

---

# Einleitung

---

*Coffee or tea? There's a growing body of research to suggest that both are probably good for you.* (<http://www.time.com/time/askdrweil>)

## Kaffee oder Tee?

Kaffee – das Wort in der Überschrift steht für die Programmiersprache Java<sup>1</sup>. Der Tee, die bekömmliche Alternative zum Kaffee, das sind die *t.Sprachen*, von denen dieses Buch handelt. Es sind sieben kleine Programmiersprachen, alle in Java implementiert: *t.Zero*, *t.Lisp*, *t.Pascal*, *t.Scheme*, *t.Lambda*, *t.Java* und *t.Prolog*.

Das „t.“ in ihren Namen wird zwar wie das englische Wort Tea gesprochen, steht aber für *tiny*, *toy*, *teaching* und *training*. Jede t.Sprache ist eine Miniaturausgabe ihres Originals, eine Art Lehr- und Lernspielzeug – syntaktisch und semantisch stark vereinfacht, aber doch nur so weit, dass die charakteristische Art, in der jeweiligen Sprache zu denken und zu programmieren, erhalten bleibt.

Zu jeder t.Sprache findet man in diesem Buch eine vollständige Implementierung, die so durchschaubar ist (das „t.“ steht auch für *transparent*), dass man im Detail nachvollziehen kann, wie die wesentlichen Elemente der Sprache funktionieren – nicht nur, wie sie benutzt werden, sondern auch, wie ihr Innenleben konstruiert ist.

Wie das einleitende Zitat schon andeutet, sind die t.Sprachen keineswegs als Alternative zu Java gedacht. Niemand sollte auf die Idee kommen, sie als Ersatz für vorhandene Programmiersprachen im Normalbetrieb zu verwenden. Sie sollen auch keine Werbung dafür machen, künftig doch hin und wieder statt in Java in Scheme oder Prolog zu programmieren.

Man könnte die Intention des Buchs mit dem alten konfuzianischen Slogan „der Weg ist das Ziel“ ausdrücken: Es geht darum, typische Strukturen und interne Mechanismen von

---

<sup>1</sup> Jede aus einem Java-Programm erzeugte class-Datei beginnt in Hexadezimaldarstellung mit der Magic Number CAFE – genauer gesagt, mit den vier Bytes CA FE BA BE. Die Magic Number einer Datei codiert das Dateiformat.

Programmiersprachen zu studieren. Das Endprodukt, die jeweilige Sprache, darf man danach getrost wieder beiseitelegen („weg ist das Ziel“ ; -).

Dieses Buch verfolgt mehrere Absichten, die aber alle letztlich eine innere Einheit bilden. Zu den Anliegen gehört es,

- die wichtigsten Programmierstile und -paradigmen exemplarisch zu behandeln,
- allen, die Interesse an Programmiersprachen haben, Stoff zum Nachdenken und Material zum eigenen Experimentieren zu geben,
- zu zeigen, dass Programmiersprachen keine „schwarzen Kästen“ mit unveränderlichen (und meistens wenig hinterfragten) Regeln sind, sondern dass es gar nicht so schwer ist, ihr Innenleben zu verstehen,
- exemplarisch die Realisierung eines Java-Programms zu dokumentieren, das umfangreicher ist als die Beispiele, mit denen man in einer Einführung in Java normalerweise zu tun hat. Objektorientierte Programmierung kann ihre Vorteile erst im Umfeld größerer Softwareprojekte ausspielen. Viele Aspekte von Java sind allein anhand kleiner einführender Programme nur schwer zu verstehen.

## Die t.Sprachen

Programmiersprachen sind Sprachen und als solche Medien des Denkens (mehr dazu unter <http://de.wikipedia.org/wiki/Sprache>). Unser Denken im Zusammenhang mit Computern wird ganz wesentlich geprägt von den Programmiersprachen, die wir beherrschen.

