature
chocolate-cookie-chocolate : (chocolate-cookie -> number)
chocolate-cookie-cookie : (chocolate-cookie -> number)
chocolate-cookie? : (any -> boolean)
make-chocolate-cookie : (number number -> chocolate-cookie)
```

Zeichenketten

```
string->strings-list : (string -> (list string))
string-append : (string string ... -> string)
string-length : (string -> natural)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (string string string ... -> boolean)
string? : (any -> boolean)
strings-list->string : ((list string) -> string)
```

Symbole

Verschiedenes

```
apply : (procedure (list %a) -> %b)
eq? : (%a %b -> boolean)
equal? : (%a %b -> boolean)
for-each : ((%a -> %b) (list %a) -> unspecific)
map : ((%a -> %b) (list %a) -> (list %b))
read : (-> any)
violation : (string -> unspecific)
write-newline : (-> unspecific)
write-string : (string -> unspecific)
```

3.1 define-record-procedures-2

```
(define-record-procedures-2 t c p (field-spec ...))
```

Die `define-record-procedures-2`-Form ist eine Definition für einen neuen Record-Typ. Dabei ist *t* der Name der Record-Signatur, *c* der Name des Konstruktors, *p* der Name des Prädikats. Jedes *field-spec* kann entweder der Name eines Selektors oder ein Paar `(id id)` aus dem Namen eines Selektors und dem Namen eines Mutators sein.

3.2 define-record-procedures-parametric-2

```
(define-record-procedures-parametric-2 t cc c p (field-spec1 ...))
```

Diese Form ist wie `define-record-procedures-2`, nur parametrisch wie `define-record-procedures-parametric`. Außerdem werden die Signaturen für die Feldinhalte, anders als bei `define-record-procedures-parametric`, sofort bei der Konstruktion überprüft und nicht erst beim Aufruf eines Selektors.

3.3 begin

```
(begin expr expr ...)
```

Bei der Auswertung eines `begin`-Ausdrucks werden nacheinander die Operanden ausgewertet. Der Wert des letzten Ausdrucks wird der Wert des `begin`-Ausdrucks.

3.4 set!

```
(set! id expr)
```

Ein `set!`-Ausdruck ist eine Zuweisung, und ändert den Inhalt der Zelle, die an *id* gebunden ist, auf den Wert von *expr*.

3.5 Signaturen

```
unspecific
```

Signatur für unspezifische Werte, die unwichtig sind - typischerweise für die Rückgabewerte von Operationen, die nur Seiteneffekte haben wie `set!` oder `write-string`.

3.6 Definitionen

```
(define id expr)
```

Diese Form ist wie in den unteren Sprachebenen, mit dem Unterschied, dass an *id* mit `set!` zugewiesen werden kann.

3.7 `lambda`

```
(lambda (id id ... . id) expr)
```

Bei `lambda` ist in dieser Sprachebene in einer Form zulässig, die es erlaubt, eine Prozedur mit einer variablen Anzahl von Paramern zu erzeugen: Alle Parameter vor dem Punkt funktionieren wie gewohnt und werden jeweils an die entsprechenden Argumente gebunden. Alle restlichen Argumente werden in eine Liste verpackt und an den Parameter nach dem Punkt gebunden.

3.8 Primitive Operationen

```
* : (number number number ... -> number)
```

Produkt berechnen

```
+ : (number number number ... -> number)
```

Summe berechnen

```
- : (number number ... -> number)
```

bei mehr als einem Argument Differenz zwischen der ersten und der Summe aller weiteren Argumente berechnen; bei einem Argument Zahl negieren

```
/ : (number number number ... -> number)
```

das erste Argument durch das Produkt aller weiteren Argumente berechnen

```
< : (real real real ... -> boolean)
```

Zahlen auf kleiner-als testen

```
<= : (real real real ... -> boolean)
```

Zahlen auf kleiner-gleich testen

```
= : (number number number ... -> boolean)
```

Zahlen auf Gleichheit testen

```
> : (real real real ... -> boolean)
```

Zahlen auf größer-als testen

```
>= : (real real real ... -> boolean)
```

Zahlen auf größer-gleich testen

```
abs : (real -> real)
```

Absolutwert berechnen

```
acos : (number -> number)
```

Arcuscosinus berechnen (in Radian)

```
angle : (number -> real)
```

Winkel einer komplexen Zahl berechnen

```
asin : (number -> number)
```

Arcussinus berechnen (in Radian)

```
atan : (number -> number)
```

Arcustangens berechnen (in Radian)

```
ceiling : (real -> integer)
```

nächste ganze Zahl oberhalb einer rellen Zahlen berechnen

```
complex? : (any -> boolean)
```

feststellen, ob ein Wert eine komplexe Zahl ist

```
cos : (number -> number)
```

Cosinus berechnen (Argument in Radian)

```
current-seconds : (-> natural)
```

aktuelle Zeit in Sekunden seit einem unspezifizierten Startzeitpunkt berechnen

```
denominator : (rational -> natural)
```

Nenner eines Bruchs berechnen

```
even? : (integer -> boolean)
```

feststellen, ob eine Zahl gerade ist

```
exact->inexact : (number -> number)
```

eine Zahl durch eine inexakte Zahl annähern

```
exact? : (number -> boolean)
```

feststellen, ob eine Zahl exakt ist

```
exp : (number -> number)
```

Exponentialfunktion berechnen (e hoch Argument)

```
expt : (number number -> number)
```

Potenz berechnen (erstes Argument hoch zweites Argument)

```
floor : (real -> integer)
```

nächste ganze Zahl unterhalb einer rellen Zahlen berechnen

```
gcd : (integer integer ... -> natural)
```

größten gemeinsamen Teiler berechnen

```
imag-part : (number -> real)
```

imaginären Anteil einer komplexen Zahl extrahieren

```
inexact->exact : (number -> number)
```

eine Zahl durch eine exakte Zahl annähern

```
inexact? : (number -> boolean)
```

feststellen, ob eine Zahl inexakt ist

```
integer? : (any -> boolean)
```

feststellen, ob ein Wert eine ganze Zahl ist

```
lcm : (integer integer ... -> natural)
```

kleinstes gemeinsames Vielfaches berechnen

```
log : (number -> number)
```

natürlichen Logarithmus (Basis e) berechnen

```
magnitude : (number -> real)
```

Abstand zum Ursprung einer komplexen Zahl berechnen

```
make-polar : (real real -> number)
```

komplexe Zahl aus Abstand zum Ursprung und Winkel berechnen

```
max : (real real ... -> real)
```

Maximum berechnen

```
min : (real real ... -> real)
```

Minimum berechnen

```
modulo : (integer integer -> integer)
```

Divisionsmodulo berechnen

```
natural? : (any -> boolean)
```

feststellen, ob ein Wert eine natürliche Zahl (inkl. 0) ist

```
negative? : (number -> boolean)
```

feststellen, ob eine Zahl negativ ist

```
number->string : (number -> string)
```

Zahl in Zeichenkette umwandeln

```
number? : (any -> boolean)
```

feststellen, ob ein Wert eine Zahl ist

```
numerator : (rational -> integer)
```

Zähler eines Bruchs berechnen

```
odd? : (integer -> boolean)
```

feststellen, ob eine Zahl ungerade ist

```
positive? : (number -> boolean)
```

feststellen, ob eine Zahl positiv ist

```
quotient : (integer integer -> integer)
```

ganzzahlig dividieren

```
random : (natural -> natural)
```

eine natürliche Zufallszahl berechnen, die kleiner als das Argument ist

```
rational? : (any -> boolean)
```

feststellen, ob eine Zahl rational ist

```
real-part : (number -> real)
```

reellen Anteil einer komplexen Zahl extrahieren

```
real? : (any -> boolean)
```

feststellen, ob ein Wert eine reelle Zahl ist

```
remainder : (integer integer -> integer)
```

Divisionsrest berechnen

```
round : (real -> integer)
```

relle Zahl auf eine ganze Zahl runden

```
sin : (number -> number)
```

Sinus berechnen (Argument in Radian)

```
sqrt : (number -> number)
```

Quadratwurzel berechnen

```
string->number : (string -> (mixed number false))
```

Zeichenkette in Zahl umwandeln, falls möglich

```
tan : (number -> number)
```

Tangens berechnen (Argument in Radian)

```
zero? : (number -> boolean)
```

feststellen, ob eine Zahl Null ist

```
boolean=? : (boolean boolean -> boolean)
```

Booleans auf Gleichheit testen

```
boolean? : (any -> boolean)
```

feststellen, ob ein Wert ein boolescher Wert ist

```
false? : (any -> boolean)
```

feststellen, ob ein Wert #f ist

```
not : (boolean -> boolean)
```

booleschen Wert negieren

```
true? : (any -> boolean)
```

feststellen, ob ein Wert #t ist

```
append : ((list-of %a) ... -> (list-of %a))
```

mehrere Listen aneinanderhängen

```
cons : (%a (list-of %a) -> (list-of %a))
```

erzeuge ein Paar aus Element und Liste

```
cons? : (any -> boolean)
```

feststellen, ob ein Wert ein Paar ist

```
empty : list
```

die leere Liste

```
empty? : (any -> boolean)
```

feststellen, ob ein Wert die leere Liste ist

```
first : ((list-of %a) -> %a)
```

erstes Element eines Paars extrahieren

```
fold : (%b (%a %b -> %b) (list-of %a) -> %b)
```

Liste einfalten.

