Kurzreferenz zu den t.Sprachen

Achim Clausing

Inhaltsverzeichnis

Die Webseite zu den t.Sprachen ist `http://cs.uni-muenster.de/tanagra`.

In dieser Referenz sind für jede der sieben t.Sprachen alle vordefinierten Befehle – Befehl im Sinn von Prozedur, also Operatoren, Funktionen und Makros – und sonstigen in den Initialisierungsfiles definierten Namen aufgelistet.

Eine Bemerkung zu `quit` und `exit`. Der Befehl `(exit)` beendet in jedem Fall das gesamte Programm. `(quit)` ist etwas weniger radikal: Für jede Eingabequelle – interaktives Terminal oder Datei – wird eine eigene Instanz der Klasse `Interpreter` erzeugt, die mit `(quit)` beendet werden kann. Damit wird aber nur diese Instanz beendet. Auf diese Weise kann man in einer Eingabedatei mit `(quit)` den darauf folgenden Text ausblenden. Beim Testen größerer Programme ist das nützlich. Nach dem `(quit)` geht es an der rufenden Stelle weiter. Oft ist dies das interaktive Terminal, es kann aber auch wieder eine Datei sein, aus der eine weitere Datei mit `(load "`*Filename*`")` aufgerufen wurde.

Die folgenden Tabellen haben jeweils drei Spalten. Die Spalte **Befehl / Name** enthält den „Aufruf-prototyp“ des Befehls, also eine Liste mit dem Namen des Befehls (rot) und den Parametern. Drei Punkte am Ende besagen, dass es sich um eine Prozedur mit variabler Argumentzahl handelt. Die Spalte **Wert / Nebenwirkung** gibt in sehr verkürzter Form an, für welches Resultat der Befehl steht bzw. für welche Nebenwirkung er ausgeführt wird. In der Spalte **Voraussetzungen** steht in ebenso verkürzter Form, welche Vorbedingungen erfüllt sein müssen, damit der Befehl erfolgreich ausgeführt werden kann.

Die Tabellen sind nur als Kurzreferenz zum schnellen Nachschauen gedacht. Detaillierte Informationen findet man im Buch zu den t.Sprachen[1].

[1] Achim Clausing: Programmiersprachen – Konzepte, Strukturen und Implementierung in Java. Spektrum Akademischer Verlag, 2011.

1 t.Zero

Befehl / Name	Wert / Nebenwirkung	Voraussetzungen
Rechenoperatoren:		
`(+ x y)`	$x + y$	`x`, `y` : Zahlen
`(- x y)`	$x - y$	`x`, `y` : Zahlen
`(* x y)`	$x * y$	`x`, `y` : Zahlen
`(/ x y)`	x/y	`x`, `y` : Zahlen
`(< x y)`	$x < y$	`x`, `y` : Zahlen
`(= x y)`	$x = y$	
`(round x n)`	x zur 0 hin gerundet auf n Dezimalstellen.	`x` : Zahl
`(num x)`	Zähler von x.	`x` : Zahl
`(den x)`	Nenner von x.	`x` : Zahl
`(if cond expr alt)`	Bedingter Ausdruck.	`cond` : Boolean
`(define function expr)`	Speichert eine Funktionsdefinition.	`function` : Nichtleere Symbol-Liste
Etwas Luxus:		
`(trace proc)`	Schaltet den Tracingmodus von `proc` ein und aus.	`proc` : Funktion, Makro, einige Operatoren.
`(protocol)`	Schaltet den Protokollmodus ein und aus.	
`(tail)`	Schaltet die Optimierung von Tail-Rekursion ein und aus.	
`(COMMENT ex1 ex2 ...)`	Ignoriert `ex1`, `ex2`,...	
Beenden:		
`(quit)`	Interpreter verlassen.	
`(exit)`	Programm beenden.	

2 t.Lisp

Befehl / Name	Wert / Nebenwirkung	Voraussetzungen
Listenoperationen:		
`(cons x ls)`	Erzeugt eine Liste mit dem Kopf `x` und dem Rumpf `ls`.	`ls` : Liste
`(car ls)`	Kopf von `ls`.	`ls` : Nichtleere Liste
`(cdr ls)`	Rumpf von `ls`.	`ls` : Nichtleere Liste
`(caar ls)`	`(car (car ls))`	`ls` : Passende Liste
`(cddr ls)`	`(cdr (cdr ls))`	`ls` : Passende Liste
`(cdar ls)`	`(cdr (car ls))`	`ls` : Passende Liste
`(cadr ls)`	`(car (cdr ls))`, 2. Element von `ls`.	`ls` : Passende Liste
`(caddr ls)`	`(car (cdr (cdr ls)))`	`ls` : Passende Liste
`(cdddr ls)`	`(cdr (cdr (cdr ls)))`	`ls` : Passende Liste
`(cadar ls)`	`(car (cdr (car ls)))`	`ls` : Passende Liste
`(caddr ls)`	`(cdr (car (cdr ls)))`, 3. Element.	`ls` : Passende Liste
`(cadddr ls)`	`(cdr (car (cdr (cdr ls))))`, 4. Element.	`ls` : Passende Liste
`nil`	Die leere Liste.	
`(nil? expr)`	Ist `expr` die leere Liste?	
`(list ex1 ex2 ...)`	Liste mit den Werten von `ex1, ex2,...`	
`(setcar ls x)`	Neuer Kopf von `ls` wird `x`.	`ls` : Nichtleere Liste
`(setcdr ls xs)`	Neuer Rumpf von `ls` wird `xs`.	`ls` : Nichtleere Liste, `xs` : Liste
Basisoperationen:		
`(bind name value)`	Erzeugt eine nicht überschreibbare Name-Wert-Bindung. Resultat `void`.	`name` : Symbol
`(quote x)`	`x` (unausgewertet).	
`(function [name] par body)`	Gibt eine Funktion mit den Parametern `par` und dem Rumpf `body` zurück. Der interne Name `name` ist optional.	`par` : Symbol-Liste `name` : Symbol
`(function [name] args body)`	Gibt eine Vararg-Funktion mit dem Rumpf `body` zurück, deren Argumentliste an den Namen `args` gebunden ist. Der interne Name `name` ist optional.	`args` : Symbol `name` : Symbol
`lambda`	Aliasname für `function`.	
`(type expr)`	Typ von `expr`.	
`(define function expr)`	Speichert eine Funktionsdefinition.	`function` : Nichtleere Symbol-Liste
`(delete name)`	Löscht die innerste Bindung von `name`.	`name` : Symbol
`(str e1 e2 ...)`	Fasst `x1, x2, ...` zu einem String zusammen.	