Deshalb wäre es für jeden, der mit Informatik zu tun hat, eine gute Sache, nach Möglichkeit mindestens ein halbes Dutzend Programmiersprachen zu lernen. „So viele Sprachen einer beherrscht, so viele Leben lebt er“, sagt ein Sprichwort.

Die Zeit zum Lernen so vieler Sprachen hat nicht jeder. Mit sehr viel weniger Zeitaufwand ist es aber möglich, sich einen Überblick über zumindest ein paar der wichtigsten Grundkonzepte zu verschaffen, die der manchmal an den Turmbau von Babel erinnernden Vielfalt von Programmiersprachen zugrunde liegen.

Die t.Sprachen sind sich untereinander äußerlich sehr ähnlich, jede weitere fühlt sich erst einmal so an wie die vorangehende. Trotzdem verkörpert jede einen ganz spezifischen Aspekt des Programmierens.

### t.Zero

t.Zero ist eine extrem einfache, rein *deklarative* Sprache. Ihre Besonderheit liegt in dem, was sie *nicht* hat: Es gibt keine Befehle oder Anweisungen, keine Konstrukte zur Beschreibung von Abläufen und keine Variablen zur Speicherung von Zwischenresultaten.

Ein t.Zero-Programm ist eine Funktion, die mit sparsamsten Mitteln ein Resultat beschreibt. Nur das Ergebnis, das *Was* des Programms, ist wichtig, nicht das *Wie* des Rechenverlaufs.

Die Sprache kennt nur Zahlen und boolesche Werte. Für den Umgang mit ihnen gibt es außer Rechen- und Vergleichsoperationen lediglich bedingte Ausdrücke und die Möglichkeit zur Definition von Funktionen. Das ganze sonstige Instrumentarium der klassischen Programmierung fehlt. Dieser Mangel führt dazu, dass man *gezwungen* ist, sich an das deskriptive Postulat zu halten.

Computer heißen so, weil sie ursprünglich als reine Rechenmaschinen konzipiert wurden. Für solche Zwecke ist t.Zero sehr gut geeignet, die Pioniere des Computerzeitalters wären mit den Möglichkeiten der Sprache sicherlich zufrieden gewesen.

Trotzdem fehlt t.Zero etwas Entscheidendes. Und es sind nicht die Befehle, die Variablen, die While- oder die For-Schleifen, die fehlen. Es ist etwas Anderes. Die Sprache bietet keine Möglichkeit, aus mehreren Daten *ein* Datenelement zu machen. Dieses Defizit wird in t.Lisp und t.Pascal auf unterschiedliche Weise behoben, durch Listen beziehungsweise durch Arrays und Records. Es wird sich zeigen, dass dieser scheinbar kleine Unterschied geradezu dramatische Konsequenzen für den Charakter der Sprachen hat.

### t.Lisp

Die Sprache t.Lisp ist, wer hätte das vermutet, eine Version der klassischen Programmiersprache Lisp. Sie beruht auf dem Konzept der Liste. Mit Listen kann man beliebig viele Daten zu einem Datenelement zusammenfassen, also gewissermaßen aus vielen Bäumen einen Wald machen.

Für die Zusammenfassung, die *Aggregation*, von Daten gibt es viele Möglichkeiten, darunter Listen, Arrays, Tupel oder Objekte. Listen sind unter diesen sicherlich die eleganteste Variante, sie sind perfekt an die Prinzipien der Rekursion und der deklarativen Programmierung angepasst.

Das Konzept der Liste ist unter anderem deshalb so mächtig, weil es in Kooperation mit Symbolen als einem eigenen Datentyp die Grenze zwischen Daten und Programm nahezu vollständig aufhebt. Man kann in Lisp mit verblüffender Leichtigkeit andere Programmiersprachen implementieren – oder auch die Sprache Lisp selbst.

Wie t.Zero ist auch t.Lisp eine deklarative Sprache. Trotzdem liegen Welten zwischen beiden: Durch den Typ Liste als Mechanismus zur Aggregation erschließt t.Lisp eine Fülle von neuen Ausdrucks- und Abstraktionsmöglichkeiten.

