Schreibe Dein Programm!
(Lehrerhandbuch)

Michael Sperber

20. September 2005

Kapitel 1

Erfolgreicher Programmierunterricht

Gute Programmierer schreiben ihre Programme nach festen Regeln: Sie erkennen Standardprobleme und wenden auf diese immer dieselben Techniken an. Damit reduzieren sie den mentalen Aufwand, den sie für die Lösung eines Problems benötigen und haben den Kopf frei für die wirklich schwierigen Aspekte eines Programmierprojekts.

Von Programmieranfängern wird heutzutage meist erwartet, daß sie sich das Wissen über Standardprobleme und deren Lösungen selbst aneignen. Stattdessen konzentriert sich die traditionelle Programmierausbildung darauf, die Konstrukte der einen oder anderen Programmiersprache zu erläutern und hofft darauf, daß die Schülerinnen und Schüler schon selbst die gedankliche Kluft zwischen diesen Konstrukten und den tatsächlichen Problemen finden, mit denen sie dann konfrontiert werden.

Dementsprechend wenig erfolgreich ist die traditionelle Programmierausbildung: nur besonders motivierte und begabte Schüler dürfen hoffen, auf diese Weise tatsächlich das Programmieren zu erlernen. Der eigentliche didaktische Effekt findet dabei nicht im Unterricht statt — die Schülerinnen und Schüler müssen ihn sich in oft frustrierender Heimarbeit selbst erkämpfen. Über die Qualität der erworbenen Fähigkeiten beschwert sich die IT-Industrie zu recht: „Trial and Error“ spielt meist eine weitaus wichtigere Rolle als die systematische Programmkonstruktion.

DeinProgramm verfolgt einen radikal anderen Ansatz der Programmierausbildung. Dieser Ansatz basiert auf dem Prinzip der *systematischen Programmkonstruktion* und richtet sich an *alle* Schülerinnen und Schüler ab der fünften Klasse.

Im Mittelpunkt von DeinProgramm steht dabei ein Satz von *Konstruktionsanleitungen* für Programme. Diese zeigen stets einen systematischen und vor allem nachvollziehbaren Weg von einer Aufgabenstellung zu deren Lösung. Die Konstruktionsanleitungen sind die wichtigste Grundlage für eine erfolgreiche Programmierausbildung — sie haben sich im Schulunterricht, der universitären und der beruflichen Programmierausbildung bereits vielfach bewährt. Alle anderen Komponenten von DeinProgramm richten sich nach ihnen aus.

Manchen professionellen Programmierern, Ausbildern und Lehrern mögen die Konstruktionsanleitungen und die aus ihnen resultierenden Programme — gerade am Anfang — oft unnötig pedantisch oder umständlich erscheinen. Dementsprechend entsteht bei den Lehrenden häufig der Impuls, die möglichen Verbesserungen im Unterricht sofort anzubringen, oder den vermeintlich pedantischen Lösungsweg gleich zu überspringen. Der didaktische Sinn der Konstruktionsanleitungen ist aber gerade, *vollständig systematische und nachvollziehbare Wege* der Problemlösung

aufzuzeigen. Nur so können viele Schülerinnen und Schüler den Unterricht in eigenständige Problemlösungen und Erfolgserlebnisse übersetzen. Um eine klischeehafte aber passende Metapher zu bemühen: Die ersten Gehversuche mögen damit etwas staksig und unbeholfen aussehen — das Laufen kommt später.

Einige Worte zur Wahl der Programmiersprache: Es versteht sich von selbst, daß die in der Anfängerausbildung erworbenen Fähigkeiten auf eine Vielfalt von späteren Aufgaben, Umfelde und damit Programmiersprachen übertragbar sein sollten. Schließlich ist es wahrscheinlich, daß die Programmiersprache, die angehende Programmierer bei ihrer ersten Aufgabe verwenden müssen nicht diejenige ist, die sie zuerst erlernt haben. Andererseits ist es sinnvoll, Anfänger zunächst an einer konkreten Programmiersprache üben zu lassen, um Erfolgs- und damit Lernerfolge zu ermöglichen.

In der traditionellen Programmierausbildung hat sich eingebürgert, die verwendete Programmiersprache ausschließlich danach auszuwählen, was gerade in der industriellen Software--Entwicklung „hip“ erscheint, und damit von Arbeitgebern und IT-Dienstleistern gefordert wird. Daran, daß angehende Programmierer diese Programmiersprachen als Teil ihrer Ausbildung erlernen, ist nichts auszusetzen. Leider sind diese Sprachen — über die vergangenen Jahre hießen sie meist Pascal, C, C++, und Java — sowie die für sie verfügbaren Programmierumgebungen nicht für Anfänger, sondern für professionelle, bereits fertig ausgebildete Programmierer konstruiert worden, und damit für die *Anfänger*ausbildung weitgehend ungeeignet.

Viele für Anfänger geeignete Programmiersprachen und -umgebungen sind vorstellbar. Gerade in Kombination gibt es jedoch traurigerweise nur wenige. DeinProgramm verwendet eine Kombination von Sprache und Programmierumgebung, die sich in der Praxis als besonders erfolgreich in der Anfängerausbildung erwiesen haben: Scheme und DrScheme. Dabei geht es nicht um einen „Scheme-Kurs“. Vielmehr dient Scheme als Ausdrucksmittel für die vermittelten Konzepte.[1] Damit funktioniert DeinProgramm auch nicht als Kurs, an dessen Ende Schüler „professionelle“ Scheme-Programme schreiben können. Vielmehr ist DeinProgramm das nötige Sprungbrett, auf das konkrete Kurse „Programmieren in X“ folgen können (und sollten).