`(trace proc)`	Schaltet den Tracingmodus von `proc` ein und aus.	`proc` : Funktion, Makro, einige Operatoren.
Numerische Funktionen:		
`(+ x y)`	$x + y$	`x`, `y` : Zahlen
`(- x y)`	$x - y$	`x`, `y` : Zahlen
`(* x y)`	$x * y$	`x`, `y` : Zahlen
`(/ x y)`	x/y	`x`, `y` : Zahlen
`(< x y)`	$x < y$	`x`, `y` : Zahlen
`(<= x y)`	$x \leq y$	`x`, `y` : Zahlen
`(> x y)`	$x > y$	`x`, `y` : Zahlen
`(abs x)`	$\|x\|$	`x` : Zahl
`(round x n)`	x zur 0 hin gerundet auf n Dezimalstellen.	`x` : Zahl
`(num x)`	Zähler von x.	`x` : Zahl
`(den x)`	Nenner von x.	`x` : Zahl
`(-- x)`	$-x$	`x` : Zahl
`(trunc x)`	x ohne Nachkommastellen.	`x` : Zahl
Boolean:		
`(= x y)`	$x = y$	
`eq`	Aliasname für `=`.	
`(cond c1 x1 c2 x2 ...)`	Bedingter Ausdruck.	`c1, c2, ...` : Boolean
`if`	Aliasname für `cond`.	
`else`	Aliasname für `true` in bedingten Ausdrücken.	
`true`	Boolesche Konstante „wahr“.	
`false`	Boolesche Konstante „falsch“.	
`(and x y)`	$x \wedge y$ (boolesches „Und“).	`x`, `y` : Boolean
`(or x y)`	$x \vee y$ (boolesches „Oder“).	`x`, `y` : Boolean
`(not x)`	Negation von `x`.	`x` : Boolean
`(number? expr)`	Ist `expr` eine Zahl?	
`(boolean? expr)`	Ist `expr` ein Boolean?	
`(symbol? expr)`	Ist `expr` ein Symbol?	
`(list? expr)`	Ist `expr` eine Liste?	
Nützliches und Triviales:		
`(COMMENT ex1 ex2 ...)`	Ignoriert `ex1`, `ex2`,...	
`(square x)`	x^2	`x` : Zahl
`void`	Der leere Ausdruck.	
Beenden:		
`(quit)`	Interpreter verlassen.	
`(exit)`	Programm beenden.	

3 t.Pascal

Befehl / Name	Wert / Nebenwirkung	Voraussetzungen
Basisoperationen:		
`(cond c1 x1 c2 x2 ...)`	Bedingter Ausdruck.	`c1, c2, ...` : Boolean
`if`	Aliasname für `cond`.	
`(str e1 e2 ...)`	Fasst `x1, x2, ...` zu einem String zusammen.	
`(print e1 e2 ...)`	Schreibt `x1, x2, ...` als String nach `stdout`.	
`(println e1 e2 ...)`	Schreibt `x1, x2, ...` mit einem Zeilenvorschub nach `stdout`.	
`(type expr)`	Typ von `expr`.	
`(quote x)`	`x` (unausgewertet).	
`(catch name expr alt)`	Fängt eine Exception mit den Namen `name` auf, die während der Auswertung von `expr` geworfen wird. In diesem Fall wird der Wert von `alt` zurückgegeben.	`name` : Symbol
`(throw name expr)`	Wirft eine Exception mit dem Namen `name`, die den Wert von `expr` enthält.	`name` : Symbol
`(throw expr)`	Wirft eine Exception mit dem Namen `ERROR`, die den Wert von `expr` enthält.	
`(block ex1 ex2 ...)`	Wertet `ex1, ex2,...` in einer anfangs leeren lokalen Umgebung aus.	
Operationen mit Seiteneffekten:		
`(set name value)`	Erzeugt eine überschreibbare Name-Wert-Bindung. Das Resultat ist der Wert von `value`.	`name` : Symbol
`(var name value)`	Definiert eine lokale Variable `name` im innersten Bindingsrahmen der Auswertungsumgebung. Das Resultat ist der Wert von `value`.	`name` : Symbol
`(define function expr)`	Speichert eine Funktionsdefinition.	`function` : Nichtleere Symbol-Liste
`program`	Aliasname für `define`.	
`(defmacro macro expr)`	Speichert eine Makrodefinition.	`macro` : Nichtleere Symbol-Liste
`(delete name)`	Löscht eine evtl. bestehende Name-Wert-Bindung. Das Resultat ist `void`.	`name` : Symbol
`(loop expr)`	Wertet `expr` in einer Endlosschleife immer wieder aus.	
`(while cond expr)`	While-Schleife.	`cond` : Boolean.
`(for i lo hi expr)`	For-Schleife, Schrittweite 1. `i` muss nicht deklariert werden.	`i` : Symbol, `lo, hi` : Zahlen.
Blockstruktur:		
`(begin ex1 ex2 ...[end])`	Block, in dem `return`, `break` und lokale Namen (mit `var` definiert) erlaubt sind. Das „`end`“ ist optional.	
`(return expr)`	Verlässt einen `begin`-Block und gibt den Wert von `expr` zurück.	