### t.Pascal

In der nächsten Sprache, t.Pascal, ist der Typ Liste durch Records und Arrays ersetzt. Damit ist t.Pascal ein typischer Vertreter der Klasse der *imperativen* Sprachen.

Arrays haben auf den ersten Blick große Ähnlichkeit mit Listen. Man könnte deshalb vermuten, dass sich der Charakter einer Programmiersprache nicht allzu sehr ändert, wenn Listen durch Arrays ersetzt werden.

Das Gegenteil ist jedoch richtig. Listen und Arrays sind sehr eigenwillige Kreaturen, die jeweils ihren ganz eigenen Programmierstil geradezu erzwingen. Arrays sind *zustandsveränderliche* Datenelemente, sie erhalten ihre Werte durch *Zuweisung*, man kann sie nur *iterativ* durchlaufen. Diese Begriffe – Zustand, Zuweisung und Iteration – sind charakteristisch für die klassische, maschinennahe, anweisungsorientierte (imperative) Art des Programmierens, für die Sprachen wie Fortran, Algol, C und Pascal typisch waren.

### t.Scheme

In t.Scheme lernen wir eine *funktionale* Sprache kennen. Diese Sprachen sind dadurch charakterisiert, dass sie Funktionen als einen eigenständigen Datentyp kennen: Funktionen können

als Argumente an andere Funktionen übergeben werden oder Resultate von Berechnungen sein. Sie sind „First Class“-Daten, gleichberechtigt mit allen anderen Typen.

Scheme war ursprünglich ein Lisp-Abkömmling. Auch in Lisp gab es funktionale Elemente, trotzdem sind Funktionen in Scheme etwas anderes. Das liegt daran, dass jede Scheme-Funktion einen eigenen *Kontext* besitzt.

Man kann sich einen Funktionskontext als einen lokalen Speicher vorstellen, in dem die Funktion „private“ Daten aufbewahren kann. Bei richtiger Betrachtung sind Funktionen mit eigenem Kontext fast schon Objekte. Tatsächlich ist die objektorientierte Programmierung eine Weiterentwicklung der Idee der Funktion mit eigenem Kontext.

## **t.Lambda**

Im Kapitel über t.Lambda geht es um eine Sicht auf Funktionen, die sich von der des vorangehenden Kapitels ganz erheblich unterscheidet.

Mit t.Lambda machen wir einen Ausflug in die Theoretische Informatik. Ziel ist es nachzuweisen, dass man prinzipiell die gesamte Programmierung ausschließlich mit dem Datentyp Funktion bestreiten kann, und zwar mit einem Funktionsbegriff im Sinne der Mathematik, ohne Vorschriften über die Auswertung von Argumenten, ohne lokalen Kontext – *rein funktional*.

Ein solcher Nachweis hat keine unmittelbar praktische Bedeutung. Die Sprache t.Lambda hat als „reales“ Vorbild den Lambda-Kalkül, einen Formalismus, den der Logiker Alonzo Church in den 1930er Jahren entwickelt hat und der auch heute noch, vor allem bei der Untersuchung von Typsystemen, von Interesse ist.

## **t.Java**

t.Java ist unser Repräsentant der Familie der *objektorientierten* Sprachen. Anders als Java ist aber t.Java keine *rein* objektorientierte Programmiersprache. Man kann in t.Java-Programmen imperative, funktionale und objektorientierte Sprachelemente mischen. t.Java ist als Erweiterung von t.Scheme konzipiert, es behebt unter anderem den Mangel, dass t.Scheme kein Mittel zur Definition eigener neuer Typen besitzt.

Als Anwendung von t.Java werden unter anderem *Ströme* definiert. Ströme sind unendliche Folgen und auf den ersten Blick meint man, dass ein „unendlich großes“ Datenelement nicht als Ganzes in einem Rechner, der ja immer nur endlich viel Speicherplatz hat, existieren könne. Die Erklärung gibt das Prinzip der *lazy evaluation*, des Aufschiebens von Berechnungen. Man kann darin ein eigenes Programmierparadigma sehen. Viele neuere Programmiersprachen nutzen dieses Prinzip, zum Beispiel Haskell und Python.