[1]Auch dies unterscheidet DeinProgramm von der traditionellen Programmierausbildung, die in der Regeln nach den Konstrukten einer konkreten Programmiersprache organisiert wird.

Kapitel 2

Die Entwicklungsumgebung DrScheme

Eine der wichtigsten Bestandteile von DEINPROGRAMM ist die Entwicklungsumgebung DrScheme. DrScheme wurde speziell für die Bedürfnisse von Programmieranfängern entwickelt. Deshalb unterscheidet sich DrScheme wesentlich von Umgebungen für andere Programmiersprachen und auch anderen Umgebungen für Scheme selbst, die sich stets an erfahrene Programmierer richten.

2.1 Definitions- und Interaktionsfenster

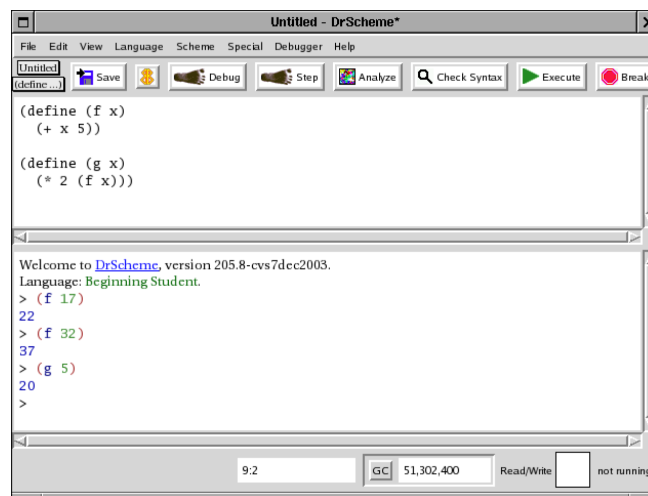


Abbildung 2.1: Das DrScheme-Hauptfenster

Abbildung 2.1 zeigt das DrScheme-Hauptfenster.[1] Das Hauptfenster enthält zwei Unterfenster: das obere ist das *Definitionsfenster*, das untere das sogenannte *Interaktionsfenster*.

Das Definitionsfenster ist ein Editor für Scheme-Programme. Das Interaktionsfenster erlaubt es Benutzern, die Funktionen des geschriebenen Programm auszuprobieren. Wenn der Benutzer den `Run`-Knopf auf der Werkzeugleiste gedrückt hat,

[1] Je nach Betriebssystem kann es leicht unterschiedlich aussehen — inbesondere, was die Anordnung und Position der Menüleiste betrifft.

werden die Definitionen und Ausdrücke des Programms (wenn es keine syntaktischen Fehler enthält) ausgewertet und im Interaktionsfenster verfügbar gemacht.

Das Interaktionsfenster erlaubt dem Benutzer dann, Ausdrücke einzugeben, deren Ergebnisse nach dem Druck auf die Return-Taste ausgedruckt werden. Abbildung 2.1 zeigt hierfür einige Beispiele.[2]

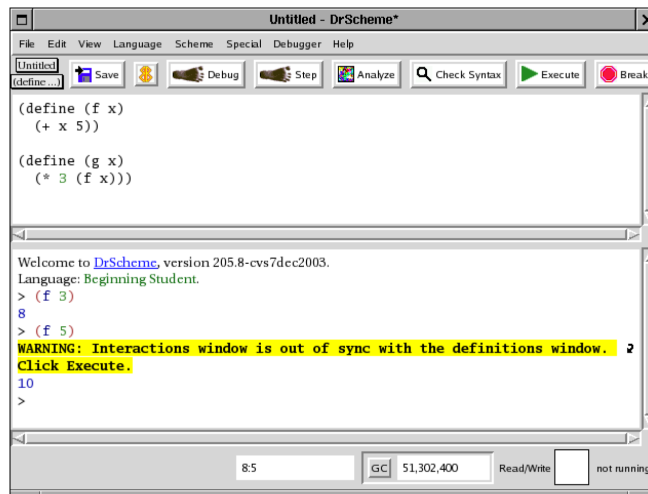


Abbildung 2.2: Die Definitionen wurden geändert, aber noch nicht übernommen.

Wenn der Benutzer etwas im Definitionsfenster ändert und dann das Interaktionsfenster benutzt, ohne die Definitionen durch Druck auf `Run` zu übertragen, zeigt DrScheme eine Warnung an (siehe Abbildung 2.2, da dann im Interaktionsfenster noch die *alten* Definitionen vor den Änderungen gelten.

2.2 Check Syntax

Zwar prüft der `Run`-Knopf vor der Auswertung des Programms auch dessen syntaktische Korrektheit; oft ist es aber sinnvoll, die syntaktische Korrektheit separat prüfen zu lassen. Dies übernimmt der Knopf `Check Syntax` auf der Werkzeugleiste.