`(break)`	Verlässt einen `begin`-Block.	
Records und Arrays:		
`(record name length)`	Erzeugt einen Record der Länge `length` mit dem Typnamen `name`.	`name` : Symbol, `length` : Zahl
`(list ex1 ex2 ...)`	Liste mit den Werten von `ex1, ex2,...`	
`(array ls)`	Gibt einen Array mit den Elementen der Liste `ls` zurück.	`ls` : Liste.
`(array n)`	Gibt einen leeren Array der Länge `n` zurück.	`n` : Natürliche Zahl.
`(length a)`	Länge des Arrays `a`.	`a` : Array.
`(rset r i expr)`	Setzt Komponente `i` des Record `r` den Wert von `expr` zu.	`r` : Record, `i` : Gültiger Index.
`(rget r i)`	Gibt Komponente `i` des Record `r` zurück.	`r` : Record, `i` : Gültiger Index.
Nützliches:		
`void`	Der leere Ausdruck	
`end`	Aliasname für `void`.	
`(load filename)`	Lädt eine Datei.	`name` : Symbol
`(read)`	Wartet auf eine Eingabe und gibt diese als String zurück.	
`(trace proc)`	Schaltet den Tracingmodus von `proc` ein und aus.	`proc` : Funktion, Makro, einige Operatoren.
`(part name expr)`	Gibt Bestandteil `name` der Java-Implementierung von `expr` zurück. In der Java-Klasse von `expr` muss ein Attribut mit diesem Namen und dem Java-Typ `Expr` vorhanden sein.	`name` : Symbol
`(line)`	Schaltet die Anzeige von Zeilennummern im Prompt aus und an.	
`(parse s)`	Parst den String `s`.	`s` : String
Strings:		
`(sget s i)`	`i`-tes Zeichen des Strings `s` (Indexanfang 0).	`s` : String
`(sget s i j)`	Teilstring des Strings `s` von Index `i` bis Index `j`.	`s` : String
`(slength s)`	Länge des Strings `s`.	`s` : String
Rechnen:		
`(= x y)`	$x = y$	
`(+ x y)`	$x + y$	`x`, `y` : Zahlen
`(- x y)`	$x - y$	`x`, `y` : Zahlen
`(* x y)`	$x * y$	`x`, `y` : Zahlen
`(/ x y)`	x/y	`x`, `y` : Zahlen
`(< x y)`	$x < y$	`x`, `y` : Zahlen
`(> x y)`	$x > y$	`x`, `y` : Zahlen
`(<= x y)`	$x \leq y$	`x`, `y` : Zahlen
`(>= x y)`	$x \geq y$	`x`, `y` : Zahlen

`(round x n)`	x zur 0 hin gerundet auf n Dezimalstellen.	`x` : Zahl
`(square x)`	x^2	`x` : Zahl
`(trunc x)`	x ohne Nachkommastellen.	`x` : Zahl
`(floor x)`	$\lfloor x \rfloor$	`x` : Zahl
`(mod m n)`	m modulo n	`m, n` : Zahl
`(max x y)`	$\max(x, y)$	`x, y` : Zahl
`(min x y)`	$\min(x, y)$	`x, y` : Zahl
`(abs x)`	$\lvert x \rvert$	`x` : Zahl
`(even? n)`	Ist n gerade?	`n` : Zahl
`(power x n)`	x^n	`x` : Zahl, `n` : Natürliche Zahl
Boolean:		
`true`	Boolesche Konstante „wahr“.	
`false`	Boolesche Konstante „falsch“.	
`(and x y)`	$x \wedge y$ (boolesches „Und“).	`x, y` : Boolean
`(or x y)`	$x \vee y$ (boolesches „Oder“).	`x, y` : Boolean
`(not x)`	Negation von `x`.	`x` : Boolean
`(number? expr)`	Ist `expr` eine Zahl?	
`(array? expr)`	Ist `expr` ein Array?	
`(record? expr)`	Ist `expr` ein Record?	
Kommentar und Beenden:		
`(COMMENT ex1 ex2 ...)`	Ignoriert `ex1`, `ex2`,...	
`(quit)`	Interpreter verlassen.	
`(exit)`	Programm beenden.	

4 t.Scheme

Befehl / Name	Wert / Nebenwirkung	Voraussetzungen
Basisoperatoren:		
`(op name)`	Gibt den Operator mit dem internen Namen `name` zurück.	`name` : Symbol.
`(bind name value)`	Erzeugt eine nicht überschreibbare Name-Wert-Bindung. Resultat `void`.	`name` : Symbol
`(set name value)`	Erzeugt eine überschreibbare Name-Wert-Bindung. Das Resultat ist der Wert von `value`.	`name` : Symbol
`(let s1 ex1 s2 ex2 ... expr)`	Erzeugt einen lokalen Kontext, mit den Bindungen `s1`→`ex1`, `s2`→`ex2`, ... Resultat ist der Wert von `expr`.	`s1, s2 ...` : Symbole
`(seq ex1 ex2 ...)`	Wertet `ex1, ex2,...` nacheinander aus.	
`&`	Aliasname für `seq`.	
`(print e1 e2 ...)`	Schreibt `x1, x2, ...` als String nach `stdout`.	
`(println e1 e2 ...)`	Schreibt `x1, x2, ...` mit einem Zeilenvorschub nach `stdout`.	
`(type expr)`	Typ von `expr`.	
Ablaufkontrolle:		
`(cond c1 x1 c2 x2 ...)`	Bedingter Ausdruck.	`c1, c2, ...` : Boolean
`(catch name expr alt)`	Fängt eine Exception mit den Namen `name` auf, die während der Auswertung von `expr` geworfen wird. In diesem Fall wird der Wert von `alt` zurückgegeben.	`name` : Symbol
`(throw name expr)`	Wirft eine Exception mit dem Namen `name`, die den Wert von `expr` enthält.	`name` : Symbol
`(throw expr)`	Wirft eine Exception mit dem Namen `ERROR`, die den Wert von `expr` enthält.	
`(eval expr)`	Wertet den *Wert* von `expr` aus.	
`(quote x)`	`x` (unausgewertet).	
Funktionen und Makros:		
`(function [name] par body)`	Gibt eine Funktion mit den Parametern `par` und dem Rumpf `body` zurück. Der interne Name `name` ist optional.	`par` : Symbol-Liste `name` : Symbol
`(function [name] args body)`	Gibt eine Vararg-Funktion mit dem Rumpf `body` zurück, deren Argumentliste an den Namen `args` gebunden ist. Der interne Name `name` ist optional.	`args` : Symbol `name` : Symbol
`lambda`	Aliasname für `function`.	
`(macro [name] par body)`	Gibt ein Makro mit den Parametern `par` und dem Rumpf `body` zurück. Der interne Name `name` ist optional.	`par` : Symbol-Liste `name` : Symbol