## **t.Prolog**

Schließlich wird im letzten Kapitel mit t.Prolog noch das Paradigma der *logischen* Programmierung behandelt.

Im Laufe der Geschichte hat es zahlreiche Versuche gegeben, logisches Denken zu automatisieren. Schon Gottfried Wilhelm Leibniz (1646–1716) hat einen Kalkül zur Automatisierung des Denkens entworfen, den *Calculus Ratiocinator*, der in der Lage sein sollte, wahre Aussagen zu finden und zu beweisen. Man kann ihn als frühe Utopie einer logischen Programmiersprache ansehen.

Programme zum Beweisen mathematischer Aussagen gab es schon in der Frühzeit des Computers. Aus diesen Anfängen entwickelte sich die Programmiersprache Prolog, die auf einer eingeschränkten Version der *Prädikatenlogik* beruht.

Ein Prolog-Programm ist im Wesentlichen eine *Regelbasis*, das heißt eine Menge von Aussagen und Regeln zur Ableitung neuer Aussagen. An die Regelbasis kann man Fragen stellen und Prolog entscheidet dann, ob es die als Frage gestellte Aussage aus den in der Menge enthaltenen Aussagen folgern kann. Es ist verblüffend zu sehen, mit welcher Leichtigkeit Prolog selbst schwierige Rätsel löst. Der Ausdruck „Künstliche Intelligenz“ ist in diesem Zusammenhang durchaus berechtigt. Wenn man andererseits die Implementierung von t.Prolog ansieht, kann man sich wundern, mit wie wenig Aufwand diese Art von „Intelligenz“ realisiert werden kann. Trotz der geringen Menge an Quellcode ist t.Prolog ein komplettes Prolog-System mit allem, was man von einer logischen Programmiersprache erwartet. Für Spezialisten: Auch Pseudoprädikate, der Cut-Mechanismus und eine Schleifenerkennung sind enthalten und selbstverständlich – wie in allen t.Sprachen – eine ordentliche Fehlerbehandlung.

In jeder der t.Sprachen gibt es Möglichkeiten zum Abfangen und zur Verarbeitung von Fehlern, zur nachträglichen genaueren Untersuchung von Fehlerursachen und einen Tracing-Modus zur Beobachtung von Programmabläufen.

## Die Java-Implementierung

Dieses Buch wendet sich in erster Linie an Leserinnen und Leser, die zwar schon Erfahrungen mit Java gemacht haben, aber noch keine „gestandenen Softwareentwickler“ sind. Nach einem einführenden Java-Kurs möchte man als relativer Programmierneuling vielleicht

- ganz generell mehr über Programmierung und Programmiersprachen wissen. Was für Konzepte und Ideen gibt es, die man in Java nicht findet?
- einmal ein größeres Java-Programm kennenlernen – nach Möglichkeit aber eines mit gut lesbarem Quellcode und ausführlichen Erläuterungen.
- besser verstehen, wie Programmiersprachen funktionieren. Was verbirgt sich alles „unter der Motorhaube“ einer Programmiersprache?

Besonders der letzte Punkt ist interessant. Wir alle sind heute daran gewöhnt, technischen Geräten zu vertrauen – wir behandeln sie als Black Box, mit deren Interna wir nichts zu tun haben und haben wollen. So, wie man Auto fahren kann, ohne zu wissen, wie ein Motor funktioniert, kann man programmieren, ohne etwas von Compilern oder Interpretern zu verstehen. Trotzdem ist ein Minimum an technischem Wissen beim Umgang mit Programmiersprachen genauso nützlich wie beim Autofahren.

Die Java-Implementierung der t.Sprachen ist deshalb ein zentraler Bestandteil dieses Buchs. Alle t.Sprachen basieren automobiltechnisch gesprochen auf derselben Plattform, einem *Interpreter*, der jeweils auf unterschiedliche, sprachspezifische Weise initialisiert wird. Das erlaubt die Realisierung der einzelnen t.Sprachen auf einer einheitlichen Grundlage unter maximaler Wiederverwendung von Java-Quellcode.