```
length : ((list-of %a) -> natural)
```

Länge einer Liste berechnen

```
list : (%a ... -> (list-of %a))
```

Liste aus den Argumenten konstruieren

```
list-ref : ((list-of %a) natural -> %a)
```

das Listenelement an der gegebenen Position extrahieren

```
make-pair : (%a (list-of %a) -> (list-of %a))
```

erzeuge ein Paar aus Element und Liste

```
pair? : (any -> boolean)
```

feststellen, ob ein Wert ein Paar ist

```
rest : ((list-of %a) -> (list-of %a))
```

Rest eines Paars extrahieren

```
reverse : ((list-of %a) -> (list-of %a))
```

Liste in umgekehrte Reihenfolge bringen

```
computer : signature
```

Signatur für Computer

```
computer-hard-drive : (computer -> rational)
```

Festplattenkapazität aus Computer extrahieren

```
computer-processor : (computer -> string)
```

Prozessorname aus Computer extrahieren

```
computer-ram : (computer -> rational)
```

Arbeitsspeicher aus Computer extrahieren

```
computer? : (any -> boolean)
```

feststellen, ob Wert ein Computer ist

```
make-computer : (string rational rational -> computer)
```

Computer aus Prozessorname, Arbeitsspeicher und Festplattenkapazität konstruieren

```
chocolate-cookie : signature
```

Signatur für Schokokekse

```
chocolate-cookie-chocolate : (chocolate-cookie -> number)
```

Schoko-Anteil eines Schokokekses extrahieren

```
chocolate-cookie-cookie : (chocolate-cookie -> number)
```

Keks-Anteil eines Schokokekses extrahieren

```
chocolate-cookie? : (any -> boolean)
```

feststellen, ob ein Wert ein Schokokeks ist

```
make-chocolate-cookie : (number number -> chocolate-cookie)
```

Schokokeks aus Schoko- und Keks-Anteil konstruieren

```
string->strings-list : (string -> (list string))
```

Eine Zeichenkette in eine Liste von Zeichenketten mit einzelnen Zeichen umwandeln

```
string-append : (string string ... -> string)
```

Hängt Zeichenketten zu einer Zeichenkette zusammen

```
string-length : (string -> natural)
```

Liefert Länge einer Zeichenkette

```
string<=? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf kleiner-gleich testen

```
string<? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf kleiner-als testen

```
string=? : (string string string ... -> boolean)
```

Zeichenketten auf Gleichheit testen

```
string>=? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf größer-gleich testen

```
string>? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf größer-als testen

```
string? : (any -> boolean)
```

feststellen, ob ein Wert eine Zeichenkette ist

`strings-list->string : ((list string) -> string)`

Eine Liste von Zeichenketten in eine Zeichenkette umwandeln

`apply : (procedure (list %a) -> %b)`

Prozedur auf Liste ihrer Argumente anwenden

`eq? : (%a %b -> boolean)`

zwei Werte auf Selbheit testen

`equal? : (%a %b -> boolean)`

zwei Werte auf Gleichheit testen

`for-each : ((%a -> %b) (list %a) -> unspecific)`

Prozedur von vorn nach hinten auf alle Elemente einer Liste anwenden

`map : ((%a -> %b) (list %a) -> (list %b))`

Prozedur auf alle Elemente einer Liste anwenden, Liste der Resultate berechnen

`read : (-> any)`

Externe Repräsentation eines Werts in der REPL einlesen und den zugehörigen Wert liefern

`violation : (string -> unspecific)`

Programmm mit Fehlermeldung abbrechen

`write-newline : (-> unspecific)`

Zeilenumbruch ausgeben

`write-string : (string -> unspecific)`

Zeichenkette in REPL ausgeben

4 Die Macht der Abstraktion fortgeschritten

This is documentation for the language level *Die Macht der Abstraktion - fortgeschritten* that goes with the German textbook *Die Macht der Abstraktion*.

```
     program = def-or-expr ...

 def-or-expr = definition
             | expr
             | test-case

  definition = (define id expr)
             | (define-record-procedures id id id (id ...))
             | (define-record-procedures-parametric (id id ...) id id (id ...))
             | (: id sig)
             | (define-record-procedures-2 id id id (field-spec ...))
             | (define-record-procedures-parametric-2 id id id id (field-spec ...))

  field-spec = id
             | (id id)

      quoted = id
             | number
             | string
             | character
             | (quoted ...)
             | 'quoted

        expr = (expr expr ...) ; Prozedurapplikation
             | #t
             | #f
             | number
             | string
             | (lambda (id ...) expr)
             | id ; Bezeichner
             | (cond (expr expr) (expr expr) ...)
             | (cond (expr expr) ... (else expr))
             | (if expr expr)
             | (and expr ...)
             | (or expr ...)
             | (let ((id expr) ...) expr)
             | (letrec ((id expr) ...) expr)
             | (let* ((id expr) ...) expr)
             | (begin expr expr ...)
             | (signature sig)
```

```
        | (for-all ((id sig) ...) expr)
        | (==> expr expr)
        | (set! id expr)
        | 'quoted ; Quote-Literal

   sig = id
        | (predicate expr)
        | (one-of expr ...)
        | (mixed sig ...)
        | (sig ... -> sig) ; Prozedur-Signatur
        | (list sig)
        | %a %b %c ; Signatur-Variable
        | (combined sig ...)
        | signature

test-case = (check-expect expr expr)
        | (check-within expr expr expr)
        | (check-member-of expr expr ...)
        | (check-range expr expr expr)
        | (check-error expr expr)
        | (check-property expr)
```

Ein *id* ist eine Folge von Zeichen, die weder Leerzeichen noch eins der folgenden Zeichen enthält:

`" , ' ` ( ) [ ] { } | ; #`

Ein *number* ist eine Zahl wie z.B. `123`, `3/2` oder `5.5`.

Ein *string* ist eine Zeichenkette, und durch ein Paar von `"` umschlossen. So sind z.B. `"abcdef"`, `"This is a string"` und `"Dies ist eine Zeichenkette, die \" enthält."` Zeichenketten.

Zahlen

```
* : (number number number ... -> number)
+ : (number number number ... -> number)
- : (number number ... -> number)
/ : (number number number ... -> number)
< : (real real real ... -> boolean)
<= : (real real real ... -> boolean)
= : (number number number ... -> boolean)
> : (real real real ... -> boolean)
>= : (real real real ... -> boolean)
abs : (real -> real)
acos : (number -> number)
angle : (number -> real)
```

```
asin : (number -> number)
atan : (number -> number)
ceiling : (real -> integer)
complex? : (any -> boolean)
cos : (number -> number)
current-seconds : (-> natural)
denominator : (rational -> natural)
even? : (integer -> boolean)
exact->inexact : (number -> number)
exact? : (number -> boolean)
exp : (number -> number)
expt : (number number -> number)
floor : (real -> integer)
gcd : (integer integer ... -> natural)
imag-part : (number -> real)
inexact->exact : (number -> number)
inexact? : (number -> boolean)
integer? : (any -> boolean)
lcm : (integer integer ... -> natural)
log : (number -> number)
magnitude : (number -> real)
make-polar : (real real -> number)
max : (real real ... -> real)
min : (real real ... -> real)
modulo : (integer integer -> integer)
natural? : (any -> boolean)
negative? : (number -> boolean)
number->string : (number -> string)
number? : (any -> boolean)
numerator : (rational -> integer)
odd? : (integer -> boolean)
positive? : (number -> boolean)
quotient : (integer integer -> integer)
random : (natural -> natural)
rational? : (any -> boolean)
real-part : (number -> real)
real? : (any -> boolean)
remainder : (integer integer -> integer)
round : (real -> integer)
sin : (number -> number)
sqrt : (number -> number)
string->number : (string -> (mixed number false))
tan : (number -> number)
zero? : (number -> boolean)
```

boolesche Werte

```
boolean=? : (boolean boolean -> boolean)
```

```
boolean? : (any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
true? : (any -> boolean)
```

Listen

```
append : ((list-of %a) ... -> (list-of %a))
cons : (%a (list-of %a) -> (list-of %a))
cons? : (any -> boolean)
empty : list
empty? : (any -> boolean)
first : ((list-of %a) -> %a)
fold : (%b (%a %b -> %b) (list-of %a) -> %b)
length : ((list-of %a) -> natural)
list : (%a ... -> (list-of %a))
list-ref : ((list-of %a) natural -> %a)
make-pair : (%a (list-of %a) -> (list-of %a))
pair? : (any -> boolean)
rest : ((list-of %a) -> (list-of %a))
reverse : ((list-of %a) -> (list-of %a))
```

Computer

```
computer : signature
computer-hard-drive : (computer -> rational)
computer-processor : (computer -> string)
computer-ram : (computer -> rational)
computer? : (any -> boolean)
make-computer : (string rational rational -> computer)
```

Schokokekse

```
chocolate-cookie : signature
chocolate-cookie-chocolate : (chocolate-cookie -> number)
chocolate-cookie-cookie : (chocolate-cookie -> number)
chocolate-cookie? : (any -> boolean)
make-chocolate-cookie : (number number -> chocolate-cookie)
```

Zeichenketten

```
string->strings-list : (string -> (list string))
string-append : (string string ... -> string)
string-length : (string -> natural)
string<=? : (string string string ... -> boolean)
string<? : (string string string ... -> boolean)
string=? : (string string string ... -> boolean)
string>=? : (string string string ... -> boolean)
string>? : (string string string ... -> boolean)
string? : (any -> boolean)
strings-list->string : ((list string) -> string)
```

Symbole

