Sprachebenen und Material zu *Schreibe Dein Programm!*

Version 9.2.0.6

June 30, 2026

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 **74**

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 id id (id id) ...)
             | (define-record id id id (id id) ...)
             | (define-record (id id ...) id id (id id) ...)
             | (define-singleton id id id)
             | (: id sig)

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

         sig = id
             | (predicate expr)
             | (enum expr ...)
             | (mixed sig ...)
             | (sig ... -> sig) ; Funktions-Signatur
             | %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)
equal? : (any any -> boolean)
false? : (any -> boolean)
not : (boolean -> boolean)
true? : (any -> boolean)
```

Listen

Zeichenketten

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

Symbole

Verschiedenes

```
read : (-> any)
signature? : (any -> boolean)
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 Namen an den Wert von *expr*. Eine Definition kann ganz außen vorkommen, dann ist sie global und kann überall verwendet werden. Eine Definition kann aber auch innerhalb eines `lambda`-Ausdrucks oder innerhalb von `cond`- und `match`-Zweigen vorkommen, dann ist sie lokal und nur dort gültig.

1.2 Record-Typ-Definitionen

```
(define-record type
  constructor
  (selector signature) ...)
(define-record type
  constructor
  predicate?
  (selector signature) ...)
```

Die `define-record`-Form ist eine Definition für einen neuen Record-Typ. Dabei ist *type* der Name der Record-Signatur, *constructor* der Name des Konstruktors und *predicate?* der (optionale) Name des Prädikats.

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

1.3 Record-Typ-Definitionen mit Signatur-Parametern

```
(define-record (type-constructor signature-parameter ...)
   constructor
   (selector signature) ...)
(define-record (type-constructor signature-parameter ...)
   constructor
   predicate?
   (selector signature) ...)
```

Diese Variante von `define-record` erlaubt die Verwendung von Signatur-Parametern: Statt einer konkreten Signatur `type` wie oben definiert die Form einen Signatur-Konstruktor *`type-constructor`*, also eine Funktion, die Signaturen als Argumente akzeptiert, entsprechend den Signatur-Parametern *`signature-parameter`*. Diese Signatur-Parameter können in den Signaturen *`signature`* der Felder verwendet werden.

1.4 Singleton-Definitionen

```
(define-singleton signature name?)
(define-singleton signature name predicate?)
```

Diese Form definiert ein Singleton, also einen einzelnen Wert namens *`name`*, der mit Hilfe von *`predicate?`* von allen anderen Werten unterschieden werden kann. Die dazu passende Signatur ist *`signature`*.

1.5 Funktionsapplikation

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

Dies ist eine Funktionsanwendung oder -applikation. Alle *`expr`*s werden ausgewertet: Der Operator (also der erste Ausdruck) muss eine Funktion 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 Funktion durch die Argumente, also die Werte der Operanden ersetzt wurden.

1.6 `#t` and `#f`

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

1.7 `lambda` / λ

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

Ein Lambda-Ausdruck ergibt bei der Auswertung eine Funktion.

Im Rumpf können interne Definitionen vorkommen, die aber nur in *expr* gelten.

```
(λ (id ...) definition ... expr)
```

λ ist ein anderer Name für `lambda`.

1.8 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.

1.9 `cond`

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

In einem `cond`-Zweig können lokale Definitionen mit `define` vorkommen.

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

1.10 `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).

1.11 `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`.

1.12 `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`.

1.13 Signaturen

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

1.13.1 `signature`

```
(signature sig)
```

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

1.13.2 Signaturdeklaration

```
(: id sig)
```

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

1.13.3 Eingebaute Signaturen

```
number
```

Signatur für beliebige Zahlen.

```
real
```

Signatur für reelle Zahlen.

```
rational
```

Signatur für rationale Zahlen.

```
integer
```

Signatur für ganze Zahlen.

```
(integer-from-to low high) → signature?
  low : integer?
  high : integer?
```

Signatur für ganze Zahlen zwischen *low* und *high*.

```
natural
```

Signatur für ganze, nichtnegative Zahlen.

```
boolean
```

Signatur für boolesche Werte.

```
true
```

Signatur für `#t`.

```
false
```

Signatur für `#f`.

```
string
```

Signatur für Zeichenketten.

```
any
```

Signatur, die auf alle Werte gültig ist.

```
signature
```

Signatur für Signaturen.

```
property
```

Signatur für Eigenschaften.