Der Interpreter selber stellt aber lediglich eine einfache Syntax und einen grundlegenden Auswertungsmechanismus zur Verfügung. Die Semantik der einzelnen t.Sprachen wird mit Hilfe spezieller Datentypen und Operatoren erzeugt.

Für die Implementierung der t.Sprachen ergibt sich daraus in natürlicher Weise eine Aufteilung des Quellcodes auf zwei Pakete. Das Paket *tanagra* enthält den Interpreter, das Paket *expressions* die Klassen für die einzelnen Datentypen und Operatoren<sup>2</sup>.

Bei der Java-Implementierung wurde großer Wert auf die Lesbarkeit des Java-Quellcodes gelegt. Die meisten .java-Dateien sind sehr kurz und übersichtlich. Selbst die umfangreichste Datei (*RuleBase.java*, zu t.Prolog gehörend) enthält ohne Kommentare weniger als zweihundert Zeilen Code. Die meisten anderen Dateien sind noch deutlich kleiner.

In den einzelnen Kapiteln wird die Implementierung jeweils im letzten Abschnitt erläutert; im Inhaltsverzeichnis sind diese Teile durch ein vorangestelltes Kaffeetassen-Symbol gekennzeichnet. Die Kaffeetassen-Sektionen sind bewusst informell gehalten: Sie bilden keine „Dokumentation“, wie man sie zum Beispiel mit javadoc erzeugen würde, sondern sie versuchen die Leitgedanken bei der Implementierung der jeweiligen Sprache zu erläutern.

Wer in einem ersten Durchgang die t.Sprachen nur aus Benutzersicht kennenlernen will, der kann die Kaffeetassen-Teile des Buchs problemlos überspringen. Wem diese Abschnitte wiederum zu informell gehalten sind, der sollte den vollständigen Quellcode konsultieren. Er ist, wie gesagt, so lesbar wie möglich geschrieben; stellenweise enthält er auch weitere Erläuterungen.

In der Softwareentwicklung wird seit langer Zeit unterschieden zwischen der *Programmierung im Kleinen*, bei der es mehr um algorithmische Details geht, und der *Programmierung im Großen*, die mit dem Entwurf von Softwaresystemen befasst ist, bei denen viele Module zusammenspielen ([9]). Alle Beispiele zu den t.Sprachen dieses Buchs gehören naturgemäß in das Gebiet der Programmierung im Kleinen.

Die Implementierung der t.Sprachen liegt dagegen an der Grenze zur Programmierung im Großen (auch wenn „richtig große“ Softwareprojekte ganz andere Dimensionen haben). Wer sie aufmerksam studiert, wird erkennen, dass es bei größeren Programmen auf Dinge ankommt, die nicht so sehr von der verwendeten Programmiersprache, sondern vor allem vom systematischen Vorgehen bei der Entwicklung abhängen. Einzelne Klassen sollen möglichst *kohärent* sein, sie sollen ein Konzept realisieren, während die Klassen untereinander möglichst *entkoppelt* sein sollen – je weniger Abhängigkeiten, desto besser. Solche Aspekte des objektorientierten Programmierens lernt man nicht an kleinen Beispielen, sondern nur beim Entwickeln größerer Programme.

Auch wenn also die in diesem Buch angesprochenen programmiersprachlichen Konzepte alle zur Programmierung im Kleinen zählen und Techniken der systematischen Entwicklung großer Softwaresysteme nicht explizit behandelt werden, kann und sollte man doch als Leser die Implementierung der t.Sprachen unter dem Aspekt des Software Engineering betrachten, als Beispiel für die Realisierung eines größeren, mit dem Ziel der einfachen Erweiterbarkeit geschriebenen Java-Programms.