```
string->symbol : (string -> symbol)
symbol->string : (symbol -> string)
```

```
symbol? : (any -> boolean)
```

Verschiedenes

```
apply : (procedure (list %a) -> %b)
eq? : (%a %b -> boolean)
equal? : (%a %b -> boolean)
for-each : ((%a -> %b) (list %a) -> unspecific)
map : ((%a -> %b) (list %a) -> (list %b))
read : (-> any)
violation : (string -> unspecific)
write-newline : (-> unspecific)
write-string : (string -> unspecific)
```

4.1 Quote-Literal

```
'quoted
(quote quoted)
```

Der Wert eines Quote-Literals hat die gleiche externe Repräsentation wie *quoted*.

4.2 Signaturen

```
symbol
```

Signatur für Symbole.

4.3 Primitive Operationen

```
* : (number number number ... -> number)
```

Produkt berechnen

```
+ : (number number number ... -> number)
```

Summe berechnen

```
- : (number number ... -> number)
```

bei mehr als einem Argument Differenz zwischen der ersten und der Summe aller weiteren Argumente berechnen; bei einem Argument Zahl negieren

```
/ : (number number number ... -> number)
```

das erste Argument durch das Produkt aller weiteren Argumente berechnen

```
< : (real real real ... -> boolean)
```

Zahlen auf kleiner-als testen

```
<= : (real real real ... -> boolean)
```

Zahlen auf kleiner-gleich testen

```
= : (number number number ... -> boolean)
```

Zahlen auf Gleichheit testen

```
> : (real real real ... -> boolean)
```

Zahlen auf größer-als testen

```
>= : (real real real ... -> boolean)
```

Zahlen auf größer-gleich testen

```
abs : (real -> real)
```

Absolutwert berechnen

```
acos : (number -> number)
```

Arcuscosinus berechnen (in Radian)

```
angle : (number -> real)
```

Winkel einer komplexen Zahl berechnen

`asin : (number -> number)`

Arcussinus berechnen (in Radian)

`atan : (number -> number)`

Arcustangens berechnen (in Radian)

`ceiling : (real -> integer)`

nächste ganze Zahl oberhalb einer rellen Zahlen berechnen

`complex? : (any -> boolean)`

feststellen, ob ein Wert eine komplexe Zahl ist

`cos : (number -> number)`

Cosinus berechnen (Argument in Radian)

`current-seconds : (-> natural)`

aktuelle Zeit in Sekunden seit einem unspezifizierten Startzeitpunkt berechnen

`denominator : (rational -> natural)`

Nenner eines Bruchs berechnen

`even? : (integer -> boolean)`

feststellen, ob eine Zahl gerade ist

`exact->inexact : (number -> number)`

eine Zahl durch eine inexakte Zahl annähern

```
exact? : (number -> boolean)
```

feststellen, ob eine Zahl exakt ist

```
exp : (number -> number)
```

Exponentialfunktion berechnen (e hoch Argument)

```
expt : (number number -> number)
```

Potenz berechnen (erstes Argument hoch zweites Argument)

```
floor : (real -> integer)
```

nächste ganze Zahl unterhalb einer rellen Zahlen berechnen

```
gcd : (integer integer ... -> natural)
```

größten gemeinsamen Teiler berechnen

```
imag-part : (number -> real)
```

imaginären Anteil einer komplexen Zahl extrahieren

```
inexact->exact : (number -> number)
```

eine Zahl durch eine exakte Zahl annähern

```
inexact? : (number -> boolean)
```

feststellen, ob eine Zahl inexakt ist

```
integer? : (any -> boolean)
```

feststellen, ob ein Wert eine ganze Zahl ist

```
lcm : (integer integer ... -> natural)
```

kleinstes gemeinsames Vielfaches berechnen

```
log : (number -> number)
```

natürlichen Logarithmus (Basis e) berechnen

```
magnitude : (number -> real)
```

Abstand zum Ursprung einer komplexen Zahl berechnen

```
make-polar : (real real -> number)
```

komplexe Zahl aus Abstand zum Ursprung und Winkel berechnen

```
max : (real real ... -> real)
```

Maximum berechnen

```
min : (real real ... -> real)
```

Minimum berechnen

```
modulo : (integer integer -> integer)
```

Divisionsmodulo berechnen

```
natural? : (any -> boolean)
```

feststellen, ob ein Wert eine natürliche Zahl (inkl. 0) ist

```
negative? : (number -> boolean)
```

feststellen, ob eine Zahl negativ ist

```
number->string : (number -> string)
```

Zahl in Zeichenkette umwandeln

```
number? : (any -> boolean)
```

feststellen, ob ein Wert eine Zahl ist

```
numerator : (rational -> integer)
```

Zähler eines Bruchs berechnen

```
odd? : (integer -> boolean)
```

feststellen, ob eine Zahl ungerade ist

```
positive? : (number -> boolean)
```

feststellen, ob eine Zahl positiv ist

```
quotient : (integer integer -> integer)
```

ganzzahlig dividieren

```
random : (natural -> natural)
```

eine natürliche Zufallszahl berechnen, die kleiner als das Argument ist

```
rational? : (any -> boolean)
```

feststellen, ob eine Zahl rational ist

```
real-part : (number -> real)
```

reellen Anteil einer komplexen Zahl extrahieren

```
real? : (any -> boolean)
```

feststellen, ob ein Wert eine reelle Zahl ist

```
remainder : (integer integer -> integer)
```

Divisionsrest berechnen

```
round : (real -> integer)
```

relle Zahl auf eine ganze Zahl runden

```
sin : (number -> number)
```

Sinus berechnen (Argument in Radian)

```
sqrt : (number -> number)
```

Quadratwurzel berechnen

```
string->number : (string -> (mixed number false))
```

Zeichenkette in Zahl umwandeln, falls möglich

```
tan : (number -> number)
```

Tangens berechnen (Argument in Radian)

```
zero? : (number -> boolean)
```

feststellen, ob eine Zahl Null ist

```
boolean=? : (boolean boolean -> boolean)
```

Booleans auf Gleichheit testen

```
boolean? : (any -> boolean)
```

feststellen, ob ein Wert ein boolescher Wert ist

```
false? : (any -> boolean)
```

feststellen, ob ein Wert #f ist

```
not : (boolean -> boolean)
```

booleschen Wert negieren

```
true? : (any -> boolean)
```

feststellen, ob ein Wert #t ist

```
append : ((list-of %a) ... -> (list-of %a))
```

mehrere Listen aneinanderhängen

```
cons : (%a (list-of %a) -> (list-of %a))
```

erzeuge ein Paar aus Element und Liste

```
cons? : (any -> boolean)
```

feststellen, ob ein Wert ein Paar ist

```
empty : list
```

die leere Liste

```
empty? : (any -> boolean)
```

feststellen, ob ein Wert die leere Liste ist

```
first : ((list-of %a) -> %a)
```

erstes Element eines Paars extrahieren

```
fold : (%b (%a %b -> %b) (list-of %a) -> %b)
```

Liste einfalten.

```
length : ((list-of %a) -> natural)
```

Länge einer Liste berechnen

```
list : (%a ... -> (list-of %a))
```

Liste aus den Argumenten konstruieren

```
list-ref : ((list-of %a) natural -> %a)
```

das Listenelement an der gegebenen Position extrahieren

```
make-pair : (%a (list-of %a) -> (list-of %a))
```

erzeuge ein Paar aus Element und Liste

```
pair? : (any -> boolean)
```

feststellen, ob ein Wert ein Paar ist

```
rest : ((list-of %a) -> (list-of %a))
```

Rest eines Paars extrahieren

```
reverse : ((list-of %a) -> (list-of %a))
```

Liste in umgekehrte Reihenfolge bringen

```
computer : signature
```

Signatur für Computer

```
computer-hard-drive : (computer -> rational)
```

Festplattenkapazität aus Computer extrahieren

```
computer-processor : (computer -> string)
```

Prozessorname aus Computer extrahieren

```
computer-ram : (computer -> rational)
```

Arbeitsspeicher aus Computer extrahieren

```
computer? : (any -> boolean)
```

feststellen, ob Wert ein Computer ist

```
make-computer : (string rational rational -> computer)
```

Computer aus Prozessorname, Arbeitsspeicher und Festplattenkapazität konstruieren

```
chocolate-cookie : signature
```

Signatur für Schokokekse

```
chocolate-cookie-chocolate : (chocolate-cookie -> number)
```

Schoko-Anteil eines Schokokekses extrahieren

```
chocolate-cookie-cookie : (chocolate-cookie -> number)
```

Keks-Anteil eines Schokokekses extrahieren

```
chocolate-cookie? : (any -> boolean)
```

feststellen, ob ein Wert ein Schokokeks ist

```
make-chocolate-cookie : (number number -> chocolate-cookie)
```

Schokokeks aus Schoko- und Keks-Anteil konstruieren

```
string->strings-list : (string -> (list string))
```

Eine Zeichenkette in eine Liste von Zeichenketten mit einzelnen Zeichen umwandeln

```
string-append : (string string ... -> string)
```

Hängt Zeichenketten zu einer Zeichenkette zusammen

```
string-length : (string -> natural)
```

Liefert Länge einer Zeichenkette

```
string<=? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf kleiner-gleich testen

```
string<? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf kleiner-als testen

```
string=? : (string string string ... -> boolean)
```

Zeichenketten auf Gleichheit testen

```
string>=? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf größer-gleich testen

```
string>? : (string string string ... -> boolean)
```

Zeichenketten lexikografisch auf größer-als testen

```
string? : (any -> boolean)
```

feststellen, ob ein Wert eine Zeichenkette ist

```
strings-list->string : ((list string) -> string)
```

Eine Liste von Zeichenketten in eine Zeichenkette umwandeln

```
string->symbol : (string -> symbol)
```

Zeichenkette in Symbol umwandeln

```
symbol->string : (symbol -> string)
```

Symbol in Zeichenkette umwandeln

```
symbol? : (any -> boolean)
```

feststellen, ob ein Wert ein Symbol ist

```
apply : (procedure (list %a) -> %b)
```

Prozedur auf Liste ihrer Argumente anwenden

```
eq? : (%a %b -> boolean)
```

zwei Werte auf Selbheit testen

```
equal? : (%a %b -> boolean)
```

zwei Werte auf Gleichheit testen

```
for-each : ((%a -> %b) (list %a) -> unspecific)
```

Prozedur von vorn nach hinten auf alle Elemente einer Liste anwenden

```
map : ((%a -> %b) (list %a) -> (list %b))
```

Prozedur auf alle Elemente einer Liste anwenden, Liste der Resultate berechnen

```
read : (-> any)
```

Externe Repräsentation eines Werts in der REPL einlesen und den zugehörigen Wert liefern

```
violation : (string -> unspecific)
```

Programmm mit Fehlermeldung abbrechen

```
write-newline : (-> unspecific)
```

Zeilenumbruch ausgeben

```
write-string : (string -> unspecific)
```

Zeichenkette in REPL ausgeben

5 Konstruktionsanleitungen 1 bis 10

This documents the design recipes of the German textbook *Die Macht der Abstraktion*.