Wenn `Check Syntax` keinen syntaktischen Fehler findet, annotiert es das Programm im Definitionsfenster mit einigen nützlichen Informationen:

- Die verschiedenen Programmelemente werden je nach Art koloriert.
- Die lexikalische Struktur des Programms läßt sich durch das Überstreichen von Bezeichnern mit der Maus visualisieren — DrScheme zeichnet dann Pfeile von der Benutzung von Bezeichnern zurück zu ihren Definitionen. Außerdem erlaubt ein Kontextmenü, Variablen zu verfolgen, umzubenennen oder die Pfeile permanent sichtbar zu machen.
- Endrekursive Aufrufe werden durch lila Pfeile kenntlich gemacht. (Für Anfänger weniger relevant.)

[2] Anders als bei traditionellen Scheme-Systemen beginnt DrScheme bei jedem Druck auf die `Run`-Taste von vorn; alle alten Definitionen werden gelöscht, bevor das Programm in einer frischen Umgebung ausgewertet wird. Damit werden subtile Fehlerquellen ausgeschlossen, die durch die Erhaltung von Zustand entstehen.

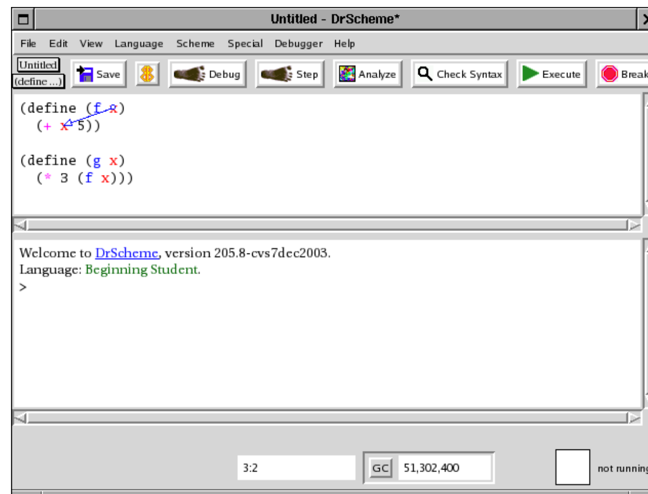


Abbildung 2.3: Check Syntax

2.3 Fehlermeldungen

DrScheme meldet zwei Sorten von Fehlern:

- Syntax-Fehler (auch genannte *statische* Fehler), die bereits von `Check Syntax` oder von `Execute` vor der Auswertung des Programms gefunden werden

- *dynamische* Fehler, die DrScheme bei der Auswertung des Programms findet

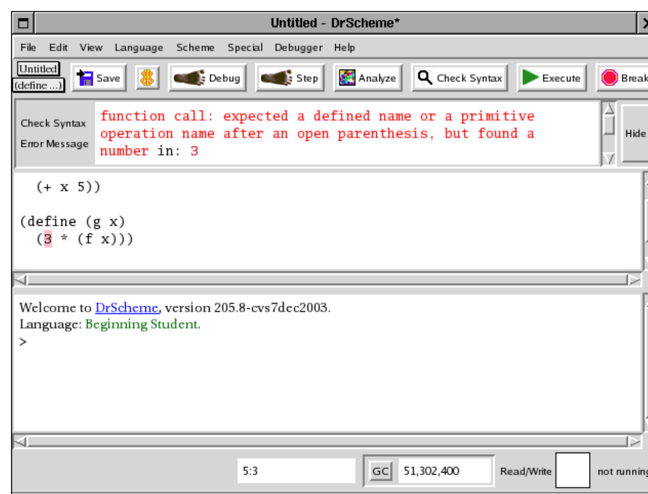


Abbildung 2.4: Statischer Fehler

Abbildung 2.4 zeigt einen statischen Fehler und die zugehörige Fehlermeldung. Der Programmteil, in dem der Fehler festgestellt wurde, ist rosa hervorgehoben.

Bei dynamischen Fehlern (siehe Abbildung 2.5): erscheint Die Fehlermeldung im Interaktionsfenster; wiederum ist der Ausdruck, in dem der Fehler aufgetreten ist, im Definitionsfenster rosa markiert.

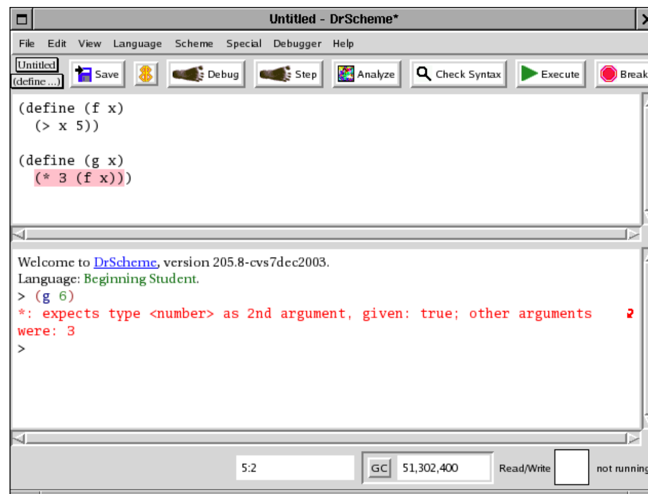


Abbildung 2.5: Dynamischer Fehler

2.4 Algebraischer Stepper

In den Sprachebenen bis „Intermediate Student With Lambda“ folgt die Programmauswertung rein algebraischen Regeln. Es ist also möglich, die Auswertung eines Programms so zu erklären, daß bei der Funktionsanwendung stets die Parameter durch die Argumente ersetzt werden, was genau der Anwendung von Formeln im Mathematikunterricht entspricht.