1.13.4 `predicate`

```
(predicate expr)
```

Bei dieser Signatur muss *`expr`* als Wert ein Prädikat haben, also eine Funktion, 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.

1.13.5 `enum`

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

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

1.13.6 `mixed`

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

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

1.13.7 Funktions-Signatur

```
->
```

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

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

1.13.8 Signatur-Variablen

```
%a
```

```
%b
```

```
%c
```

```
...
```

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

1.13.9 `combined`

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

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

1.14 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 Funktion *`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 viele Signaturen der Fall, aber nicht für solche mit Signaturvariablen.

1.15 Pattern-Matching

```
(match expr (pattern definition ... 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.
- Das Pattern `...` passt auf jeden Wert, ohne dass ein Bezeichner gebunden wird.
- 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.

1.16 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)))))
```

1.17 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

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

feststellen, ob zwei Werte gleich sind

```
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-of 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-of 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

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

feststellen, ob ein Wert eine Signatur ist

```
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 id id (id id) ...)
            | (define-record id id id (id id) ...)
            | (define-record (id id ...) id id (id id) ...)
            | (define-singleton id id id)
            | (: id sig)

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

        sig = id
            | (predicate expr)
            | (enum expr ...)
            | (mixed sig ...)
            | (sig ... -> sig) ; Funktions-Signatur
            | %a %b %c ; Signatur-Variable
```

```
          | (combined sig ...)
          | (list-of sig)
          | (cons-list-of sig)

  pattern = #t
          | #f
          | number
          | string
          | id
          | ...
          | (constructor pattern ...)
          | empty
          | (cons 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)
equal? : (any 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-of 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-of string) -> string)
```

Symbole

Verschiedenes

```
for-each : ((%a -> %b) (list-of %a) -> unspecific)
map : ((%a -> %b) (list-of %a) -> (list-of %b))
read : (-> any)
signature? : (any -> boolean)
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.

```
(cons-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 definition ... expr) ...)

pattern = ...
        | empty
        | (cons 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 *(cons pattern pattern)* passt auf Cons-Listen, 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

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

feststellen, ob zwei Werte gleich sind

```
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-of 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-of string) -> string)
```

Eine Liste von Zeichenketten in eine Zeichenkette umwandeln

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

Funktion von vorn nach hinten auf alle Elemente einer Liste anwenden

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

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

feststellen, ob ein Wert eine Signatur ist

```
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 id id (id id) ...)
             | (define-record id id id (id id) ...)
             | (define-record (id id ...) id id (id id) ...)
             | (define-singleton id id id)
             | (: id sig)

  field-spec = id
             | (id id)

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

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

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

     sig = id
          | (predicate expr)
          | (enum expr ...)
          | (mixed sig ...)
          | (sig ... -> sig) ; Funktions-Signatur
          | %a %b %c ; Signatur-Variable
          | (combined sig ...)
          | (list-of sig)
          | (cons-list-of sig)

 pattern = #t
          | #f
          | number
          | string
          | id
          | ...
          | (constructor pattern ...)
          | (cons 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)
equal? : (any 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-of 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-of string) -> string)
```

Symbole

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

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

Verschiedenes

```
apply : (function (list-of %a) -> %b)
eq? : (%a %b -> boolean)
for-each : ((%a -> %b) (list-of %a) -> unspecific)
map : ((%a -> %b) (list-of %a) -> (list-of %b))
read : (-> any)
signature? : (any -> boolean)
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 Funktion 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 `begin`

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

Ein `begin`-Ausdruck wertet die *expr*s nacheinander aus und liefert das Ergebnis des letzten *expr*.