`(macro [name] args body)`	Gibt ein Vararg-Makro mit dem Rumpf `body` zurück, dessen Argumentliste an den Namen `args` gebunden ist. Der interne Name `name` ist optional.	`args` : Symbol `name` : Symbol
`(define function expr)`	Speichert eine Funktionsdefinition.	`function` : Nichtleere Symbol-Liste
`(defmacro macro expr)`	Speichert eine Makrodefinition.	`macro` : Nichtleere Symbol-Liste
`(context f)`	Kontext von `f`.	`f` : Funktion.
`(global-context)`	Der globale Kontext.	
`(actual-context)`	Der aktuelle Kontext.	
`(symbols env)`	Liste von Listen aller in den Frames von `env` gebundenen Symbole.	`env` : Kontext.

Gleichheit:

`(= x y)`	$x = y$	
`eq`	Aliasname für `=`.	
`(# x y)`	$x \neq y$	
`!=`	Aliasname für `#`.	

Rechnen:

`(+ x y)`	$x + y$	`x`, `y` : Zahlen
`(- x y)`	$x - y$	`x`, `y` : Zahlen
`(* x y)`	$x * y$	`x`, `y` : Zahlen
`(/ x y)`	x/y	`x`, `y` : Zahlen
`(< x y)`	$x < y$	`x`, `y` : Zahlen
`(<= x y)`	$x \leq y$	`x`, `y` : Zahlen
`(> x y)`	$x > y$	`x`, `y` : Zahlen
`(abs x)`	$\lvert x \rvert$	`x` : Zahl
`(round x n)`	x zur 0 hin gerundet auf n Dezimalstellen.	`x` : Zahl
`(num x)`	Zähler von x.	`x` : Zahl
`(den x)`	Nenner von x.	`x` : Zahl
`(-- x)`	$-x$	`x` : Zahl
`(// x y)`	Ganzzahliger Anteil von x/y	`x`, `y` : Zahlen
`div`	Aliasname für `//`.	
`(trunc x)`	x ohne Nachkommastellen.	`x` : Zahl
`(floor x)`	$\lfloor x \rfloor$	`x` : Zahl
`(mod m n)`	m modulo n	`m, n` : Zahlen
`(isqrt x)`	Ganzzahlige Quadratwurzel von `x`	`x` : Zahl ≥ 0.
`(power x n)`	x^n	`x` : Zahl, `n` : Natürliche Zahl
`(random)`	Zufällige ganze Zahl in $[0, \ldots, 2^{32} - 1]$.	
`(random_01)`	Zufällige Zahl in $[0, \ldots, 1)$.	

Elemente von t.Pascal:		
`(block ex1 ex2 ...)`	Wertet `ex1, ex2,...` in einer anfangs leeren lokalen Umgebung aus.	
`(record name length)`	Erzeugt einen Record der Länge `length` mit dem Typnamen `name`.	`name` : Symbol, `length` : Zahl
`(var name value)`	Definiert eine lokale Variable `name` im innersten Bindungsrahmen der Auswertungsumgebung. Das Resultat ist der Wert von `value`.	`name` : Symbol
`end`	Aliasname für `void`.	
`(loop expr)`	Wertet `expr` in einer Endlosschleife immer wieder aus.	
`(while cond expr)`	While-Schleife.	`cond` : Boolean.
`(for i lo hi expr)`	For-Schleife, Schrittweite 1. `i` muss nicht deklariert werden.	`i` : Symbol, `lo, hi` : Zahlen.
`(array ls)`	Gibt einen Array mit den Elementen der Liste `ls` zurück.	`ls` : Liste.
`(array n)`	Gibt einen leeren Array der Länge `n` zurück.	`n` : Natürliche Zahl.

Elemente von t.Lisp:		
`(list ex1 ex2 ...)`	Liste mit den Werten von `ex1, ex2,...`	
`(cons x ls)`	Erzeugt eine Liste mit dem Kopf `x` und dem Rumpf `ls`.	`ls` : Liste
`nil`	Die leere Liste.	
`(car ls)`	Kopf von `ls`.	`ls` : Nichtleere Liste
`(cdr ls)`	Rumpf von `ls`.	`ls` : Nichtleere Liste
`(caar ls)`	`(car (car ls))`	`ls` : Passende Liste
`(cadr ls)`	`(car (cdr ls))`, 2. Element von `ls`.	`ls` : Passende Liste
`(cddr ls)`	`(cdr (cdr ls))`	`ls` : Passende Liste
`(cadar ls)`	`(car (cdr (car ls)))`	`ls` : Passende Liste
`(caddr ls)`	`(car (cdr (cdr ls)))`	`ls` : Passende Liste
`(cdddr ls)`	`(cdr (cdr (cdr ls)))`	`ls` : Passende Liste
`(setcar ls x)`	Neuer Kopf von `ls` wird `x`.	`ls` : Nichtleere Liste
`(setcdr ls xs)`	Neuer Rumpf von `ls` wird `xs`.	`ls` : Nichtleere Liste, `xs` : Liste
`(map f ls)`	Wendet `f` auf alle Elemente von `ls` an.	`f` : Prozedur, `ls` : Liste.
`(bimap f l1 l2)`	Wendet `f` paarweise auf die Elemente von `l1` und `l2` an.	`f` : Prozedur, `l1, l2` : Listen.
`(join l1 l2)`	Vereinigt die Listen `l1` und `l2`.	`l1, l2` : Listen.
`(reverse ls)`	Liste `ls` in umgekehrter Reihenfolge.	`ls` : Liste.
`(rotate-left ls)`	Rotiert Liste `ls` um eine Position nach links.	`ls` : Liste.
`(element i ls)`	`i`-tes Element der Liste `ls`. (Ab Index 0.)	`ls` : Liste, `i` : gültiger Index.
`(element? x ls)`	Ist `x` Element der Liste `ls`?.	`ls` : Liste