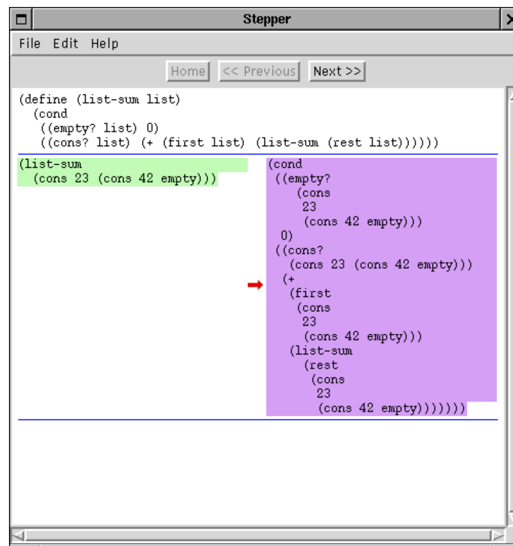


Abbildung 2.6: Der algebraische Stepper

DrScheme erlaubt es, genau diesen Prozeß (in der Version 205 nur für die beiden Sprachebenen zu „Beginning Student“) zu visualisieren. Dieser verbirgt sich in Form des sogenannten *algebraischen Steppers* hinter dem Knopf mit der Aufschrift `Step`. Damit der Stepper funktioniert, muß im Definitionsfenster am Schluß ein Ausdruck stehen, dessen Auswertung visualisiert wird.

Abbildung 2.6 zeigt ein Beispiel: auf der linken Seite steht ein auszuwertender Ausdruck, auf der rechten Seite steht der reduzierte Ausdruck nach einem Auswer-

tungsschritt. Dementsprechend ist links der Teilausdruck grün markiert, der ersetzt wird (der sogenannte *Redex*), auf der rechten Seite ist lila markiert, *wodurch* er ersetzt wurde. Die Auswertung funktioniert auch rückwärts und ermöglicht damit Schülern, durch Ausprobieren und Beobachten ein intuitives Verständnis für den Auswertungsprozeß zu erlangen.

2.5 Systematisches Testen

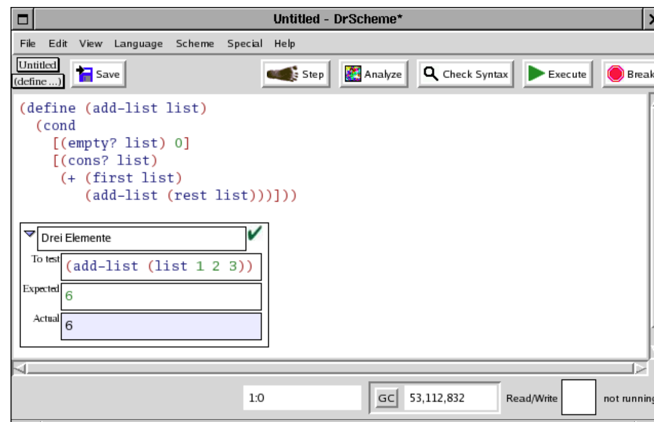


Abbildung 2.7: Testfall als Teil des Scheme-Programms

Die meisten Konstruktionsanleitungen enthalten die Formulierung von Testfällen und die Überprüfung des fertigen Programms anhand der Testfälle als festen Bestandteil. DrScheme bietet spezielle Unterstützung für die Eingabe von Testfällen und automatisierte Durchführung von Tests an. Der Benutzer kann einen Testfall formulieren, indem er den Menüpunkt `Special->Insert Test Case` aktiviert. Es erscheint dann ein in das Programm eingebetteter Kasten. (Siehe Abbildung 2.7.) Dieser hat drei Eingabefelder:

- Das erste Eingabefeld nimmt einen Namen für den Testfall auf. (Dieser kann auch weggelassen werden.)
- Das zweite Feld nimmt einen auszuwertenden Ausdruck auf.
- Das dritte Feld nimmt das erwartete Resultat der Auswertung des Test-Ausdrucks auf.

Die Testfall-Kasten müssen dabei im Programm immer *nach* den Funktionsdefinitionen stehen, die sie testen sollen.

Bei der Ausführung des Programms (nach dem Druck auf den `Run`-Knopf) wertet DrScheme die Ausdrücke aller Testfälle aus und vergleicht die tatsächlichen Resultate mit den eingegebenen. Die Testfall-Kästen werden entsprechend mit einem grünen Haken (für einen erfolgreichen Test) und einem roten Kreuz (für einen fehlgeschlagenen Test) markiert.

Der Benutzer kann durch Druck auf das kleine Dreieck im Testfall-Kasten den Inhalt des Kastens bis auf den Namen verschwinden bzw. wieder auftauchen lassen.

2.6 Sprachebenen

Eines der wichtigsten Aufgaben einer Entwicklungsumgebung für Anfänger ist es, bei auftretenden Fehlern möglichst gutes Feedback zu liefern — idealerweise derge-

stalt, daß sie den Fehler selbst beheben können.

Dabei machen Anfänger (überraschend) häufig Fehler, die damit zusammenhängen, daß sie versehentlich in einen Bereich der Sprache vordringen, der noch nicht im Unterricht behandelt wurde. Die dazugehörigen Fehlermeldungen sind darum häufig für die Schüler nicht verständlich.