## Ockhams Rasiermesser

Der mittelalterliche Logiker und Theologe Wilhelm von Ockham (1285–1347) wäre vermutlich entsetzt darüber, dass man ihm das *Prinzip vom Rasiermesser* (Occam's Razor)

---

<sup>2</sup> Das Tanagra-Theater war ein optisches Spielzeug, das in den 1920er Jahren von der Firma Zeiss gebaut wurde. Über ein Spiegelsystem verkleinerte es eine reale Bühne auf das Format eines Fernsehbildes. Ein Tanagra-Theater war die Miniaturversion eines Theaters und zugleich ein echtes Theater, so wie die t.Sprachen Miniaturversionen von Programmiersprachen und zugleich echte Programmiersprachen sind.

in die Schuhe schiebt: „Die einfachste Lösung ist die beste, alles Überflüssige sollte man wegrasieren“. Ob nun authentisch oder nicht, Ockhams Rasiermesser beschreibt eine Grundeinstellung, die man sich als Softwareentwickler möglichst früh zu eigen machen sollte und die auch eine Leitlinie dieses Buchs ist. Sie stimmt übrigens im Wesentlichen mit dem „KISS-Prinzip“ überein (*Keep it simple, stupid!*), von dem die Wikipedia behauptet, es entstamme dem Bereich der Informatik (vgl. [de.wikipedia.org/wiki/Unix-Philosophie](http://de.wikipedia.org/wiki/Unix-Philosophie) und [de.wikipedia.org/wiki/KISS-Prinzip](http://de.wikipedia.org/wiki/KISS-Prinzip)).

Diesem Prinzip folgt das Buch in mehrfacher Hinsicht:

- Die gemeinsame Syntax und der gemeinsame Interpreter sorgen dafür, dass die Implementierung der t.Sprachen klein ist.
- Es wurde sehr darauf geachtet, Optimierungen, die den Programmcode komplizierter gemacht hätten, schon im Planungsstadium wegzurasieren. Die Verständlichkeit des Java-Codes der Implementierung hatte bei allen Entwurfsentscheidungen Vorrang vor der Geschwindigkeit der in den t.Sprachen geschriebenen Programme.
- Das, was in den t.Sprachen programmiert wird, ist nicht immer einfach. Aber es wurde versucht, die Darstellung so einfach wie möglich zu halten.

Aus Gründen der Kürze und Einfachheit wurde fast völlig auf eine Behandlung der theoretischen Grundlagen der verschiedenen Programmierparadigmen verzichtet. Dem Buch liegt eine „Hands on“-Philosophie zugrunde, bei der das Ausprobieren, das Experimentieren und die konkrete Struktur im Vordergrund stehen, nicht eine Theorie der Programmiersprachen. Die t.Sprachen sind Do-it-yourself-Bausätze, keine normierten Programmiersprachen. Sie sollen dazu einladen, sie zu erweitern, zu verändern, auf ihrer Basis neue Ideen im Kleinen zu testen.

Der Java-Quellcode ist so einfach und lesbar gehalten, dass man für jedes Beispielprogramm in einer der t.Sprachen ohne Weiteres bis auf die Java-Ebene hinunter mitverfolgen kann, was im Einzelnen passiert.

Dahinter steht die Erweiterung des KISS-Prinzips zum Prinzip KISMET: *Keep it simple, make everything transparent*. Transparenz bedeutet hier, dass man die Semantik der t.Sprachen immer bis auf die Ebene von Java zurückverfolgen kann. Notfalls, wenn man den Ablauf eines Programms gar nicht versteht, kann man einen Debugger zu Hilfe nehmen, den es für praktisch alle Java-Entwicklungsumgebungen oder auch als eigenständiges Programm gibt. Erfahrungsgemäß reichen aber die in die t.Sprachen integrierten Tracing- und Protokollierungsmöglichkeiten völlig aus.

## Leserkreis

Das Buch ist in erster Linie für Studierende der Informatik und verwandter Studiengänge gedacht, die Grundkenntnisse in Java besitzen und mehr über Programmierung und Programmiersprachen lernen möchten. Es will Stoff für alle bringen, die nach einer ersten Einführung in Java neugierig darauf sind, wie es weitergehen könnte mit der Programmierung.