`(take n ls)`	Die ersten `n` Elemente der Liste `ls`.)	`ls` : Liste, `n` : nat. Zahl.
`(unique ls)`	Liste `ls` ohne Duplikate.	`ls` : Liste.
`(flatten ls)`	Liste `ls` flach machen.	`ls` : Liste von Listen.
`(range i j)`	Liste `(i i+1 ...)` (Elemente $\leq$ `j`).	`i, j` : Zahlen.
`(column ls)`	Spaltendarstellung von `ls`.	`ls` : Liste.

Strings:

`(str e1 e2 ...)`	Fasst `x1, x2, ...` zu einem String zusammen.	
`(sget s i)`	`i`-tes Zeichen des Strings `s` (Indexanfang 0).	`s` : String
`(sget s i j)`	Teilstring des Strings `s` von Index `i` bis Index `j`.	`s` : String

Nützliches:

`void`	Der leere Ausdruck.	
`(length x)`	Länge von `x`.	`x` : Liste, Array oder String.
`(clock)`	Systemzeit in ms.	
`(time expr)`	Dauer (in ms) der Auswertung von `expr`.	
`(part name expr)`	Gibt Bestandteil `name` der Java-Implementierung von `expr` zurück. In der Java-Klasse von `expr` muss ein Attribut mit diesem Namen und dem Java-Typ `Expr` vorhanden sein.	`name` : Symbol
`(load filename)`	Lädt eine Datei.	`name` : Symbol
`(read)`	Wartet auf eine Eingabe und gibt diese als String zurück.	
`(parse s)`	Parst den String `s`.	`s` : String
`(trace proc)`	Schaltet den Tracingmodus von `proc` ein und aus.	`proc` : Funktion, Makro, einige Operatoren.
`(tail)`	Schaltet die Optimierung von Tail-Rekursion ein und aus.	
`(line)`	Schaltet die Anzeige von Zeilennummern im Prompt aus und an.	
`(exec cmd)`	Führt das Kommando `cmd` aus.	`cmd` : String
`(protocol)`	Schaltet Protokollmodus an und aus.	
`(operators)`	Liste aller verfügbaren Operatoren.	
`(square x)`	x^2	`x` : Zahl
`(cls)`	Löscht den Bildschirminhalt (Nur Unix).	
`(clear)`	Löscht den Bildschirminhalt (Nur Unix, Implementierung ohne `exec`).	

Ströme:

`(scons x s)`	Erzeugt einen Strom mit dem Kopf `x` und dem Rumpf `s`.	`s` : Strom
`(scar s)`	Gibt den Kopf des Stroms `s` zurück.	`s` : Strom

`(scdr s)`	Gibt den Rumpf des Stroms `s` zurück.	`s` : Strom
`Cons`	Aliasname für `scons`.	
`Car`	Aliasname für `scar`.	
`Cdr`	Aliasname für `scdr`.	
Boolean:		
`true`	Boolesche Konstante „wahr“.	
`false`	Boolesche Konstante „falsch“.	
`(and x y)`	$x \wedge y$ (boolesches „Und“, mit Kurzschluss-Semantik).	`x`, `y` : Boolean
`(or x y)`	$x \vee y$ (boolesches „Oder“, mit Kurzschluss-Semantik).	`x`, `y` : Boolean
`(not x)`	Negation von `x`.	`x` : Boolean
`(number? expr)`	Ist `expr` eine Zahl?	
`(boolean? expr)`	Ist `expr` ein Boolean?	
`(symbol? expr)`	Ist `expr` ein Symbol?	
`(list? expr)`	Ist `expr` eine Liste?	
`(string? expr)`	Ist `expr` ein String?	
`(array? expr)`	Ist `expr` ein Array?	
`(function? expr)`	Ist `expr` eine Funktion?	
`(macro? expr)`	Ist `expr` ein Makro?	
`(integer? x)`	Ist `x` eine ganze Zahl?	`x` : Zahl
`(even? x)`	Ist x gerade?	`x` : Zahl
`(nil? expr)`	Ist `expr` die leere Liste?	
`(void? expr)`	Ist `expr` der leere Ausdruck?	
`(vararg? f)`	Ist `f` ein Makro oder eine Funktion mit variabler Argumentzahl?	`f` : Makro oder Funktion.
`(bound? sym)`	Ist das Symbol `sym` im aktuellen Kontext gebunden?	`sym`: Symbol
`(symlist? x)`	Ist `x` eine Symbol-Liste?	
Info-Funktion:		
`(? expr)`	Kurzinformation zu `expr`.	
`(formals f)`	Formale Parameterliste von `f`.	`f` : Makro oder Funktion.
`(name expr)`	Interner Name von `expr` (falls vorhanden).	
Kommentar und Beenden:		
`(COMMENT ex1 ex2 ...)`	Ignoriert `ex1`, `ex2`,....	
`(quit)`	Interpreter verlassen.	
`(exit)`	Programm beenden.	