Dies ist in Scheme nicht anders. Typisches Beispiel:

```
(define (list-length list)
  (cond
    [(empty? list) 0]
    [(cons? list)
     1 + (list-length (rest list))]))
```

Hier hat der Schüler versehentlich Infix-Syntax benutzt. Dies ist ein syntaktisch korrektes Programm im vollen Sprachumfang von Scheme: Die Ausdrücke `1`, `+` und `(list-length (rest list))` werden hintereinander ausgewertet, nur der letzte aber wird zum Ergebnis. Damit liefert diese Funktion für alle Listen den Wert 0.

Die Hintereinanderauswertung von Ausdrücken wird aber erst relativ spät im Material (mit der Behandlung von Zustand) eingeführt. Darum ist für Schüler nicht unbedingt ersichtlich, warum das obige Programm zwar „läuft“, aber meist das falsche Ergebnis liefert. Darum schränken die frühen Sprachebenen die Sprache ein, um das obige Programm als fehlerhaft zurückweisen zu können und dem Schüler eine entsprechende Fehlermeldung zu präsentieren.

Die Sprachebenen sind aus sehr sorgfältiger Beobachtung entstanden, wie Anfänger lernen und was für Fehler sie machen. Es ist zwar gut möglich, daß es möglich ist, die Sprachebenen besser zu gestalten (ich habe vor, irgendwann einen eigenen Satz Sprachebenen für DeinProgramm zu realisieren), in Ermangelung besseren Wissens kann ich aber nur (im positiven Sinne) raten, die HtDP-Sprachebenen zu verwenden.

Details zum Sprachumfang der jeweiligen Sprachebenen findet sich in der DrScheme-Dokumentation im Abschnitt „Manuals“.

2.6.1 Beginning Student

Diese Sprachebene enthält die nötigsten Sprachelemente, um mit der Programmierung anzufangen. Es gibt jedoch bereits Strukturen, Listen und Rekursion.

Die Listen werden zunächst stets als geschachtelte `cons`-Ausdrücke repräsentiert, um die induktive Struktur klar visuell darzustellen. Das Standard-Ausgabeformat für Listen in Scheme (nur mit Klammern um die Elemente) ist für Anfänger oft verwirrend und nicht mit der Reduktionssemantik, die z.B. der Stepper visualisiert, verträglich.

2.6.2 Beginning Student With List Abbreviations

Der Sprachumfang ist in dieser Ebene unverändert gegenüber der vorigen. Lediglich das Ausgabeformat für Listen ändert sich — sie werden als `list`-Ausdrücke dargestellt. Damit wird den Schülern die Idee der „Elemente einer Liste“ und ihre lineare Struktur verdeutlich. Die Verwendung von `list` sichert außerdem die Visualisierbarkeit im Stepper.

2.6.3 Intermediate Student

Die wichtigste Neuerung in dieser Sprachebene ist die Einführung von lokalen Variablen für die Benennung von Zwischenergebnissen. Diese Erweiterung erfolgt relativ spät, um sicherzustellen, daß die Schüler zunächst trainieren, für jede Größe eines Programms eine separate Funktion zu schreiben.

Die Einführung lokaler Variablen führt häufig dazu, daß Schüler zunächst umfangreiche „Monster-Funktionen“, weil die resultierenden Programme scheinbar kürzer sind. Nach einiger Zeit entdecken sie jedoch erfahrungsgemäß, daß diese Programme schwerer lesbar sind und weniger wiederverwendbare Teile enthalten, als die Programme, die sie vorher geschrieben haben — genügend Übung im Schreiben von Funktionen vorausgesetzt.

2.6.4 Intermediate Student with Lambda

In dieser Sprachebene kommen Funktionen erster Klasse hinzu — eine direkte Übertragung des Abstraktionsprinzips auf Funktionen. Damit können nun beliebige Ausdrücke an der ersten Stelle von Funktionsanwendungen stehen.

2.6.5 Advanced Student

In dieser Sprachebene kommen schließlich Seiteneffekte hinzu — insbesondere also `set!` sowie Mutatoren für Strukturen.

2.7 Teachpacks

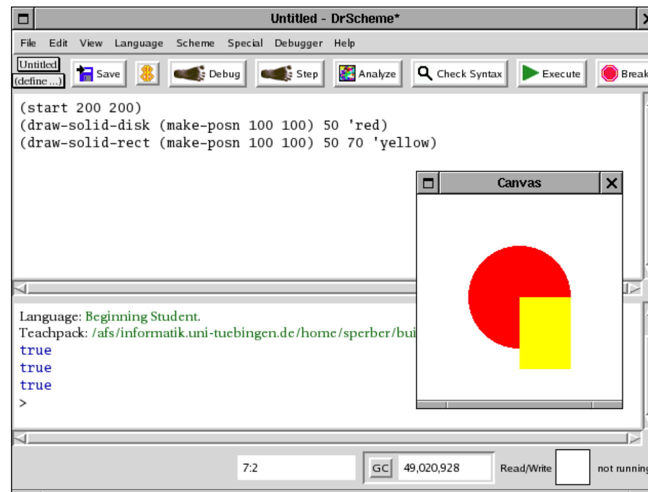


Abbildung 2.8: Benutzung eines Teachpacks

Für manche Programmierübungen ist es notwendig, eine Bibliothek mit Zusatzfunktionalität zu benutzen — etwa für die Programmierung von Grafik, grafischen Benutzeroberflächen, Netzwerkprogrammen, Web-Servern, usw. DrScheme wird mit einer Vielfalt solcher Bibliotheken ausgeliefert.