3.7 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

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

feststellen, ob zwei Werte gleich sind

```
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-of 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-of 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-of %a) -> %b)
```

Funktion auf Liste ihrer Argumente anwenden

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

zwei Werte auf Selbheit testen

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

Funktion von vorn nach hinten auf alle Elemente einer Liste anwenden

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

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

feststellen, ob ein Wert eine Signatur ist

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

Programmm mit Fehlermeldung abbrechen

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

Zeilenumbruch ausgeben

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

Zeichenkette in REPL ausgeben

4 Konstruktionsanleitungen

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

Contents

4.1 Ablauf

Gehe bei der Konstruktion einer Funktion in folgender Reihenfolge vor:

- Kurzbeschreibung
- Datenanalyse
- Signatur
- Testfälle
- Gerüst
- Schablonen
- Rumpf

4.2 Kurzbeschreibung

Schreibe für die Funktion zunächst einen Kommentar, der ihren Zweck kurz beschreibt. Ein Satz, der auf eine Zeile passen sollte, reicht. Beispiel:

```
; monatlichen Rechnungsbetrag für Tarif Billig-Strom berechnen
```

4.3 Signatur-Deklaration

Schreibe für die Funktion direkt unter die Kurzbeschreibung eine Signatur-Deklaration. Dazu denke Dir zunächst einen möglichst aussagekräftigen Namen aus. Überlege dann, welche Sorten die Ein- und Ausgaben haben und schreibe dann eine Signatur, welche die Ein- und Ausgaben der Funktion möglichst präzise beschreiben. Beispiel:

```
(: billig-strom (natural -> rational))
```

Achte bei den Zahlen-Signaturen besonders auf eine möglichst präzise Signatur. Bei `billig-strom` wäre auch die Signatur `(number -> number)` korrekt, aber nicht so genau.

4.4 Tests

Schreibe unter die Signatur Tests für die Funktion. Denke Dir dafür möglichst möglichst einfache, aber auch möglichst interessante Beispiele für Aufrufe der Funktion auf und lege fest, was dabei herauskommen soll. Mache aus den Beispielen Tests mit `check-expect`. Beispiel:

```
(check-expect (billig-strom 0) 4.9)
(check-expect (billig-strom 10) 6.8)
(check-expect (billig-strom 20) 8.7)
(check-expect (billig-strom 30) 10.6)
```

Achte darauf, dass die Tests dafür sorgen, dass der Code Deiner Funktion durch die Tests vollständig abgedeckt wird.

4.5 Gerüst

Schreibe unter die Tests ein Gerüst für die Funktion: Dazu übernimmst Du den Namen aus der Signatur-Deklaration in eine Funktionsdefinition wie zum Beispiel:

```
(define billig-strom
  (lambda (...)
    ...))
```

Denke Dir Namen für die Eingaben der Funktion aus. Das müssen genauso viele sein, wie die Signatur Eingaben hat. Schreibe dann diese Namen als Eingaben in die `lambda`-Abstraktion. Beispiel:

```
(define billig-strom
  (lambda (kWh)
    ...))
```

4.6 Rumpf

Als letzten Schritt fülle mit Hilfe des Wissens über das Problem den Rumpf der Funktion aus.

```
(define billig-strom
  (lambda (kWh)
    (+ 4.9 (* 0.19 kWh))))
```

4.7 Datenanalyse

Suche in der Aufgabenstellung nach problemrelevanten Größen; Kandidaten sind immer die Substantive. Schreibe für jede dieser Größen eine Datendefinition, es sei denn, diese ist aus dem Kontext offensichtlich.

Wenn es für die Datendefinition noch keine Signatur gibt, schreibe eine Signaturdefinition dazu. Schreibe außerdem Beispiele auf und schreibe jeweils einen Kommentar, der die Beziehung zwischen Daten und Information beschreibt.

4.8 Fallunterscheidung: Datenanalyse

Versuche, für die Datendefinition eine Formulierung "... ist eins der folgenden" zu finden. Wenn das möglich ist, beschreibt Deine Datendefinition eine *Fallunterscheidung*. Schreibe dann eine Auflistung aller Fälle, jeder Fall auf eine separate Zeile:

```
; Ein X ist eins der folgenden:
; - Fall 1
; - Fall 2
; - ...
; - Fall n
```

4.9 Aufzählung: Datenanalyse

Falls Deine Datendefinition eine Fallunterscheidung beschriebt und jeder der Fälle nur aus einem einzelnen Wert besteht, handelt es sich um eine *Aufzählung*.