Contents

5.1 Konstruktion von Prozeduren

Gehen Sie bei der Konstruktion einer Prozedur in folgender Reihenfolge vor:

- **Kurzbeschreibung** Schreiben Sie eine einzeilige Kurzbeschreibung.
- **Datenanalyse** Führen Sie eine Analyse der beteiligten Daten durch. Stellen Sie dabei fest, zu welcher Sorte die Daten gehören, ob Daten mit Fallunterscheidung vorliegen und ob zusammengesetzte oder gemischte Daten vorliegen.
- **Signatur** (im Buch "Vertrag") Wählen Sie einen Namen und schreiben Sie eine Signatur für die Prozedur.
- **Testfälle** Schreiben Sie einige Testfälle.
- **Gerüst** Leiten Sie direkt aus der Signatur das Gerüst der Prozedur her.
- **Schablone** Leiten Sie aus der Signatur und der Datenanalyse mit Hilfe der Konstruktionsanleitungen eine Schablone her.
- **Rumpf** Vervollständigen Sie den Rumpf der Prozedur.
- **Test** Vergewissern Sie sich, daß die Tests erfolgreich laufen.

5.2 Fallunterscheidung

Wenn ein Argument einer Prozedur zu einer Fallunterscheidung gehört, die möglichen Werte also in feste Kategorien sortiert werden können, steht im Rumpf eine Verzweigung. Die Anzahl der Zweige entspricht der Anzahl der Kategorien.

Die Schablone für eine Prozedur `proc`, deren Argument zu einer Sorte gehört, die n Kategorien hat, sieht folgendermaßen aus:

```
(: proc (sig -> ...))
(define proc
  (lambda (a)
    (cond
      (test_1 ...)
      ...
      (test_n ...))))
```

Dabei ist `sig` die Signatur, den die Elemente der Sorte erfüllen müssen. Die $\texttt{test}_i$ müssen Tests sein, welche die einzelnen Kategorien erkennen. Sie sollten alle Kategorien abdecken. Der letzte Zweig kann auch ein `else`-Zweig sein, falls klar ist, daß `a` zum letzten Fall gehört, wenn alle vorherigen $\texttt{test}_i$ `#f` ergeben haben. Anschließend werden die Zweige vervollständigt.

Bei Fallunterscheidungen mit zwei Kategorien kann auch `if` statt `cond` verwendet werden.

5.3 zusammengesetzte Daten

Wenn bei der Datenanalyse zusammengesetzte Daten vorkommen, stellen Sie zunächst fest, welche Komponenten zu welchen Sorten gehören. Schreiben Sie dann eine Datendefinition, die mit folgenden Worten anfängt:

```
; Ein x besteht aus / hat:
; - Feld_1 (sig_1)
; ...
; - Feld_n (sig_n)
```

Dabei ist `x` ein umgangssprachlicher Name für die Sorte (“Schokokeks”), die $\texttt{Feld}_i$ sind umgangssprachliche Namen und kurze Beschreibungen der Komponenten und die $\texttt{sig}_i$ die dazugehörigen Signaturen.

Übersetzen Sie die Datendefinition in eine Record-Definition, indem Sie auch Namen für die Record-Signatur `sig`, Konstruktor `constr`, Prädikat `pred?` und die Selektoren $\texttt{select}_i$ wählen:

```
(define-record-procedures sig
  constr pred?
  (select_1 ... select_n))
```

Schreiben Sie außerdem eine Signatur für den Konstruktor der Form:

```
(: constr (sig_1 ... sig_n -> sig))
```

Ggf. schreiben Sie außerdem Signaturen für das Prädikat und die Selektoren:

```
(: pred? (any -> boolean))
(: select_1 (sig -> sig_1))
...
(: select_n (sig -> sig_n))
```

5.4 zusammengesetzte Daten als Argumente

Wenn ein Argument einer Prozedur zusammengesetzt ist, stellen Sie zunächst fest, von welchen Komponenten des Records das Ergebnis der Prozeduren abhängt.

Schreiben Sie dann für jede Komponente `(select a)` in die Schablone, wobei `select` der Selektor der Komponente und `a` der Name des Parameters der Prozedur ist.

Vervollständigen Sie die Schablone, indem Sie einen Ausdruck konstruieren, in dem die Selektor-Anwendungen vorkommen.

5.5 zusammengesetzte Daten als Ausgabe

Eine Prozedur, die einen neuen zusammengesetzten Wert zurückgibt, enthält einen Aufruf des Konstruktors des zugehörigen Record-Typs.

5.6 gemischte Daten

Wenn bei der Datenanalyse gemischte Daten auftauchen, schreiben Sie eine Datendefinition der Form:

```
; Ein x ist eins der Folgenden:
; - Sorte_1 (sig_1)
; ...
; - Sorte_n (sig_n)
; Name: sig
```

Dabei sind die $\texttt{Sorte}_i$ umgangssprachliche Namen für die möglichen Sorten, die ein Wert aus diesen gemischten Daten annehmen kann. Die $\texttt{sig}_i$ sind die zu den Sorten gehörenden Signaturen. Der Name `sig` ist für die Verwendung als Signatur.

Aus der Datendefinition entsteht eine Signaturdefinition folgender Form:

```
(define sig
  (signature
    (mixed sig_1
           ...
           sig_n)))
```

Wenn die Prädikate für die einzelnen Sorten $\texttt{pred?}_1$ ... $\texttt{pred?}_n$ heißen, hat die Schablone für eine Prozedur, die gemischte Daten konsumiert, die folgende Form:

```
(: proc (sig -> ...))

(define proc
  (lambda (a)
    (cond
      ((pred?_1 a) ...)
      ...
      ((pred?_n a) ...))))
```

Die rechten Seiten der Zweige werden dann nach den Konstruktionsanleitungen der einzelnen Sorten ausgefüllt.

5.7 Listen

Eine Prozedur, die eine Liste konsumiert, hat die folgende Schablone:

```
(: proc ((list-of elem) -> ...))

(define proc
  (lambda (lis)
    (cond
      ((empty? lis) ...)
      ((pair? lis)
       ... (first lis)
       ... (proc (rest lis)) ...))))
```

Dabei ist `elem` die Signatur für die Elemente der Liste. Dies kann eine Signaturvariable (`%a`, `%b`, ...) sein, falls die Prozedur unabhängig von der Signatur der Listenelemente ist.

Füllen Sie in der Schablone zuerst den `empty?`-Zweig aus. Vervollständigen Sie dann den anderen Zweig unter der Annahme, daß der rekursive Aufruf `(proc (rest lis))` das gewünschte Ergebnis für den Rest der Liste liefert.

Beispiel:

```
(: list-sum ((list-of number) -> number))

(define list-sum
  (lambda (lis)
    (cond
      ((empty? lis) 0)
      ((pair? lis)
       (+ (first lis)
          (list-sum (rest lis)))))))
```

5.8 natürliche Zahlen

Eine Prozedur, die natürliche Zahlen konsumiert, hat die folgende Schablone:

```
(: proc (natural -> ...))

(define proc
  (lambda (n)
    (if (= n 0)
        ...
        ... (proc (- n 1)) ...)))
```

Füllen Sie in der Schablone zuerst den 0-Zweig aus. Vervollständigen Sie dann den anderen Zweig unter der Annahme, daß der rekursive Aufruf `(proc (- n 1))` das gewünschte Ergebnis für `n`-1 liefert.

Beispiel:

```
(: factorial (natural -> natural))

(define factorial
  (lambda (n)
    (if (= n 0)
        1
        (* n (factorial (- n 1))))))
```

5.9 Prozeduren mit Akkumulatoren

Eine Prozedur mit Akkumulator, die Listen konsumiert, hat die folgende Schablone:

```
(: proc ((list-of elem) -> ...))

(define proc
  (lambda (lis)
    (proc-helper lis z)))

(: proc ((list-of elem) sig -> ...))

(define proc-helper
  (lambda (lis acc)
    (cond
      ((empty? lis) acc)
      ((pair? lis)
       (proc-helper (rest lis)
                    (... (first lis) ... acc ...))))))
```

Hier ist `proc` der Name der zu definierenden Prozedur und `proc-helper` der Name der Hilfsprozedur mit Akkumulator. Der Anfangswert für den Akkumulator ist der Wert von `z`. Die Signatur `sig` ist die Signatur für den Akkumulator. Der Ausdruck `(... (first lis) ... acc ...)` macht aus dem alten Zwischenergebnis `acc` das neue Zwischenergebnis.