5 t.Lambda

Befehl / Name	Wert / Nebenwirkung	Voraussetzungen
Operatoren:		
`(bind name value)`	Erzeugt eine nicht überschreibbare Name-Wert-Bindung. Resultat `void`.	`name` : Symbol
`(lambda [name] par body)`	Gibt eine Funktion mit den Parametern `par` und dem Rumpf `body` zurück. Der interne Name `name` ist optional.	`par` : Symbol-Liste `name` : Symbol
`(macro [name] par body)`	Gibt ein Makro mit den Parametern `par` und dem Rumpf `body` zurück. Der interne Name `name` ist optional.	`par` : Symbol-Liste `name` : Symbol
Der Rest gehört nicht zu t.Lambda und könnte auch fehlen:		
`(trace proc)`	Schaltet den Tracingmodus von `proc` ein und aus.	`proc` : Funktion, Makro, einige Operatoren.
`(protocol)`	Schaltet Protokollmodus an und aus.	
`(eval expr)`	Wertet den *Wert* von `expr` aus.	
`(quote x)`	`x` (unausgewertet).	
`(context f)`	Kontext von `f`.	`f` : Funktion.
`(relambda f)`	Wandelt Function oder Macro `f` in einen Lambda-Ausdruck um.	`f` : Funktion oder Makro.
`(value sym f)`	Wert von `sym` im Kontext von `f`.	`sym` : Symbol, `f` : Funktion.
Beenden:		
`(quit)`	Interpreter verlassen.	
`(exit)`	Programm beenden.	

6 t.Java

t.Java kennt alle Funktionen von t.Scheme und zusätzlich die folgenden:

Befehl / Name	Wert / Nebenwirkung	Voraussetzungen
Klassen, Objekte und Methoden:		
`(class A symlist)`	Erzeugt eine Klasse mit dem internen Namen `A` und den Attributen, die in `symlist` angegeben sind.	`A` : Symbol, `symlist` : Symbol-Liste.
`(class A)`	Kurzform für `(class A '())`.	`A` : Symbol.
`(class B A symlist)`	Erzeugt eine Unterklasse mit dem internen Namen `B` der Klasse `A`; mit den zusätzlichen Attributen `symlist`.	`B` : Symbol, `A` : Klasse, `symlist` : Symbol-Liste.
`(class B A)`	Kurzform für `(class B A '())`.	`B` : Symbol, `A` : Klasse.
`(defclass A ...)`	Kurzform für `bind A (class A ...)`.	`A` : Klasse, `...` : alles, was in `(class A ...)` für „...“ stehen darf.
`(method A formals body)`	Erweitert die Klasse `A` um eine Methode mit den formalen Parametern `formals` und dem Rumpf `body`. Der erste Name in `formals` ist der Methodenname.	`A` : Klasse, `formals` : Symbol-Liste.
`(new A x1 x2 ...)`	Erzeugt ein Objekt der Klasse `A` und ruft `(A init x1 x2 ...)` auf.	`A` : Klasse, Existenz eines passenden Konstruktors `init`.
`Class`	Gemeinsamer Typ aller t.Java-Klassen.	
`(seal A)`	Versiegelt die Klasse `A` (Verbot weiterer Methodendeklarationen).	`A` : Klasse.
Nützliches:		
`(object? expr)`	Ist `expr` ein Objekt?	
`(class? expr)`	Ist `expr` eine Klasse?	
`(superclass class)`	Die Oberklasse von `class`, oder `class` im Fall von Klassen ohne Oberklasse.	`class` : Klasse.
`(superclasses class)`	Liste aller Oberklasse von `class`.	`class` : Klasse.
`(rootclass? class)`	Ist `class` eine Klasse ohne Oberklasse?	`class` : Klasse.
`(fieldlist class)`	Die Liste der Attributnamen von `class` (mit Attributen der Oberklassen).	`class` : Klasse.
`(instanceof? expr class)`	Ist `expr` eine Instanz von `class`?	`class` : Klasse.
`(methods class)`	Liste aller Methoden von `class` (mit Methoden der Oberklassen).	`class` : Klasse.

7 t.Prolog

t.Prolog kennt alle Funktionen von t.Scheme außer `record` und `array`, sowie zusätzlich die folgenden:

Befehl / Name	Wert / Nebenwirkung	Voraussetzungen
Basiselemente von t.Prolog:		
`(term symlist expr)`	Wandelt `expr` in einen Term um, indem es die Namen aus `symlist` durch gleichnamige Unbestimmte ersetzt.	`symlist` : Symbol-Liste
`(unify t1 t2)`	Unifiziert `t1` und `t2`. Gibt die allgemeinste unifizierende Substitution zurück oder das Symbol `fail`.	
`(substitute sub expr)`	Wendet Substitution `sub` auf `expr` an.	`sub` : Substitution.
`(product sub1 sub2)`	Gibt das Produkt der Substitutionen `sub1` und `sub2` zurück.	`sub1, sub2` : Substitutionen.
`(torule ls)`	Erzeugt aus der Liste `ls = (f ...)` die Regel `f :- ...`	`ls` : Nichtleere Liste.
`(rulebase name)`	Erzeugt eine leere Regelbasis mit dem internen Namen `name`.	`name` : Symbol.
`(addrule rb rule)`	Erweitert die Regelbasis `rb` um die Regel `rule`.	`rb` : Regelbasis, `rule` : Regel.
`(resolve rb symlist ex1 ex2 ...)`	Wandelt `ex1 ex2 ...` mit `symlist` in Terme um und startet die Resolution dieser Terme bzgl. der Regelbasis `rb`. Gibt bei Erfolg `true` zurück, sonst `false`.	`rb` : Regelbasis, `symlist` : Symbol-Liste.
Nützliches:		
`(indet x)`	Gibt die Unbestimmte `x` zurück.	`x` : Symbol.
`(indeterminate? expr)`	Ist `expr` eine Unbestimmte?	
`(lhs rule)`	Linke Seite von `rule`	`rule` : Regel.
`(rhs rule)`	Rechte Seite von `rule`	`rule` : Regel.
`(content rb)`	Liste der Regeln in `rb`.	`rb` : Regelbasis.
`(rules rb)`	Die Regeln in `rb`, spaltenweise formatiert.	`rb` : Regelbasis.
`(getrule i rb)`	Die `i`-te Regel in `rb`.	`rb` : Regelbasis, `i` : Index.
`(reset rb)`	Löscht alle Regeln in `rb`.	`rb` : Regelbasis.
Die Standard-Regelbasis `Prolog`:		
`Prolog`	Die vordefinierte Regelbasis.	
`(rule [indets] lhs :- rhs)`	Speichert Regel mit den Unbestimmten `indets`, mit linker Seite `lhs` und rechter Seite `rhs` in `Prolog`. `indets` ist optional.	`indets` : Symbol-Liste.
`(fact [indets] expr)`	Speichert Regel mit den Unbestimmten `indets`, mit linker Seite `expr` und leerer rechter Seite in `Prolog`. `indets` ist optional.	`indets` : Symbol-Liste.
`(ask expr)`	Ist `expr` mit den Regeln von `Prolog` beweisbar?	.