Während reguläre „Profi“-Scheme-Programme diese Bibliotheken durch spezielle Import-Anweisungen einbinden, bieten die Anfänger-Sprachebenen für viele von ihnen eine einfachere Möglichkeit der Benutzung in Form sogenannter *Teachpacks*. Der Benutzer kann ein Teachpack einfach durch die Anwahl des Menüpunkts `Language->Add Teachpack` und die Auswahl des entsprechenden Teachpacks aktivieren. Die Funktionen des Teachpacks stehen danach zur Verfügung.

Abbildung 2.8 zeigt eine Beispiel-Anwendung des Teachpacks `draw.ss`: Es stellt einige einfache Funktionen zum Zeichnen von Computergrafik zur Verfügung. Weitere Beispiele für Teachpacks und deren Benutzung in speziellen Aufgaben finden sich im Text.

Kapitel 3

Zur Auswahl und Reihenfolge der Themen

3.1 Auswahl

Die Themenauswahl bei DeinProgramm unterscheidet sich signifikant von tradionellen Programmier-Curricula. Dieser Abschnitt kommentiert einige der wichtigsten Unterschiede.

3.1.1 Textuelle Ein-/Ausgabe

Textuelle Ein-/Ausgabe fehlt als Thema vollständig in DeinProgramm — im Gegensatz zur traditionellen Programmierausbildung, wo sie eine entscheidende Rolle spielt: Die meisten Programmierumgebungen erlauben es nicht, die Resultate eines Programmlaufs zu sehen, wenn im Programm keine Ein-/Ausgabe-Anweisungen stehen.

Zunächst einmal ist es didaktisch nicht sinnvoll, überhaupt textuelle Ein-/Ausgabe als Teil der Anfängerausbildung zu behandeln: sie spielt keine fundamentale Rolle im Repertoire eines Programmierers. Da Ein-/Ausgabe (besonders die Eingabe) häufig mühsam zu programmieren ist, stiehlt ihre Behandlung wertvolle Zeit im Unterricht, die besser auf andere Themen verwendet werden können. Hinzu kommt, daß Ein-/Ausgabe Seiteneffekte und damit Fragen zur Auswertungsreihenfolge mit sich bringt. Diese aber werden in DeinProgramm erst viel später im Curriculum behandelt. (Mehr dazu weiter unten in Abschnitt 3.2.)

Die frühe Behandlung von Ein-/Ausgabe hat traditionelle Schüler dazu ermutigt, ihre Programme ausgehend von den Ein-/Ausgabe-Anweisungen zu strukturieren. Die dabei meist entstehenden Abstraktionen sind allerdings nicht wiederverwendbar, da sie Eingaben von der Tastatur einlesen (anstatt sie vom Aufrufer als Argumente geliefert zu bekommen), und berechnete Ergebnisse ausgeben, anstatt sie an den Aufrufer zurückzugeben. Ihre frühe Behandlung hat damit sogar schädliche Auswirkungen.

Textuelle Ein-/Ausgabe spielt außerdem eine zunehmend untergeordnete Rolle bei der Entwicklung von Programmen — grafische Benutzeroberflächen bzw. Web-Interfaces nehmen ihren Platz ein. Zu diesen Themen (insbesondere der Web-Programmierung) findet sich allerdings einiges Material in DeinProgramm.

Glücklicherweise erlaubt Scheme es, beim Schreiben der meisten Programme auf explizite Ein-/Ausgabe ganz zu verzichen: In Scheme haben die meisten Daten eine direkte Schreibweise als Ausdrücke (auch z.B. Listen oder Strukturen), und DrScheme druckt das Ergebnis der Auswertung eines Ausdrucks in der REPL aus.

3.1.2 Objektorientierte Programmierung

Die objektorientierte Programmierung fehlt ebenfalls im Grundcurriculum von DeinProgramm — obwohl die Omnipräsenz des Wortes „objektorientiert“ in der Programmierliteratur suggeriert, daß es sich um eine fundamentale Programmiertechnik handelt. Dies ist allerdings nicht der Fall. Typischerweise handelt es sich bei der objektorientierten Programmierung um eine Kombination mehrerer Techniken. Die wichtigsten sind:

- zusammengesetzte Daten
- gemischte Daten
- datengesteuerte Programmierung
- Mutation
- dynamische Bindung
- Vererbung

Mit Ausnahme der dynamischen Bindung und der Vererbung werden alle diese Themen in DeinProgramm ausführlich behandelt. Der Sprung von DeinProgramm zur „echten“ objektorientierten Programmierung ist damit klein. Es ist geplant, zu diesem Sprung zu einem späteren Zeitpunkt in die Materialien von DeinProgramm zu integrieren.

Das Hauptproblem mit der frühen Behandlung der objektorientierten Programmierung ist, daß die obige Zusammenstellung von Techniken vollkommen willkürlich ist: Verschiedene Schulen der objektorientierten Programmierung und verschiedene objektorientierte Programmiersprachen verwenden unterschiedliche Zusammenstellungen. Außerdem hat der Begriff „Vererbung“ eine Vielzahl möglicher Interpretationen, von denen sich manche paarweise ausschließen. (Des weiteren ist die Interaktion von Vererbung und dynamischer Bindung ziemlich kompliziert und führt oft zu Verwirrung.) Aus diesen Gründen ist es nicht sinnvoll, die objektorientierte Programmierung früh im Curriculum zu behandeln.