Beispiel:

```
(: invert ((list-of %a) -> (list-of %a)))

(define invert
  (lambda (lis)
```

```
    (invert-helper lis empty)))

(: invert ((list-of %a) (list-of %a) -> (list-of %a)))

(define invert-helper
  (lambda (lis acc)
    (cond
      ((empty? lis) acc)
      ((pair? lis)
       (invert-helper (rest lis)
                      (make-pair (first lis) acc))))))
```

Eine Prozedur mit Akkumulator, die natürliche Zahlen konsumiert, hat die folgende Schablone:

```
(: proc (natural -> ...))

(define proc
  (lambda (n)
    (proc-helper n z)))

(define proc-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        (proc-helper (- n 1) (... acc ...)))))
```

Dabei ist `z` das gewünschte Ergebnis für `n` = 0. Der Ausdruck `(... acc ...)` muß den neuen Wert für den Akkumulator berechnen.

Beispiel:

```
(: ! (natural -> natural))

(define !
  (lambda (n)
    (!-helper n 1)))

(define !-helper
  (lambda (n acc)
    (if (= n 0)
        acc
        (!-helper (- n 1) (* n acc)))))
```

5.10 gekapselter Zustand

Falls ein Wert Zustand enthalten soll, schreiben Sie eine Datendefinition wie bei zusammengesetzten Daten.

Schreiben Sie dann eine Record-Definition mit `define-record-procedures-2` und legen Sie dabei fest, welche Bestandteile veränderbar sein sollen. Geben Sie Mutatoren für die betroffenen Felder an. Wenn der Selektor für das Feld `select` heißt, sollte der Mutator i.d.R. `set-select!` heißen. Die Form sieht folgendermaßen aus, wobei an der Stelle `k` ein veränderbares Feld steht:

```
(define-record-procedures-2 sig
  constr pred?
  (select1 ... (sk mutatek) ... sn))
```

In der Schablone für Prozeduren, die den Zustand eines Record-Arguments `r` ändern, benutzen Sie den dazugehörigen Mutator $\texttt{mutate}_k$ Wenn `a` der Ausdruck für den neuen Wert der Komponente ist, sieht der Aufruf folgendermaßen aus: $\texttt{(mutate}_k\ \texttt{r a)}$.

Um mehrere Komponenten in einer Prozedur zu verändern, oder um einen sinnvollen Rückgabewert nach einer Mutation zu liefern, benutzen Sie `begin`.

6 Bilder konstruieren: "image.ss"

Note: This is documentation for the "image.rkt" teachpack that goes with the German textbook *Die Macht der Abstraktion*.

Dieses Teachpack definiert Prozeduren für die Konstruktion von Bildern. Einfache Bilder werden als geometrische Formen oder Bitmaps konstruiert. Zusätzliche Prozeduren erlauben die Komposition von Bildern.

6.1 Bilder

`image : signature`

Ein *Bild* (Name: `image`) ist die Repräsentation eines Bildes.

`empty-image : image`

Ein leeres Bild mit Breite und Höhe 0.

`image? : (any -> boolean?)`

Der Aufruf `(image? x)` stellt fest, ob `x` ein Bild ist.

6.2 Modi und Farben

`mode : signature`

```
(one-of "solid" "outline")
```

Ein Modus (Name: `mode`) legt fest, ob die Darstellung einer Form diese füllt oder nur einen Umriss zeichnet.

`octet : signature`

```
(combined natural (predicate (lambda (n) (<= n 255))))
```

Ein Oktet (Name: `octet`) ist eine natürliche Zahl zwischen 0 und 255.

`rgb-color : signature`

Eine *RGB-Farbe* ist eine Farbe (Name: `color`, die vom Record-Konstruktor `make-color` zurückgegeben wird:

```
make-color : (octet octet octet -> rgb-color)
```

Eine RGB-Farbe beschreibt eine Farbe mit den roten, blauen und grünen Anteilen, also z.B. `(make-color 100 200 30)`.

```
color-red : (color -> octet)
```

liefert den Rot-Anteil einer RGB-Farbe.

```
color-green : (color -> octet)
```

liefert den Grün-Anteil einer RGB-Farbe.

```
color-blue : (color -> octet)
```

liefert den Blau-Anteil einer RGB-Farbe.

```
color? : (any -> boolean)
```

stellt fest, ob ein Objekt eine RGB-Farbe ist.

```
image-color : signature

(mixed string rgb-color)
```

Eine *Farbe* (Name: `image-color`) ist eine Zeichenkette aus einer Farbbezeichnung (z.B. `"blue"`) oder eine RGB-Farbe.

```
image-color? : (any -> boolean?)
```

stellt fest, ob ein Objekt eine Farbe ist.

```
alpha-rgb-color : signature
```

Eine *Alpha/RGB-Farbe* ist eine Farbe (Name: `color`, die vom Record-Konstruktor `make-alpha-color` zurückgegeben wird:

```
make-alpha-color : (octet octet octet octet -> alpha-color)
```

Eine Alpha/RGB-Farbe beschreibt eine Farbe mit den Alpha-, roten, blaue und grünen Anteilen, also z.B. `(make-color 50 100 200 30)`. Der Alpha-Anteil beschreibt, wie durchsichtig die Farbe ist.

```
alpha-color-red : (color -> octet)
```

liefert den Rot-Anteil einer RGB-Farbe.

```
alpha-color-green : (color -> octet)
```

liefert den Grün-Anteil einer RGB-Farbe.

```
alpha-color-blue : (color -> octet)
```

liefert den Blau-Anteil einer RGB-Farbe.

```
alpha-color-alpha : (color -> octet)
```

liefert den Alpha-Anteil einer RGB-Farbe.

```
alpha-color? : (any -> boolean)
```

stellt fest, ob ein Objekt eine Alpha/RGB-Farbe ist.

6.3 Einfache geometrische Figuren

Die folgenden Prozeduren erzeugen Bilder mit einfachen geometrischen Formen:

```
rectangle : (natural natural mode image-color -> image)
```

Der Aufruf `(rectangle w h m c)` erzeugt ein Rechteck mit Breite `w` und Höhe `h`, gefüllt mit Modus `m` und in Farbe `c`.

```
circle : (natural mode image-color -> image)
```

Der Aufruf `(circle r m c)` erzeugt einen Kreis oder eine Scheibe mit Radius `r`, gefüllt mit Modus `m` und in Farbe `c`.

```
ellipse : (natural natural mode image-color -> image)
```

Der Aufruf `(ellipse w h m c)` erzeugt eine Ellipse mit Breite `w` und Höhe `h`, gefüllt mit Modus `m` und in Farbe `c`.

```
triangle : (integer mode image-color -> image)
```

Der Aufruf `(triangle s m c)` erzeugt ein nach oben zeigendes gleichseitiges Dreieck, wobei `s` die Seitenlänge angibt, gefüllt mit Modus `m` und in Farbe `c`.

```
line
 : (natural natural number number number number image-color -> image)
```

Der Aufruf `(line w h sx sy ex ey c)` erzeugt ein Bild mit einer farbigen Strecke, wobei `w` die Breite und `h` die Höhe des Bilds, sowie `sx` die X- und `sx` die Y-Koordinate des Anfangspunkts und `ex` die X- und `ey` die Y-Koordinate des Endpunkts angeben, gefüllt mit Modus `m` und in Farbe `c`.

```
text : (string natural image-color -> image)
```

Der Aufruf `(text s f c)` erzeugt ein Bild mit Text `s`, wobei die Buchstaben die Größe `f` haben, in Farbe `c`

Außerdem können beliebige Bitmap-Bilder in ein Scheme-Programm eingeklebt werden.

6.4 Eigenschaften von Bildern

Zwei Eigenschaften von Bildern sind für ihre Manipulation nützlich, nämlich Breite und Höhe:

```
image-width : (image -> natural)
```

liefert die Breite von `i` in Pixeln.

```
image-height : (image -> natural)
```

liefert die Höhe von `i` in Pixeln.

```
image-inside? : (image image -> boolean)
```

Der Aufruf `(image-inside? i1 i2)` stellt fest, ob das Bild `i2` im Bild `i1` enthalten ist.

```
find-image : (image image -> posn)
```

Der Aufruf `(find-image i1 i2)` findet die Position von `i2` im Bild `i1` (in dem es vorkommen muss).