`(ask indets : expr)`	Ist `(term indets expr)` mit den Regeln von `Prolog` beweisbar?	`indets` : Symbol-Liste.
`(define-print-rule)`	Speichert die Regel `print` in `Prolog`.	
`(define-continue-rule)`	Speichert die Regel `continue` in `Prolog`.	
`(define-basic-rules)`	Speichert die Regeln `print`, `continue`, `=`, `#`, `not` und `!` in `Prolog`.	
`(restart)`	Löscht `Prolog`, setzt `recursionLimit` und `loopThreshold` auf Vorgabewerte und ruft `define-print-rule` auf.	
`(deleterule i)`	Löscht die `i`-te Regel in `Prolog`.	`i` : Index.

8 Alphabetische Liste aller Operatoren

Nachstehend sind in alphabetischer Reihenfolge alle in den t.Sprachen verfügbaren Operatoren aufgelistet. In Rot ist jeweils der *interne* Name angegeben. Extern können die Operatoren auch andere Namen haben. So ist zum Beispiel der Operator `equal` in der Regel an den Namen „=“ gebunden oder der t.Prolog-Operator `rule` an den externen Namen `torule`. Erst bei der Auswertung des externen Operatornamens sieht man den internen Namen:

```
-> =
 op[equal]
```

Der (unveränderliche) interne Name ergibt sich aus dem Klassennamen des Operators: Die Klasse `Operator_`*name* definiert den Operator `op[`*name*`]`. Mit `(op` *name*`)` kann der Operator geladen werden (falls der Operator op nicht gelöscht wurde). Weil nicht jede Operatorklasse in einer eigenen `.java`-Datei steht, ist in der letzten Spalte angegeben, wo die Klasse `Operator_`*name* zu finden ist.

Die genaue Semantik der einzelnen Operatoren wird ausführlich im Buch zu den t.Sprachen erläutert. In Zweifelsfällen lohnt es sich, den Quellcode zu konsultieren.

Will man etwa wissen, wie der Operator `var` funktioniert, so sieht man aus der Tabelle, dass der Operator in `PascalOp.java` definiert ist und findet dort den Code:

```
 0  class Operator_var extends Operator {
 1      public Expr apply(List args, Env env) throws Alarm {
 2          checkArity(args, 2);
 3          Symbol sym = checkSymbol(args.first());
 4          Frame frame = env.first();
 5          frame.checkProtected(sym);
 6          Expr e = args.second().eval(env);
 7          frame.bind(sym, e);
 8          return e;
 9      }
10  }
```

Damit hat man genaue Auskunft über den Operator: Er erwartet zwei Argumente (Zeile 2), das erste muss ein Symbol sein, es wird nicht ausgewertet (Zeile 3). Dieses Symbol darf im innersten Bindungsrahmen der Auswertungsumgebung nicht in einer geschützten Bindung stehen (Zeilen 4 und 5). Ist die Bedingung erfüllt, so wird der Wert des zweiten Arguments in diesem Bindungsrahmen an das Symbol gebunden (Zeilen 6 und 7); er ist zugleich das Resultat des Aufrufs `(var sym expr)` (Zeile 8).

Für alle anderen Operatoren gilt Entsprechendes: Die exakte Semantik entnehme man notfalls dem Quellcode. Die Semantik aller übrigen Funktionen, Makros etc. ergibt sich aus ihrer Definition in den Initialisierungsdateien zu den t.Sprachen im Ordner `init`.

Operator	Wert / Nebenwirkung	Voraussetzungen	Datei
(addrule rb rule)	Erweitert die Regelbasis rb um die Regel rule.	rb : Regelbasis, rule : Regel.	PrologOp.java
(bind name value)	Erzeugt eine nicht überschreibbare Name-Wert-Bindung.	name : Symbol	BasicOp.java
(block ex1 ex2 ...)	Wertet ex1, ex2,... in einer anfangs leeren lokalen Umgebung aus.		PascalOp.java
(car ls)	Kopf von ls.	ls : Nichtleere Liste	ListOp.java
(catch name expr alt)	Fängt eine Exception mit den Namen name auf, die während der Auswertung von expr geworfen wird. In diesem Fall wird der Wert von alt zurückgegeben.	name : Symbol	ControlOp.java
(cdr ls)	Rumpf von ls.	ls : Nichtleere Liste	ListOp.java
(class A [B] [sl])	Erzeugt eine neue Klasse mit dem internen Namen A; ggfs. mit der Oberklasse B und/oder den Attributen sl.	A : Symbol, B : Klasse, sl : Symbol-Liste.	ListOp.java
(clock)	Systemzeit in ms.		SpecialOp.java
(cond c1 x1 c2 x2 ...)	Bedingter Ausdruck.	c1, c2, ... : Boolean	ControlOp.java
(cons x ls)	Erzeugt eine Liste mit dem Kopf x und dem Rumpf ls.	ls : Liste	ListOp.java
(define function expr)	Speichert eine Funktionsdefinition.	function : Nichtleere Symbol-Liste	FunctionOp.java
(defmacro macro expr)	Speichert eine Makrodefinition.	macro : Nichtleere Symbol-Liste	FunctionOp.java
(delete name)	Löscht eine evtl. bestehende Name-Wert-Bindung. Das Resultat ist void.	name : Symbol	BasicOp.java
(den x)	Nenner von x.	x : Zahl	NumberOp.java
(div x y)	x/y	x, y : Zahlen	NumberOp.java
(equal x y)	$x = y$		BasicOp.java
(eval expr)	Wertet den *Wert* von expr aus.		ControlOp.java
(exec cmd)	Führt das Kommando cmd aus.	cmd : String	SpecialOp.java
(exit)	Programm beenden.		ControlOp.java
(function [nm] par body)	Gibt eine Funktion mit den Parametern par und dem Rumpf body zurück.	par : Symbol-Liste oder Symbol, nm : Symbol.	FunctionOp.java
(length ls)	Länge von ls.	ls : Liste.	ListOp.java
(less x y)	$x < y$	x, y : Zahlen	NumberOp.java
(let s1 ex1 ... expr)	Erzeugt einen lokalen Kontext, mit den Bindungen s1→ex1, ... Resultat ist der Wert von expr.	s1, ... : Symbole	BasicOp.java
(line)	Schaltet die Anzeige von Zeilennummern im Prompt aus und an.		SpecialOp.java
(load filename)	Lädt eine Datei.	name : Symbol	SpecialOp.java
(loop expr)	Wertet expr in einer Endlosschleife immer wieder aus.		ControlOp.java