3.2 Reihenfolge

3.2.1 Rekursion

Die wichtigsten Entscheidungen, die in DeinProgramm bezüglich der Reihenfolge getroffen wurden, haben mit der Behandlung der Rekursion zu tun:

Traditionelle Curricula behandeln häufig die Rekursion zunächst anhand von Funktionen auf Zahlen, meist Fakultät und Potenz. Dabei entstehen folgende Probleme:

- Schüler haben häufig große Schwierigkeiten, die Funktionsweise rekursiver Funktionen anhand dieser Beispiele zu verstehen. Insbesondere ist es schwer, aufzuzeigen, daß die Rekursion völlig systematisch aus dem Aufbau der natürlichen Zahlen hervorgeht, da Schülern dieser (induktive) Aufbau der Intuition der Schüler nicht entspricht.
- Es ist schwierig, einfache systematische Programmschablonen zu erstellen, da die Basis der Induktion manchmal bei 0 und manchmal bei 1 liegt — je nach Aufgabe.

Es entstehen die allseits bekannten „Kopfschmerzen“ vieler Schüler, wenn sie an Rekursion denken. Diese setzten sich häufig bis ins Hauptstudium der Informatik fort.

Die Probleme werden noch verschärft durch die Behandlung nicht direkt strukturell rekursiver Funktionen wie z.B. der Fibonacci-Funktion. Schüler werden zwar gelegentlich sagen, daß sie diese Funktionen verstehen — meist sind sie aber nicht in der Lage, ähnliche rekursive Funktionen selbst zu schreiben. Letzteres — die systematische Entwicklung von Funktionen — ist aber gerade das Hauptanliegen von DeinProgramm.

Aus diesen Gründen ist es sinnvoller, mit Listen zu beginnen — diese haben eine klar erkennbar induktive Struktur. Außerdem ist es möglich, die schon bekannten Techniken und Schablonen zum Umgang mit zusammengesetzten und gemischten Daten zu verwenden — lediglich der rekursive Aufruf kommt noch hinzu. Damit können die Schüler bereits rekursive Funktionen mit Hilfe der Schablone schreiben, auch wenn sie das Konzept der induktiven Datenstruktur mental noch nicht vollständig durchdrungen haben. Der praktische Umgang besorgt dann erfahrungsgemäß den Rest, auch bei Schülern, die in traditionellen Curricula keinerlei Zugang zur Rekursion finden.

Bei der Auswahl der Beispiele und Übungsaufgaben müssen wir noch einmal betonen, daß diese anfangs *vollkommen systematisch durch Anwendung der Funktionsschablonen* bearbeitet werden können müssen, um früh für Erfolgserlebnisse zu sorgen und damit zu unverkrampftem Umgang mit dem Thema Rekursion führen.

3.2.2 Zusammengesetzte Daten

Bei DeinProgramm werden zusammengesetzte Daten relativ früh eingeführt: Zusammengesetzte Daten sind der (erste) entscheidende Schlüssel zu erfolgreicher Datenmodellierung und spielen damit eine zentrale Rolle in der Programmierung. Gleichzeitig müssen Schüler die Spezifikation zusammengesetzter Daten möglichst viel üben, um dies als selbstverständlichen Teil des Schreibens von Programmen anzusehen.

Viele Schüler haben anfangs Schwierigkeiten mit der Idee, daß „mehrere Dinge zusammen eins werden“, und es ist darum sinnvoll, mit der Materie früh und behutsam anzufangen und diese dann über einen längeren Zeitraum zu üben. (Bei diesem scheinbar einfachen Prinzip haben nicht nur Schüler, sondern auch viele professionelle Programmierer auffallende Defizite.)

3.2.3 Zuweisung erst am Schluß

Zuweisungen (bzw. Mutation) werden in DeinProgramm erst sehr spät behandelt. Dies steht wiederum im Gegensatz zur traditionellen Programmierausbildung: In den dort üblichen imperativen Programmiersprachen muß die Zuweisung sehr früh behandelt werden, weil es ohne sie nicht möglich ist, nicht-triviale Programme zu schreiben. (Technisch liegt das daran, daß in diesen Sprachen die Menge der ausdrückbaren Werte sehr klein ist.) Stattdessen sind bei DeinProgramm alle Programme am Anfang der Ausbildung *rein funktional*, verwenden also keinerlei *Seiteneffekte* (wozu u.a. Zuweisung, Mutation und Ein-/Ausgabe gehören).

Die rein funktionale Programmierung vermeidet viele der Probleme, die mit dem Verständnis und dem Schreiben korrekter Programme zusammenhängen:

- Alle Ausdrücke lassen sich nach den vertrauten Regeln der Algebra auswerten.
- Die Auswertung läßt sich (z.B. mit dem Stepper in DrScheme) visualisieren.
- Es ist realistisch, die Korrektheit kleiner Programme zu beweisen.

Die mathematischen und mentalen Modelle für die Auswerung imperativer Programme hingegen sind deutlich komplizierter und belasten den anfänglichen Unterricht unnötig. Deswegen schreiben Scheme-Experten auch, wann immer möglich, rein funktionalen Programmcode.