6.5 Bilder zusammensetzen

The nächste Gruppe von Prozeduren baut aus Bildern neue Bilder:

```
h-place : signature
```

```
(mixed integer (one-of "left" "right" "center"))
```

Eine *horizontale Positionsangabe* (Name: `h-place`) gibt an, wie zwei Bilder horizontal zueinander positioniert werden

Im ersten Fall, wenn es sich um eine Zahl `x` handelt, wird das zweite Bild `x` Pixel vom linken Rand auf das erste gelegt. Die drei Fälle mit Zeichenketten sagen, daß die Bilder am linken Rand bzw. am rechten Rand bündig plaziert werden, bzw. das zweite Bild horizontal in die Mitte des ersten gesetzt wird.

```
v-place : signature
```

```
(mixed integer (one-of "top" "bottom" "center"))
```

Eine *vertikale Positionsangabe* (Name: `v-place`) gibt an, wie zwei Bilder vertikal zueinander positioniert werden

Im ersten Fall, wenn es sich um eine Zahl `y` handelt, wird das zweite Bild `y` Pixel vom oberen Rand auf das erste gelegt. Die drei Fälle mit Zeichenketten sagen, daß die Bilder am oberen Rand bzw. am unteren Rand bündig plaziert werden, bzw. das zweite Bild vertikal in die Mitte des ersten gesetzt wird.

```
h-mode : signature
```

`(one-of "left" "right" "center")` Eine *horizontale Justierungsangabe* (Name: `h-mode`) gibt an, ob zwei Bilder, die übereinander angeordnet werden, entlang der linken Kante, der rechten Kante oder der Mitte angeordnet werden.

```
v-mode : signature
```

```
(one-of "top" "bottom" "center")
```

Eine *vertikale Justierungsangabe* (Name: `V-mode`) gibt an, ob zwei Bilder, die nebenander angeordnet werden, entlang der oberen Kante, der untern Kante oder der Mitte angeordnet werden.

```
overlay : (image image h-place v-place -> image)
```

Der Aufruf `(overlay img other h v)` legt zweite Bild `other` auf das erste `img`. Die beiden anderen Argumente geben an, wie die beiden Bilder zueinander positioniert werden.

```
beside : (image image v-mode -> image)
```

Der Aufruf `(beside img other v)` ordnet die beiden Bilder entsprechend des `v`-Arguments nebeneinander an.

```
above : (image image h-mode -> image)
```

Der Aufruf `(img other h -> image)` ordnet die beiden Bilder entsprechend des `h`-Arguments übereinander an.

```
clip : (image natural natural natural natural -> image)
```

Der Aufruf `(clip img x y w h)` liefert das Teilrechteck des Bildes `img` bei (`x`, `y`), Breite `w` und Höhe `h`.

```
pad : (image natural natural natural natural -> image)
```

Der Aufruf `(pad img l r t b)` fügt an den Seiten von `img` noch transparenten Leerraum an: `l` Pixel links, `r` Pixel rechts, `t` Pixel oben und `b` Pixel unten.

6.6 Bilder aus Pixeln konstruieren

```
color-list->image : ((list-of color) natural natural -> image)
```

Der Aufruf `(color-list->image lis w h)` stellt ein Bild mit Breite `w` und Höhe `h` her, in dem die Pixel die Farben aus der Liste `lis` (welche die Länge `(* w h)` haben muß) haben.

```
image->color-list : (image -> (list-of rgb-color))
```

Diese Prozedur liefert eine Liste der RGB-Farben der Pixel eines Bildes.

```
alpha-color-list->image
 : ((list-of alpha-rgb-color) natural natural -> image)
```

Der Aufruf `(color-list->image lis w h)` stellt ein Bild mit Breite `w` und Höhe `h` her, in dem die Pixel die Farben aus der Liste `lis` (welche die Länge `(* w h)` haben muß) haben.

```
image->alpha-color-list : (image -> (list-of rgb-color))
```

Diese Prozedur liefert eine Liste der Alpha/RGB-Farben der Pixel eines Bildes.

7 Animationen: "world.ss"

Note: This is documentation for the "world.rkt" teachpack that goes with the German textbook *Die Macht der Abstraktion*.

Dieses Teachpack ermöglicht, kleine Animationen und Spiele zu programmieren. Es enthält alle Prozeduren aus dem image-Teachpack.

```
world : signature
```

Eine *Welt* (Name: world) ist die Repräsentation des Zustands, der durch die Animation abgebildet wird.

```
mouse-event-kind : signature
```

```
(one-of "enter" "leave" "motion" "left-down" "left-up" "middle-down"
"middle-up" "right-down" "right-up")
```

Eine *Mausereignis-Art* (Name: mouse-event-kind) bezeichnet die Art eines Maus-Ereignisses:

"enter" bedeutet, daß der Mauszeiger gerade in das Fenster hinein bewegt wurde. "leave" bedeutet, daß der Mauszeiger gerade aus dem Fenster heraus bewegt wurde. "motion" bedeutet, daß der Mauszeiger innerhalb des Fensters bewegt wurde. Die anderen Zeichenketten bedeuten, daß der entsprechende Mausknopf gedrückt oder losgelassen wurde.

```
big-bang : (natural natural number world -> (one-of #t))
```

Der Aufruf (big-bang w h n w) erzeugt eine Leinwand mit Breite w und Höhe h, startet die Uhr, die alle n Sekunden tickt, und macht w zur ersten Welt.

```
on-tick-event : ((world -> world) -> (one-of #t))
```

Der Aufruf (on-tick-event tock) meldet tock als Prozedur an, die bei jedem Uhren-Tick aufgerufen wird, um aus der alten Welt eine neue zu machen.

```
on-key-event : ((world string -> world) -> (one-of #t))
```

Der Aufruf (on-key-event change) meldet change als Prozedur an, die bei jedem Tastendruck aufgerufen wird, um aus der alten Welt eine neue zu machen. Dabei wird als Argument eine Zeichenkette übergeben, welche die Taste darstellt, also "a" für die A-Taste etc., sowie "up", "down", "left", und "right" für die entsprechenden Pfeiltasten und "wheel-up" für die Bewegung des Mausrads nach oben und "wheel-down" für die Bewegung des Mausrads nach unten.

```
on-mouse-event
 : ((world natural natural mouse-event-kind -> world) -> (one-of #t))
```

Der Aufruf `(on-mouse-event change)` meldet `change` als Prozedur an, die bei jedem Mausereignis aufgerufen wird, um aus der alten Welt eine neue zu machen. Die `change`-Prozedur wird als `(change w x y k)` aufgerufen. Dabei ist `w` die alte Welt, `x` und `y` die Koordinaten des Mauszeigers, und `k` die Art des Mausereignisses.

```
on-redraw : ((world -> image) -> (one-of #t))
```

Der Aufruf `(world->image world->image)` meldet die Prozedur `world->image` an, die aus einer Welt ein Bild macht, das auf der Leinwand dargestellt wird.

```
end-of-time : (string -> world)
```

Diese Prozedur hält die Welt an und druckt ihr Argument in der REPL aus.

8 Turtle-Grafik: `"turtle.ss"`

Note: This is documentation for the `"turtle.rkt"` teachpack that goes with the German textbook *Die Macht der Abstraktion*.

Turtle-Grafik ist eine Methode zum Erstellen von Computergrafiken. Das Zeichnen wird dabei durch das Bewegen einer virtuellen Schildkröte über den Zeichenbereich modelliert. Eine Schildkröte kann durch drei Befehle bewegt werden:

- `(move n)` Bewegt die Schildkröte um `n` Pixel ohne zu zeichnen.
- `(draw n)` Bewegt die Schildkröte um `n` Pixel und zeichnet dabei.
- `(turn n)` Dreht die Schildkröte um n Grad im Uhrzeigersinn.

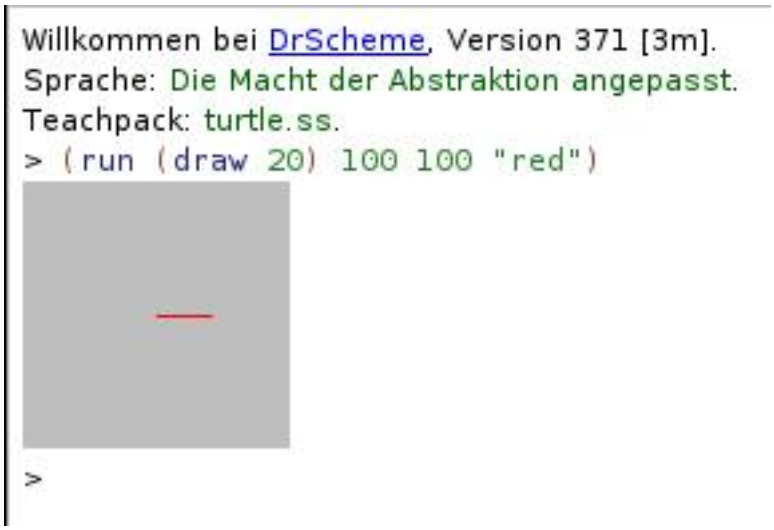
Wir stellen jetzt ein Teachpack für DrRacket vor, mit dessen Hilfe solche Turtle-Grafiken erstellt werden können.

8.1 Tutorial

Unser Ziel ist es, in diesem Tutorial ein Quadrat mithilfe der Prozeduren des Teachpacks zu zeichnen. Aus diesem Grund müssen wir zunächst mit der Prozedur `draw` eine Linie nach rechts malen. Die initiale Ausgansposition der Turtle ist in der Bildmitte mit Blick nach rechts. Mit `(draw 20)` bewegen wir die Turtle dann 20 Pixel nach rechts und zeichnen dabei. Um das resultierende Bild zu sehen ist, müssen wir die Turtle mittels der Prozedur run laufen lassen. Die restlichen Parameter für run sind die Höhe und die Breite des Bildes sowie die Farbe, in der gezeichnet werden soll. Geben Sie also folgenden Befehl in die REPL ein, um Ihre erste Turtle-Grafik zu erstellen:

```
(run (draw 20) 100 100 "red")
```

Sie erhalten dann eine Ausgabe wie die folgende:



Nun vervollständigen wir die Linie zu einem rechten Winkel: wir drehen die Turtle um 90° nach rechts und zeichnen dann eine Line der Länge 20 Pixel nach unten. Zum Drehen einer Turtle verwenden wir die Prozedur `turn`.

