Sprachebenen und Material zu *Schreibe Dein Programm!*

Version 7.3

May 13, 2019

Note: This is documentation for the teachpacks that go with the German textbook *Schreibe Dein Programm!*.

Das Material in diesem Handbuch ist für die Verwendung mit Buch *Schreibe Dein Programm!* gedacht.

Contents

Index **67**

1 Schreibe Dein Programm! - Anfänger

This is documentation for the language level *Schreibe Dein Programm! - Anfänger* to go with the German textbook *Schreibe Dein Programm!*.

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

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

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

       expr = (expr expr ...) ; Prozedurapplikation
             | #t
             | #f
             | number
             | string
             | (lambda (id ...) expr)
             | (λ (id ...) expr)
             | id ; Name
             | (cond (expr expr) (expr expr) ...)
             | (cond (expr expr) ... (else expr))
             | (if expr expr)
             | (and expr ...)
             | (or expr ...)
             | (match expr (pattern expr) ...)
             | (signature sig)
             | (for-all ((id sig) ...) expr)
             | (==> expr expr)

      field = id
             | (id id)

        sig = id
             | (predicate expr)
             | (one-of expr ...)
             | (mixed sig ...)
             | (sig ... -> sig) ; Prozedur-Signatur
             | (list-of sig)
             | (nonempty-list-of sig)
```

```
          | %a %b %c ; Signatur-Variable
          | (combined sig ...)

  pattern = #t
          | #f
          | number
          | string
          | id
          | (constructor pattern ...)

test-case = (check-expect expr expr)
          | (check-within expr expr expr)
          | (check-member-of expr expr ...)
          | (check-satisfied 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
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 *expr*.

1.2 Record-Typ-Definitionen

```
(define-record-functions t c p (sel sig) ...)
(define-record-functions t c (sel sig) ...)
```

Die `define-record-functions`-Form ist eine Definition für einen neuen Record-Typ. Dabei ist *t* der Name der Record-Signatur, *c* der Name des Konstruktors und *p* der (optionale) Name des Prädikats.

Jedes `(sel sig)` beschreibt ein *Feld* des Record-Typs, wobei *sel* der Name des Selektors für das Feld und *sig* die Signatur des Feldes ist.

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` / λ

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

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

```
(λ (id ...) expr)
```

λ ist ein anderer Name für `lambda`.

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.

```
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-satisfied expr pred)
```

Ähnlich wie `check-expect`: Der Testfall überprüft, ob der Wert des Ausdrucks *`expr`* vom Prädikat *`pred`* erfüllt wird - das bedeutet, daß die Prozedur *`pred`* den Wert `#t` liefert, wenn sie auf den Wert von *`expr`* angewendet wird.

Der folgende Test wird also bestanden:

```
(check-satisfied 1 odd?)
```

Der folgende Test hingegen wird hingegen nicht bestanden:

```
(check-satisfied 1 even?)
```

```
(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-functions` definiert wurden - wohl aber für Signaturen, die mit dem durch `define-record-functions-parametric` definierten Signaturkonstruktor erzeugt wurden.

1.13 Pattern-Matching

```
(match expr (pattern expr) ...)

pattern = id
        | #t
        | #f
        | string
        | number
        | (constructor pattern ...)
```

Ein `match`- Ausdruck führt eine Verzweigung durch, ähnlich wie `cond`. Dazu wertet match zunächst einmal den Ausdruck *expr* nach dem match zum Wert *v* aus. Es prüft dann nacheinander jeden Zweig der Form (*pattern expr*) dahingehend, ob das Pattern *pattern* darin auf den Wert *v* paßt ("matcht"). Beim ersten passenden Zweig (*pattern expr*) macht `match` dann mit der Auswertung voh *expr* weiter.

Ob ein Wert *v* paßt, hängt von *pattern* ab:

- Ein Pattern, das ein Literal ist (`#t`, `#f`, Zeichenketten *string*, Zahlen *number*) paßt nur dann, wenn der Wert *v* gleich dem Pattern ist.
- Ein Pattern, das ein Bezeichner *id* ist, paßt auf *jeden* Wert. Der Bezeichner wird dann an diesen Wert gebunden und kann in dem Ausdruck des Zweigs benutzt werden.
- Ein Pattern (*constructor pattern ...*), bei dem *constructor* ein Record-Konstruktor ist (ein *Konstruktor-Pattern*), paßt auf *v*, falls *v* ein passender Record ist, und dessen Felder auf die entsprechenden Patterns passen, die noch im Konstruktor-Pattern stehen.

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

```
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 Schreibe Dein Programm!

This is documentation for the language level *Schreibe Dein Programm!* to go with the German textbooks *Schreibe Dein Programm!*.

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

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

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

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

        field = id
              | (id id)

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

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

    pattern = #t
              | #f
              | number
              | string
              | id
              | (constructor pattern ...)
              | empty
              | (make-pair pattern pattern)
              | (list pattern ...)

  test-case = (check-expect expr expr)
              | (check-within expr expr expr)
              | (check-member-of expr expr ...)
              | (check-satisfied 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)
filter : ((%a -> boolean) (list-of %a) -> (list-of %a))
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)
rest : ((list-of %a) -> (list-of %a))
reverse : ((list-of %a) -> (list-of %a))
```
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

```
empty-list
```

Signatur für die leere Liste.

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

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

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

Diese Signatur ist dann für einen Wert gültig, wenn dieser eine nichtleere 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 Pattern-Matching

```
(match expr (pattern expr) ...)