Schreibe für jede Aufzählung eine Signaturdefinition der Form:

```
(define s (signature (enum ...)))
```

Achte darauf, dass die Anzahl der Fälle der Signaturdefinition der Anzahl der Fälle der Datendefinition entspricht.

4.10 Schablone

Wenn Du das Gerüst fertiggestellt hast, benutze die Signatur und die dazugehörigen Datendefinitionen, um Konstruktionsanleitungen mit ein oder mehreren Schablonen auszuwählen und übertrage die Elemente der Schablonen in den Rumpf der Funktion.

4.11 Fallunterscheidung: Schablone

Wenn Du eine Funktion schreibst, die eine Fallunterscheidung als Eingabe verarbeitet, schreibe als Schablone in den Rumpf eine Verzweigung mit sovielen Zweigen, wie es in der Fallunterscheidung Fälle gibt, nach folgendem Muster:

```
(define f
  (lambda (a)
    (cond
      (... ....)
      ...
      (... ...))))
```

Schreibe danach Bedingungen in die Zweige, welche die einzelnen Fälle voneinander unterscheiden.

4.12 boolesche Fallunterscheidung: Schablone

Wenn sich das Ergebnis einer Funktion nach einer booleschen Größe richtet, welche die Funktion mit Hilfe der Eingaben berechnen kann, benutze als Schablone im Rumpf eine binäre Verzweigung:

```
(define f
  (lambda (e)
    (if ... ; hier wird die boolesche Größe berechnet
        ...
        ....)))
```

4.13 Zusammengesetzte Daten: Datenanalyse

Zusammengesetzte Daten kannst Du an Formulierungen wie "ein *X* besteht aus ...", "ein *X* ist charakterisiert durch ..." oder "ein *X* hat ..." erkennen. Manchmal lautet die Formulierung etwas anders. Die daraus resultierende Datendefinition ist ein Kommentar im Programm in folgender Form:

```
; Ein X hat / besteht aus / ist charakterisiert durch:
; - Bestandteil / Eigenschaft 1
; - Bestandteil / Eigenschaft 2
; ...
; - Bestandteil / Eigenschaft n
```

Auf die Datendefinition folgt eine entsprechende Record-Definition. Dafür überlege Dir Namen für den Record-Typ `T` und für die Felder, f_1 ... f_n. Für jedes Feld solltest Du außerdem die dazu passende Signatur sig_i angeben. Die Record-Definition hat dann folgende Form:

```
(define-record T
  make-T
  (T-f1 sig1)
```

```
...
(T-f_n sig_n))
```

Der Name des Record-Typs `T` ist die Record-Signatur, `make-T` ist der Konstruktor und $\texttt{T-f}_i$ sind die Selektoren.

Dass der Konstruktorname mit `make-` anfängt und dass die Selektornamen sich aus dem Namen des Typs und der Felder zusammensetzt, ist reine Konvention. Von ihr solltest Du aber nur aus guten Gründen abweichen.

Unter die Record-Definition gehören die Signaturen für den Konstruktor und die Selektoren:

```
(: make-T (sig_1 ... sig_n) -> T)
(: T-f_1 (T -> sig_1))
...
(: T-f_n (T -> sig_n))
```

4.14 Zusammengesetzte Daten als Eingabe: Schablone

Wenn Deine Funktion zusammengesetzte Daten als Eingabe akzeptiert (das ergibt sich aus der Signatur), gehe nach Schreiben des Gerüstes folgendermaßen vor:

- Für jede Komponente, schreibe `(sel r)` in die Schablone, wobei `sel` der Selektor der Komponente und `r` der Name des Record-Parameters ist, also zum Beispiel:

  ```
  (wallclock-time-hour wt)
  ```

- Vervollständige die Schablone, indem Du einen Ausdruck konstruieren, in dem die Selektor-Anwendungen vorkommen.
- Es ist möglich, dass nicht alle Selektor-Anwendungen im Rumpf verwendet werden: In diesem Fall lösche die Selektor-Anwendung wieder.