Da wir ein Quadrat aus zwei rechten Winkeln zusammensetzen können, abstrahieren wir über das Zeichnen des rechten Winkels. Dazu schreiben wir eine Prozedur `right-angle` die als Parameter eine Turtle erhält:

```
(: right-angle (turtle -> turtle))
(define right-angle
  (lambda (t1)
    (let* ((t2 ((draw 20) t1))
           (t3 ((turn -90) t2))
           (t4 ((draw 20) t3)))
      t4)))
```

Das Ergebnis sieht dann so aus:

```
Willkommen bei DrScheme, Version 371 [3m].
Sprache: Die Macht der Abstraktion angepasst.
Teachpack: turtle.ss.
> (run right-angle 100 100 "red")
```

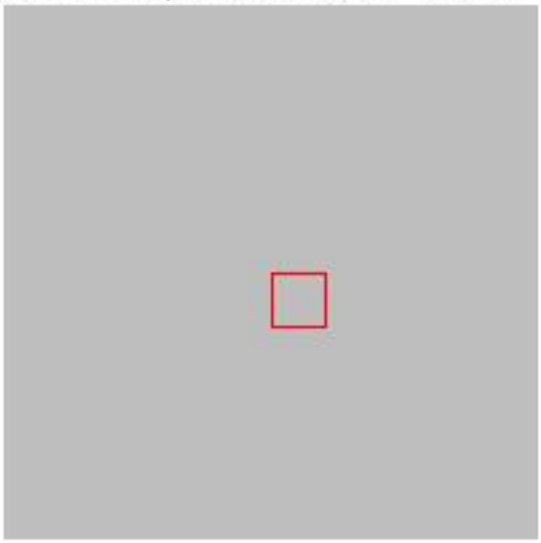
```
>
```

Um das Quadrat komplett zu zeichnen, sollen nun zwei rechte Winkel verwendet werden. Wir zeichnen also einen rechten Winkel, drehen uns um 90° nach rechts, und zeichnen einen zweiten rechten Winkel.

```
(: square (turtle -> turtle))
(define square
  (lambda (t1)
    (let* ((t2 (right-angle t1))
           (t3 ((turn -90) t2))
           (t4 (right-angle t3)))
      t4)))
```

So sieht das Ergebnis aus:

```
Willkommen bei DrScheme, Version 371 [3m].
Sprache: Die Macht der Abstraktion angepasst.
Teachpack: turtle.ss.
> (run square 200 200 "red")
```



```
>
```

8.1.1 Verbesserungen

An dem Beispiel ist leicht zu sehen, dass es zum Zeichnen mit Hilfe von Turtle-Grafik oft erforderlich ist, Zwischenwerte wie `t1`, `t2` etc., an die nächste Prozedur weiterzureichen, die Werte ansonsten aber nicht weiterverwendet werden. Beispielsweise werden in der obigen Definition von square die Variablen `t1`, ..., `t4` nur gebraucht, um die Prozeduren `right-angle`, `(turn -90)` und `right-angle` hintereinander auszuführen.

Um solche Fälle einfach programmieren zu können, enthält das Turtle-Teachpack die Prozedur `sequence`. Damit können wir eine zu `right-angle` äquivalente Version wesentlicher einfacher aufschreiben:

```
(define right-angle2
  (sequence (draw 20) (turn -90) (draw 20)))
```

Ebenso wie `right-angle` können wir square leichter schreiben als:

```
(define square2
  (sequence right-angle (turn -90) right-angle))
```

8.2 Prozeduren

```
turtle : signature
```

Dies ist die Signatur für Turtles.

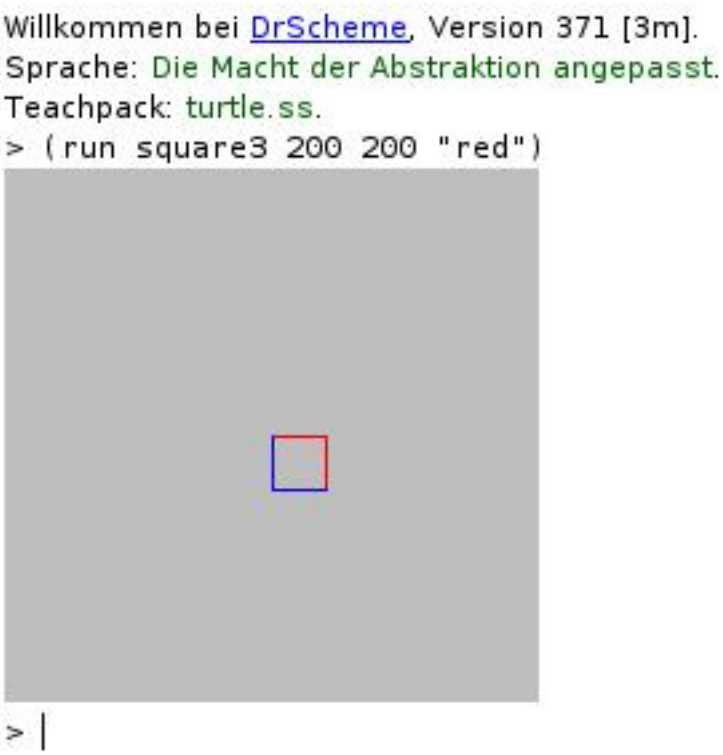
```
set-color : (color -> (turtle -> turtle))
```

Diese Prozedur ist eine Prozedurfabrik. Sie liefert als Ergebnis eine Prozedur, die auf eine Turtle anwendbar ist. Wendet man das Ergebnis auf eine Turtle an, so ändert dies die Farbe mit der gezeichnet wird.

Folgender Code

```
(define square3
  (sequence right-angle (turn -90) (set-color "blue") right-
angle))
```

liefert dieses Bild:




```
turn : (number -> (turtle -> turtle))
```

Diese Prozedur ist eine Prozedurfabrik. Sie liefert als Ergebnis eine Prozedur, die auf eine Turtle anwendbar ist. Wendet man das Ergebnis auf eine Turtle an, so ändert sich die Blickrichtung der Turtle um die gegebene Gradzahl gegen den Uhrzeigersinn.

```
draw : (number -> (turtle -> turtle))
```

Diese Prozedur ist eine Prozedurfabrik. Sie liefert als Ergebnis eine Prozedur, die auf eine Turtle anwendbar ist. Wendet man das Ergebnis auf eine Turtle an, so bewegt sich die Schildkröte um die gegebene Anzahl von Pixel und zeichnet dabei eine Linie.

```
move : (number -> (turtle -> turtle))
```

Diese Prozedur ist eine Prozedurfabrik. Sie liefert als Ergebnis eine Prozedur, die auf ein Turtle anwendbar ist. Wendet man das Ergebnis auf eine Turtle an, so bewegt sich die Schildkröte um die gegebene Anzahl von Pixel, zeichnet dabei aber keine Linie.

```
run : ((turtle -> turtle) number number color -> image)
```

Diese Prozedur wendet die übergebene Prozedur von Turtle nach Turtle auf die initiale Schildkröte an und zeigt das daraus resultierende Bild an. Der zweite Parameter ist die Höhe des Bilds, der dritte Parameter die Breite des Bilds und der vierte Parameter die Farbe, mit der gezeichnet wird.

```
sequence : ((turtle -> turtle) ... -> (turtle -> turtle))
```

Diese Prozedur nimmt eine beliebige Anzahl von Turtle-Veränderungen (d.h. Prozeduren mit Signatur `turtle -> turtle`) und erstellt eine neue Prozedur, die die Veränderungen der Reihe nach von links nach rechts abarbeitet.

9 Abspielen von Audio-Dateien: `"sound.ss"`

Note: This is documentation for the `"sound.rkt"` teachpack that goes with the German textbook *Die Macht der Abstraktion*.

Dieses Teachpack definiert eine Prozedur zum Abspielen einer Audio-Datei. Diese Prozedur ist je nach Plattform unterschiedlich realisiert, und funktioniert möglicherweise nicht auf jedem Rechner.

```
play-sound-file : (string -> unspecific)
```

Der Aufruf `(play-sound-file f)` spielt die Audio-Datei mit dem Namen `f` ab.

```
background-play-sound-file : (string -> unspecific)
```

Der Aufruf `(background-play-sound-file f)` spielt die Audio-Datei mit dem Namen `f` im Hintergrund ab, also ohne dass das Scheme-Programm anhält.

10 3D-Liniengraphik: "line3d.ss"

Note: This is documentation for the "line3d.rkt" teachpack that goes with the German textbook *Die Macht der Abstraktion*.

Dieses teachpack definiert Prozeduren für lineare Algebra und 3D-Rendering:

10.1 Szenen erzeugen

```
render-scene
 : (natural natural (list-of line3d) matrix4x4 -> image)
```

Der Aufruf (render-scene width height scene camera-matrix)erzeugt die Szene in ein Bild mit Breite width und Höhe height. Position, Orientierung und Projektion werden durch die camera-matrix festgelegt.

```
create-camera-matrix
 : (vec3 vec3 number natural natural -> matrix4x4)
```

Der Aufruf (create-camera-matrix position lookat vertical-fov width height) erzeugt eine 4x4 Matrix. Diese kodiert eine Kamera an der Position position, die auf die Position lookat schaut. vertical-fov bezeichnet das *vertikale Feld* der Szene.