pattern = ...
        | empty
        | (make-pair pattern pattern)
        | (list pattern ...)
```

Zu den Patterns aus der "Anfänger"-Sprache kommen noch drei neue hinzu:

- Das Pattern *empty* paßt auf die leere Liste.
- Das Pattern (*make-pair pattern pattern*) paßt auf Paare, bei denen die beiden inneren Patterns auf `first` bzw. `rest` passen.
- Das Pattern [(list pattern ...)] paßt auf Listen, die genauso viele Elemente haben, wie Teil-Patterns im *list*-Pattern stehen und bei denen die inneren Patterns auf die Listenelemente passen.

2.4 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 Cons aus Element und Liste

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

feststellen, ob ein Wert ein Cons ist

```
empty : list
```

die leere Liste

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

feststellen, ob ein Wert die leere Liste ist

```
filter : ((%a -> boolean) (list-of %a) -> (list-of %a))
```

Alle Elemente einer Liste extrahieren, für welche die Funktion #t liefert.

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

erstes Element eines Cons 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

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

Rest eines Cons extrahieren

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

Liste in umgekehrte Reihenfolge bringen

```
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)
```

Funktion von vorn nach hinten auf alle Elemente einer Liste anwenden

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

Funktion 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 Schreibe Dein Programm! - fortgeschritten

This is documentation for the language level *Schreibe Dein Programm - fortgeschritten* that goes with the German textbook *Schreibe Dein Programm!*.

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

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

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

  field-spec = id
             | (id id)

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

        expr = (expr expr ...) ; Prozedurapplikation
             | #t
             | #f
             | number
             | string
             | (lambda (id ...) expr)
             | (λ (id ...) expr)
             | id ; Name
             | (cond (expr expr) (expr expr) ...)
             | (cond (expr expr) ... (else expr))
             | (if expr expr)
             | (and expr ...)
             | (or expr ...)
             | (match expr (pattern expr) ...)
             | (signature sig)
             | (for-all ((id sig) ...) expr)
             | (==> expr expr)
```

```
        | (let ((id expr) (... ...)) expr)
        | (letrec ((id expr) (... ...)) expr)
        | (let* ((id expr) (... ...)) expr)
        | quoted
        | 'quoted ; Quote-Literal

  field = id
        | (id id)

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

pattern = #t
        | #f
        | number
        | string
        | id
        | (constructor pattern ...)
        | (make-pair pattern pattern)
        | (list pattern ...)
        | 'quoted

test-case = (check-expect expr expr)
          | (check-within expr expr expr)
          | (check-member-of expr expr ...)
          | (check-satisfied 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)
filter : ((%a -> boolean) (list-of %a) -> (list-of %a))
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)
rest : ((list-of %a) -> (list-of %a))
reverse : ((list-of %a) -> (list-of %a))
```

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

Verschiedenes

```
apply : (function (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 Quote-Literal

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

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

3.2 Signaturen

```
symbol
```

Signatur für Symbole.

3.3 Pattern-Matching

```
(match expr (pattern expr) ...)

pattern = ...
        | 'quoted
```

Zu den Patterns kommt noch eins hinzu:

- Das Pattern `'quoted` paßt auf genau auf Werte, welche die gleiche externe Repräsentation wie `quoted` haben.

3.4 Definitionen

```
(define id expr)
```

Diese Form ist wie in den unteren Sprachebenen.

3.5 `lambda` / λ

```
(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.

```
(λ (id id ... . id) expr)
```

λ ist ein anderer Name für `lambda`.

3.6 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 Cons aus Element und Liste

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

feststellen, ob ein Wert ein Cons ist

```
empty : list
```

die leere Liste

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

feststellen, ob ein Wert die leere Liste ist

```
filter : ((%a -> boolean) (list-of %a) -> (list-of %a))
```

Alle Elemente einer Liste extrahieren, für welche die Funktion #t liefert.

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

erstes Element eines Cons 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

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

Rest eines Cons extrahieren

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

Liste in umgekehrte Reihenfolge bringen

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

Sind zwei Symbole gleich?

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

feststellen, ob ein Wert ein Symbol ist

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

Funktion 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)
```

Funktion von vorn nach hinten auf alle Elemente einer Liste anwenden

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

Funktion 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 Konstruktionsanleitungen 1 bis 10

This documents the design recipes of the German textbook *Schreibe Dein Programm!*.

Contents

4.1 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.

4.2 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.

4.3 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-functions sig
  constr pred?
  (select_1 sig_n)
  ...
  (select_n sig_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))
```

4.4 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.

4.5 zusammengesetzte Daten als Ausgabe

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

4.6 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.

4.7 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)))))))
```

4.8 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))))))
```

4.9 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 sdp: Sprachen als Libraries

Note: This is documentation for the language levels that go with the German textbook *Schreibe Dein Programm!*.

5.1 *Schreibe Dein Programm* - Anfänger

```
(require deinprogramm/sdp/beginner)
```
package: `deinprogramm`

Das Modul `deinprogramm/sdp/beginner` implementiert die Anfängersprache für *Schreibe Dein Programm!*; siehe §1 “Schreibe Dein Programm! - Anfänger”.

5.2 *Schreibe Dein Programm!*

```
(require deinprogramm/sdp/vanilla)
```
package: `deinprogramm`

Das Modul `deinprogramm/sdp/vanilla` implementiert die Standardsprache für *Schreibe Dein Programm!*; siehe §2 “Schreibe Dein Programm!”.

5.3 *Schreibe Dein Programm!* - fortgeschritten

```
(require deinprogramm/sdp/advanced)
```
package: `deinprogramm`

Das Modul `deinprogramm/sdp/advanced` implementiert die fortgeschrittene Sprachebene für *Schreibe Dein Programm!*; siehe §3 “Schreibe Dein Programm! - fortgeschritten”.

Index