4.15 Zusammengesetzte Daten als Ausgabe: Schablone

Wenn Deine Funktion zusammengesetzte Daten als Ausgabe hat, schreibe einen Aufruf des passenden Record-Konstruktors in den Rumpf, zunächst mit einer Ellipse für jedes Feld des Records, also zum Beispiel:

```
(make-wallclock-time ... ...)
```

4.16 Gemischte Daten: Datenanalyse

Gemischte Daten sind Fallunterscheidungen, bei denen jeder Fall eine eigene Klasse von Daten mit eigener Signatur ist. Schreibe bei gemischten Daten eine Signaturdefinition der folgenden Form unter die Datendefinition:

```
(define sig
  (signature
    (mixed sig1 ... sign)))
```

`Sig` ist die Signatur für die neue Datensorte; $\texttt{sig}_1$ bis $\texttt{sig}_{sn}$ sind die Signaturen, aus denen die neue Datensorte zusammengemischt ist.

4.17 Gemischte Daten als Eingabe: Schablone

Eine Schablone für eine Funktion und deren Testfälle, die gemischte Daten akzeptiert, kannst Du folgendermaßen konstruieren:

- Schreibe Tests für jeden der Fälle.
- Schreibe eine `cond`-Verzweigung als Rumpf in die Schablone, die genau *n* Zweige hat - also genau soviele Zweige, wie es Fälle in der Datendefinition beziehungsweise der Signatur gibt.
- Schreibe für jeden Zweig eine Bedingung, der den entsprechenden Fall identifiziert.
- Vervollständige die Zweige, indem Du eine Datenanalyse für jeden einzelnen Fall vornimmst und entsprechende Hilfsfunktionen und Konstruktionsanleitungen benutzt. Die übersichtlichsten Programme entstehen meist, wenn für jeden Fall separate Hilfsfunktionen definiert sind.

4.18 Selbstbezüge als Eingabe: Schablone

Wenn Du eine Funktion schreibst, die Daten konsumiert, in denen Selbstbezüge enthalten sind, dann schreibe an die Stellen der Selbstbezüge jeweils einen rekursiven Aufruf.

4.19 Listen als Eingabe: Schablone

Eine Funktion, die eine Liste akzeptiert, hat folgende Schablone:

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

(define f
  (lambda (... list ...)
    (cond
      ((empty? list) ...)
      ((cons? list)
       ... (first list)
       ... (f ... (rest list) ...) ...))))
```

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

Fülle in der Schablone den `empty`-Zweig aus. Vervollständige den `cons`- Zweig unter der Annahme, dass der rekursive Aufruf `(f (rest lis))` das gewünschte Ergebnis für den Rest der Liste liefert.

Beispiel:

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

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

4.20 Natürliche Zahlen als Eingabe: Schablone

Eine Funktion, die natürliche Zahlen akzeptiert, hat folgende Schablone:

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

(define f
  (lambda (... n ...)
    (cond
      ((zero? n) ...)
      ((positive? n)
       ...
       (f ... (- n 1) ...)
       ...))))
```

Beispiel:

```
(: power (number natural -> number))

(define power
  (lambda (base exponent)
    (cond
      ((zero? exponent) 1)
      ((positive? base)
       (* base
          (power base (predecessor exponent)))))))
```

4.21 Abstraktion

Wenn Du zwei Definitionen geschrieben hast, die inhaltlich verwandt sind und viele Ähnlichkeiten aufweisen, abstrahiere wie folgt:

1. Kopiere eine der beiden Definitionen und gib ihr einen neuen Namen.
2. Ersetze die Stellen, bei denen sich die beiden Definitionen unterscheiden, jeweils durch eine neue Variable.
3. Füge die neuen Variablen als Parameter zum `lambda` der Definition hinzu oder füge ein neues `lambda` mit diesen Parametern ein. Du muss gegebenenfalls rekursive Aufrufe der Funktion anpassen.
4. Schreibe eine Signatur für die neue Funktion.
5. Ersetze die beiden alten Definitionen durch Aufrufe der neuen Definition.