Zum Beispiel:

```
; scene-data (simple box example)
(define box
  (create-box 1.0 1.0 1.0 "brown"))
; screen
(define screenWidth 320)
(define screenHeight 240)
; camera
(define pos (make-vec3 5 5 3))
(define lookat (make-vec3 0 0 0))
(define camera
  (create-camera-matrix pos lookat 70.0 screenWidth screenHeight))
; render image
(render-scene screenWidth screenHeight box camera)
```

10.2 3D-Vektoren

```
vec3 : signature
```

Ein *3D-Vektor* (Name: `vec3`) ist ein Record, der durch den Aufruf `make-vec3` erstellt wird.

```
make-vec3 : (number number number -> vec3)
```

`(make-vec3 x y z)` erstellt einen Vektor (x,y,z).

```
add-vec3 : (vec3 vec3 -> vec3)
```

`(add-vec3 a b)` gibt die Summe von `a` und `b` zurück.

```
sub-vec3 : (vec3 vec3 -> vec3)
```

`(sub-vec3 a b)` gibt die Differenz zwischen `a` und `b` zurück.

```
mult-vec3 : (vec3 number -> vec3)
```

`(mult-vec3 a s)` gibt den das Produkt von `a` und `s` zurück.

```
div-vec3 : (vec3 number -> vec3)
```

`(div-vec3 a s)` gibt den das Produkt von `a` und dem Kehrwert von `s` zurück.

```
dotproduct-vec3 : (vec3 vec3 -> number)
```

`(dotproduct-vec3 a b)` gibt das Produkt von `a` und `b` zurück.

```
normQuad-vec3 : (vec3 -> number)
```

`(normQuad-vec3 a)` gibt die quadrierte Norm/Länge $|a|^2$ eines Vektors `a` zurück (Quadrat der Euklidischen Norm.)

```
norm-vec3 : (vec3 -> number)
```

`(norm-vec3 a)` gibt die Norm/Länge $|a|$ eines Vektors a zurück (Euklidische Norm.)

```
normalize-vec3 : (vec3 -> vec3)
```

`(normalize-vec3 a)` normalisiert `a`.

```
crossproduct-vec3 : (vec3 vec3-> vec3)
```

`(crossproduct-vec3 a b)` gibt das Kreuzprodukt von `a` und `b` zurück (einen Vektor der senkrecht auf `a` und `b` steht).

10.3 4D-Vektoren

```
vec4 : signature
```

Ein *4D-Vektor* `vec4` ist ein 4D-Vektor. Folgende Prozeduren werden bereitgestellt:

```
make-vec4 : (number number number number -> vec4)
```

`(make-vec4 a b c d)` erzeugt einen Vektor aus `a`, `b`, `c` und `d`.

```
add-vec4 : (vec4 vec4 -> vec4)
```

`(add-vec4 a b)` gibt die Summe von `a` und `b` zurück.

```
sub-vec4 : (vec4 vec4 -> vec4)
```

`(sub-vec4 a b)` gibt die Differenz zwischen `a` und `b` zurück.

```
mult-vec4 : (vec4 number -> vec4)
```

`(mult-vec4 a s)` gibt den das Produkt von `a` und `s` zurück.

```
div-vec4 : (vec4 number -> vec4)
```

`(div-vec4 a s)` gibt den das Produkt von `a` und dem Kehrwert von `s` zurück.

```
dotproduct-vec4 : (vec3 vec4 -> number)
```

`(dotproduct-vec4 a b)` gibt die quadrierte Norm/Länge $|a|^2$ eines Vektors `a` zurück (Quadrat der Euklidischen Norm.)

```
normQuad-vec4 : (vec4 -> number)
```

`(normQuad-vec4 a)` gibt die quadrierte Norm/Länge $|a|^2$ eines Vektors `a` zurück (Quadrat der Euklidischen Norm.)

```
norm-vec4 : (vec4 -> number)
```

`(norm-vec4 a)` gibt die Norm/Länge |a| eines Vektors a zurück (Euklidische Norm)

```
normalize-vec4 : (vec4 -> vec4)
```

`(normalize-vec4 a)` normalisiert `a`.

```
expand-vec3 : (vec3 number -> vec4)
```

`(expand-vec3 a s)` gibt den 4D-Vektor mit `s` als letze Komponente zurück (erweitert `a` mit `s`).

10.4 4x4 Matrizen

```
matrix4x4 : signature
```

Eine *Matrix* `matrix4x4` ist ein Record, der durch den Aufruf `make-matrix4x4` erstellt wird.

```
make-matrix4x4 : (vec4 vec4 vec4 vec4 -> matrix4x4)
```

`(make-matrix4x4 a b c d)` erstellt eine Matrix aus `a`, `b`, `c` und `d`.

```
create-matrix4x4 : (vec3 vec3 vec3 vec3 -> matrix4x4)
```

`(create-matrix4x4 a b c d)` erweitert jeden Vektor in einen 4D-Vektor und kombiniert diese zu einer Matrix `a`, `b`, `c` und `d`, wobei `a`, `b`, `c` mit 0 und `d` mit 1 erweitert wird, um eine homogene Matrix zu erzeugen.

```
transpose-matrix4x4 : (matrix4x4 -> matrix4x)
```

`(transpose-matrix4x4 m)` erstellt die transponierte Matrix `m^T`.

```
multiply-matrix-vec4 : (matrix vec4 -> vec4)
```

`(multiply-matrix-vec4 m v)` gibt die Matrix `mv` zurück. Die `w`-Komponente ist nicht normalisiert.

```
transform-vec3 : (matrix4x4 vec3 -> vec3)
```

`(transform-vec3 m v)` erweitert `v` mit 1, multipliziert `m` mit `v` und dividiert das Ergebnis mit `w`.

```
multiply-matrix : (matrix4x4 matrix4x4 -> matrix4x4)
```

`(multiply-matrix a b)` gibt die Matrix `a*b` zurück.

```
create-translation-matrix : (vec3 -> matrix4x4)
```

`(create-translation-matrix v)` gibt die Translations-Matrix zurück.

```
create-rotation-x-matrix : (number -> matrix4x4)
```

`(create-rotation-x-matrix a)` gibt eine Rotations-Matrix zurück die um die X-Achse mit dem Winkel `a` rotiert.

```
create-rotation-y-matrix : (number -> matrix4x4)
```

`(create-rotation-y-matrix a)` gibt eine Rotations-Matrix zurück die um die Y-Achse mit dem Winkel `a` rotiert.

```
create-rotation-z-matrix : (number -> matrix4x4)
```

`(create-rotation-z-matrix a)` gibt eine Rotations-Matrix zurück die um die Z-Achse mit dem Winkel `a` rotiert.

```
create-lookat-matrix : (vec3 vec3 vec3 -> matrix4x4)
```

`(create-lookat-matrix pos lookat up)` gibt eine Kameramatrix. Ursprungspunkt ist `pos`, die Z-Achse zeigt auf `lookat`.

```
create-projection-matrix : (number -> matrix4x4)
```

`(create-projection-matrix vertical-fov/2)` erzeugt eine Projektions-Matrix. `vertical-fov`/2 gibt den vertikalen Winkel der Ansicht dividiert durch 2 an.

```
create-viewport-matrix : (natural natural -> matrix4x4)
```

`(create-viewport-matrix width height)` gibt einen Ausschnitt an.

10.5 3d-Linien

```
line3d : signature
```

Eine *3d-Linie* `line3d` ist ein Record, der durch den Aufruf `make-line3d` erstellt wird und eine farbige Linie zwischen zwei Punkten im 3-dimensionalen Raum darstellt.

```
make-line3d : (vec3 vec3 color -> line3d)
```

`(make-line3d a b col)` erstellt eine 3D-Linie zwischen Punkt `a` und Punkt `b` mit der Farbe `col`.

```
line3d-a : (line3d -> vec3)
```

extrahiert den Anfangspunkt einer 3D-Linie.

```
line3d-b : (line3d -> vec3)
```

extrahiert den Endpunkt einer 3D-Linie.

```
line3d-color : (line3d -> color)
```

extrahiert die Farbe einer 3D-Linie.

```
create-box : (number number number color -> (list-of line3d))
```

`(create-box width height depth color)` erstellt eine Box am Punkt (0,0,0) in den angebenen Ausmaßen.

```
transform-primitive-list
 : ((list-of line3d) matrix4x4 -> (list-of line3d))
```

`(transform-primitive-list scene transformationr)` wendet `transformation` auf alle Punkte der Linien in `scene` an und gibt diese zurück.

11 DMdA: Sprachen als Libraries

Note: This is documentation for the language levels that go with the German textbook *Die Macht der Abstraktion*.

11.1 *Die Macht der Abstraktion* - Anfänger

```
(require deinprogramm/DMdA-beginner)
```
package: deinprogramm

Das Modul deinprogramm/DMdA-beginner implementiert die Anfängersprache für *Die Macht der Abstraktion*; siehe §1 “Die Macht der Abstraktion - Anfänger”.

11.2 *Die Macht der Abstraktion*

```
(require deinprogramm/DMdA-vanilla)
```
package: deinprogramm

Das Modul deinprogramm/DMdA-vanilla implementiert die Standardsprache für *Die Macht der Abstraktion*; siehe §2 “Die Macht der Abstraktion”.

11.3 *Die Macht der Abstraktion* mit Zuweisungen

```
(require deinprogramm/DMdA-assignments)
```
package: deinprogramm

Das Modul deinprogramm/DMdA-assignments implementiert die Sprachebene für *Die Macht der Abstraktion* mit Zuweisungen und Mutationen; siehe §3 “Die Macht der Abstraktion mit Zuweisungen”.

11.4 *Die Macht der Abstraktion* - fortgeschritten

```
(require deinprogramm/DMdA-advanced)
```
package: deinprogramm

Das Modul deinprogramm/DMdA-advanced implementiert die fortgeschrittene Sprachebene für *Die Macht der Abstraktion*; siehe §4 “Die Macht der Abstraktion fortgeschritten”.

Index