(macro [name] par body)	Gibt ein Makro mit den Parametern par und dem Rumpf body zurück.	par : Symbol-Liste oder Symbol.	FunctionOp.java
(method A formals body)	Erweitert die Klasse A um eine Methode mit den formalen Parametern formals und dem Rumpf body.	A : Klasse, formals : Symbol-Liste.	ObjectOp.java
(minus x y)	$x - y$	x, y : Zahlen	NumberOp.java
(mult x y)	$x * y$	x, y : Zahlen	NumberOp.java
(new A x1 x2 ...)	Erzeugt ein Objekt der Klasse A und ruft (A init x1 x2 ...) auf.	A : Klasse, Existenz eines passenden Konstruktors init.	ObjectOp.java
(num x)	Zähler von x.	x : Zahl	NumberOp.java
(op name)	Gibt den Operator mit dem internen Namen name zurück.	name : Symbol.	BasicOp.java
(parse s)	Parst den String s.	s : String	SpecialOp.java
(part name expr)	Gibt Bestandteil name der Java-Implementierung von expr zurück. In der Java-Klasse von expr muss ein Attribut mit diesem Namen und dem Java-Typ Expr vorhanden sein.	name : Symbol	SpecialOp.java
(plus x y)	$x + y$	x, y : Zahlen	NumberOp.java
(print e1 e2 ...)	Schreibt x1, x2, ... als String nach stdout.		BasicOp.java
(product sub1 sub2)	Gibt das Produkt der Substitutionen sub1 und sub2 zurück.	sub1, sub2 : Substitutionen.	PrologOp.java
(protocol)	Schaltet Protokollmodus an und aus.		SpecialOp.java
(quote x)	x (unausgewertet).		ListOp.java
(read)	Wartet auf eine Eingabe und gibt diese als String zurück.		SpecialOp.java
(record name length)	Erzeugt einen Record der Länge length mit dem Typnamen name.	name : Symbol, length : Zahl	PascalOp.java
(resolve rb sl ex1 ...)	Wandelt ex1 ... mit sl in Terme um und startet die Resolution dieser Terme bzgl. der Regelbasis rb. Gibt bei Erfolg true zurück, sonst false.	rb : Regelbasis, sl : Symbol-Liste.	PrologOp.java
(round x n)	x zur 0 hin gerundet auf n Dezimalstellen.	x : Zahl	NumberOp.java
(rule ls)	Erzeugt aus der Liste ls = (f ...) die Regel f :- ...	ls : Nichtleere Liste, rb : Regelbasis, rule : Regel.	PrologOp.java
(rulebase name)	Erzeugt eine leere Regelbasis mit dem internen Namen name.	name : Symbol.	PrologOp.java
(scar s)	Gibt den Kopf des Stroms s zurück.	s : Strom	StreamOp.java
(scdr s)	Gibt den Rumpf des Stroms s zurück.	s : Strom	StreamOp.java
(scons x s)	Erzeugt einen Strom mit dem Kopf x und dem Rumpf s.	s : Strom	StreamOp.java
(seal A)	Versiegelt die Klasse A.	A : Klasse.	ObjectOp.java
(seq ex1 ex2 ...)	Wertet ex1, ex2,... nacheinander aus.		BasicOp.java
(set name value)	Erzeugt eine überschreibbare Name-Wert-Bindung. Das Resultat ist der Wert von value.	name : Symbol	BasicOp.java

(setcar ls x)	Neuer Kopf von ls wird x.	ls : Nichtleere Liste	ListOp.java
(setcdr ls xs)	Neuer Rumpf von ls wird xs.	ls : Nichtleere Liste, xs : Liste	ListOp.java
(slength s)	Länge des Strings s.	s : String	StringOp.java
(sget s i [j])	Teilstring des Strings s von Index i bis Index j. Default für j ist i.	s : String	StringOp.java
(str e1 e2 ...)	Fasst x1, x2, ... zu einem String zusammen.		StringOp.java
(subst sub expr)	Wendet Substitution sub auf expr an.	sub : Substitution.	PrologOp.java
(tail)	Schaltet die Optimierung von Tail-Rekursion ein und aus.		SpecialOp.java
(term symlist expr)	Wandelt expr in einen Term um, indem es die Namen aus symlist durch gleichnamige Unbestimmte ersetzt.	symlist : Symbol-Liste	PrologOp.java
(throw [name] expr)	Wirft eine Exception mit dem Namen name, die den Wert von expr enthält. Der Default für name ist ERROR.	name : Symbol	ControlOp.java
(trace proc)	Schaltet den Tracingmodus von proc ein und aus.	proc : Funktion, Makro, einige Operatoren.	SpecialOp.java
(type expr)	Typ von expr.		BasicOp.java
(unify t1 t2)	Unifiziert t1 und t2. Gibt die allgemeinste unifizierende Substitution zurück oder das Symbol fail.		PrologOp.java
(var name value)	Definiert eine lokale Variable name im innersten Bindingsrahmen der Auswertungsumgebung. Das Resultat ist der Wert von value.	name : Symbol	PascalOp.java