Beispiel:

```
; Definition 1
(define home-points
  (lambda (game)
    (define goals1 (game-home-goals game))
    (define goals2 (game-guest-goals game))
    (cond
      ((> goals1 goals2) 3)
      ((< goals1 goals2) 0)
      ((= goals1 goals2) 1))))

; Definition 2
(define guest-points
  (lambda (game)
    (define goals1 (game-guest-goals game))
    (define goals2 (game-home-goals game))
```

```
    (cond
      ((> goals1 goals2) 3)
      ((< goals1 goals2) 0)
      ((= goals1 goals2) 1))))

; Abstraktion 1
(define compute-points
  (lambda (game)
    (define goals1 (game-guest-goals game))
    (define goals2 (game-home-goals game))
    (cond
      ((> goals1 goals2) 3)
      ((< goals1 goals2) 0)
      ((= goals1 goals2) 1))))

; Abstraktion 2
(define make-compute-points
  (lambda (get-goals-1 get-goals-2)
    (lambda (game)
      (define goals1 (get-goals-1 game))
      (define goals2 (get-goals-2 game))
      (cond
        ((> goals1 goals2) 3)
        ((< goals1 goals2) 0)
        ((= goals1 goals2) 1)))))
```

4.22 Listen als Eingabe, mit Akkumulator: Schablone

Wenn Du eine Funktion schreibst, die eine Liste akzeptiert und einen Akkumulator benutzen soll, gehe folgendermaßen vor:

1. Überlege Dir, was für Information der Akkumulator repräsentieren soll. Das ist typischerweise ein Zwischenergebnis - also ein vorläufiger Wert für das Endergebnis.
2. Konstruiere die Schablone wie folgt:

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

(define f
  (lambda (list0)
    (define accumulate
      ; Invariante
      (lambda (list acc)
        (cond
```

```
            ((empty? list) acc)
            ((cons? list)
             (accumulate (rest list) (... (first list) ... acc))))))
      (accumulate list0 ...)))
```

3. Formuliere eine möglichst konkrete Invariante zwischen `list0`, `list` und `acc` und schreibe sie als Kommentar zu `accumulate`.
4. Fülle mit Hilfe der Invariante die Ellipsen in der Funktion aus.

Beispiel:

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

(define list-sum
  (lambda (list0)
    (define accumulate
      ; sum ist die Summer aller Elemente in list0 vor list
      (lambda (list sum)
        (cond
          ((empty? list) sum)
          ((cons? list)
           (accumulate (rest list) (+ (first list) sum))))))
    (accumulate list0 0)))
```

4.23 Natürliche Zahlen als Eingabe, mit Akkumulator: Schablone

Wenn Du eine Funktion schreibst, die eine natürliche Zahl akzeptiert und einen Akkumulator benutzen soll, gehe folgendermaßen vor:

1. Überlege Dir, was für Information der Akkumulator repräsentieren soll. Das ist typischerweise ein Zwischenergebnis - also ein vorläufiger Wert für das Endergebnis.
2. Konstruiere die Schablone wie folgt:

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

(define f
  (lambda (... n0 ...)
    (define accumulate
      ; Invariante
      (lambda (n acc)
        (cond
          ((zero? n) ... acc ...)
```

```
          ((positive? n)
           (accumulate (- n 1) (... n ... acc ...))))))
    (accumulate n0 ...)))
```

3. Formuliere eine möglichst konkrete Invariante zwischen `n0`, `n` und `acc` und schreibe sie als Kommentar zu `accumulate`.
4. Fülle mit Hilfe der Invariante die Ellipsen in der Funktion aus.

Beispiel:

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

(define factorial
  (lambda (n0)
    (define accumulate
      ; acc ist das Produkt aller Zahlen von (+ n 1) bis n0
      (lambda (n acc)
        (cond
          ((zero? n) acc)
          ((positive? n)
           (accumulate (- n 1) (* n acc))))))
    (accumulate n0 1)))
```

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-lib`

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)
```
package: `deinprogramm-lib`

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

5.3 *Schreibe Dein Programm!* - fortgeschritten

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

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

Index