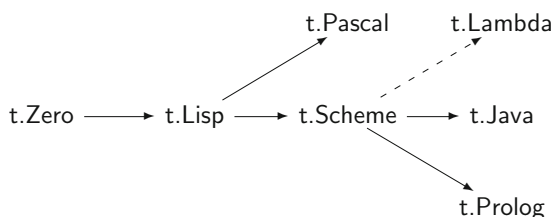
Es kann als Grundlage für ein Seminar oder eine Vorlesung etwa ab dem dritten Semester benutzt werden. Die Vorgehensweise ist, wie schon im vorangehenden Abschnitt gesagt wurde, pragmatisch-praktisch. Durch den weitgehenden Verzicht auf die Erörterung theoretischer Grundlagen (sogar beim Lambda-Kalkül und in der logischen Programmierung) sollte es unmittelbar im Anschluss an einen ersten Java-Kurs verwendbar sein.

Es gibt eine Menge guter Java-Bücher für Programmieranfänger. Was man seltener findet, ist Literatur, die einem nach den ersten Schritten weiterhilft, die Anregungen gibt, wie ein größeres Programmierprojekt realisiert werden kann, die aber nicht gleich auf das Ziel der professionellen Softwareentwicklung ausgerichtet ist, sondern eher darauf, wie man nach dem Erlernen der technischen Details von Java ein Verständnis für das Wesen von Programmierung bekommt.

Wer sich also selber irgendwo in dem weiten Bereich zwischen Programmieranfänger und -profi einordnet und zugleich neugierig darauf ist, mehr über Programmierung zu erfahren, der könnte aus diesem Buch Nutzen ziehen.

## Hinweise zur Benutzung

Die einzelnen t.Sprachen sind nicht völlig unabhängig voneinander. Das folgende Diagramm zeigt die Beziehungen:



Wer sich beispielsweise für t.Scheme interessiert, sollte vorher zumindest überschlägig die Kapitel zu t.Zero und t.Lisp angesehen haben. t.Java und t.Prolog sind wechselseitig unabhängige Erweiterungen von t.Scheme, während t.Lambda eine viel kleinere Sprache als t.Scheme ist, die aber trotzdem eine vorherige Beschäftigung mit t.Scheme voraussetzt.

Unter Berücksichtigung dieser Abhängigkeiten kann man einzelne t.Sprachen herausgreifen und die entsprechenden Kapitel wahlweise mit oder ohne Bezugnahme auf die Java-Implementierung lesen.

Die Implementierung der Sprachen ist jeweils am Ende des entsprechenden Kapitels dargestellt. Man kann diese „Kaffeetassen-Abschnitte“ auch zurückstellen und am Ende zusammengefasst studieren.

Das Buch ist nicht dafür gedacht, es von der vorderen zur hinteren Umschlagseite durchzulesen. Die vielen Beispiele sollen dazu anregen, sie auszuprobieren, zu variieren und ähnliche Programme selber zu schreiben. Mit der Zeit lernt man dann, die t.Sprachen zu verändern und schließlich auch selber neue Sprachen zu konstruieren. Das geht ziemlich leicht durch Abändern der Initialisierungsdateien, aber auch Eingriffe in den Quellcode des Tanagra-Interpreters sind gut machbar. Den ganzen Java-Code und die Beispiele sollte man als eine große Bastelkiste betrachten und keinesfalls als Fertigprodukte.

## Webseite zum Buch

Die t.Sprachen können von der Webseite zu diesem Buch heruntergeladen werden:

<http://cs.uni-muenster.de/tanagra>

Nähere Hinweise zur Installation findet man in Anhang A.



Programmiersprachen – Konzepte, Strukturen und  
Implementierung in Java

Clausing, A.

2011, XII, 450 S. 40 Abb., Softcover

ISBN: 978-3-8274-2850-9