Trotz allem ist die Zuweisung bzw. die Verwendung von Zustand eine wichtige Technik für die Entwicklung modularer Programme. Aus diesem Grund wird sie auch — allerdings gegen Ende des Curriculums — in DeinProgramm ausführlich behandelt.

Kapitel 4

Ausdrücke

4.1 Bilder

Bilder sind beim Einstieg in die Programmierung eine attraktive Alternative zu den gelegentlich etwas trockenen Zahlen. Die HtDP-Sprachebenen enthalten einige Funktionen für die Konstruktion und Kombination von Bildern. (Ein Überblick befindet sich in Abschnitt A.3.5.)

Die Konstruktionsmethode für diese Bilder unterscheidet sich dabei von der traditioneller Grafikprogrammierung, wie etwa im Teachpack `draw.ss`: Die Bilder sind Objekte, die als ganzes erzeugt und mit anderen kombiniert werden, anstatt daß auf eine virtuelle Leinwand Linien und Figuren gezeichnet werden. Die Funktionen, die auf Bildern operieren, sind also auch Funktionen im mathematischen Sinne, ohne Seiteneffekte — damit eignen sie sich auch für den Einsatz ganz am Anfang der Programmierausbildung. Die Bilder werden dabei, genau wie etwa Zahlen auch, in der REPL direkt angezeigt.

Abbildung 4.1 zeigt ein Beispiel für die Konstruktion eines Gesichts aus Einzelteilen.

Es ist außerdem möglich, externe Bilder, die als Grafikdateien vorhanden sind, einfach über `Special` → `Insert Image ...` in das Programm einzukleben.

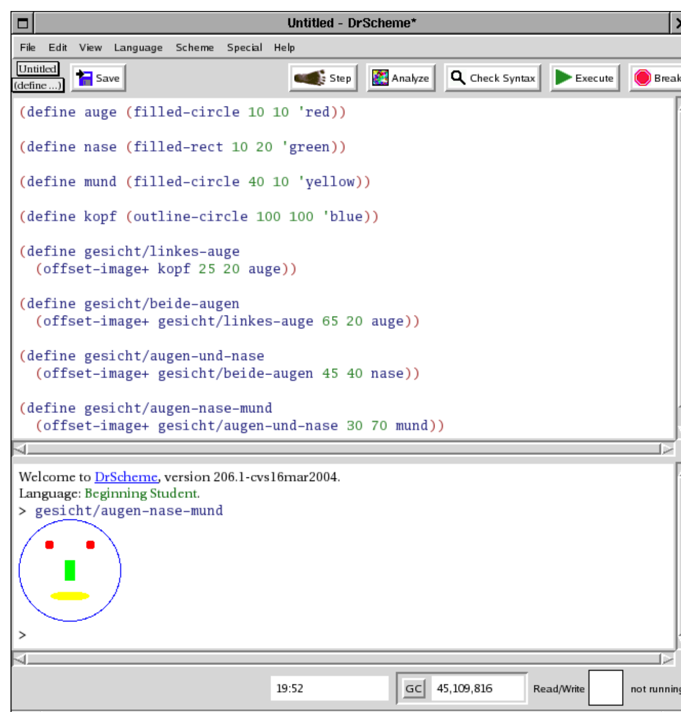


Abbildung 4.1: Konstruktion eines Gesichts

Kapitel 5

Konstruktionsanleitungen für Programme

Das wichtigste Hilfsmittel beim Erlernen des Programmierens sind die *Konstruktionsanleitungen*. Eine Konstruktionsanleitung bezieht sich zunächst auf das Schreiben einer einzelnen Funktion.[1] Eine Konstruktionsanleitung ist systematisch und weitgehend eindeutig nachvollziehbar. Jede Konstruktionsanleitung folgt dabei im wesentlichen dem gleichen Arbeitsplan:

1. Problem- und Datenanalyse
2. Beschreibung der Aufgabe der Funktion
3. Vertrag für die Funktion
4. Funktionsgerüst
5. Beispiele
6. Funktionsschablone
7. Funktionsrumpf
8. Tests

Dabei produziert jeder dieser Arbeitsschritte ein konkretes Produkt — dieses Produkt dient jeweils sowohl Schülern als auch Lehrern der Kontrolle des Fortschritts:

1. eine Beschreibung der Daten, die in der Aufgabe vorkommen
2. eine informelle Beschreibung der Aufgabe der Funktion
3. eine halb-formelle Beschreibung der Ein- und Ausgabedaten der Funktion
4. die erste Zeile der späteren Funktionsdefinition
5. Beispiele, welche die Erwartungen an die Funktionsweise der Funktion aufzeigen
6. die Teile der Funktion, die sich direkt aus Vertrag und Datenanalyse ergeben
7. die vollständige Funktionsdefinition

[1] Später werden wir Richtlinien und Techniken entwickeln, die entscheiden helfen, wie eine Problemlösung in mehrere Funktionen zerlegt werden kann.

8. Belege für das korrekte (oder inkorrekte) Verhalten der Funktion

Es ist dabei wichtig, für *jede* Aufgabe die Abarbeitung *aller* Arbeitsschritte zu fordern, so pedantisch dies auch erscheinen mag: Der Arbeitsplan und die Konstruktionsanleitungen geben Schülern stets eine klar identifizierten nächsten Arbeitsschritt an die Hand und verhindern damit, daß sie an einer komplexen Aufgabenstellung scheitern, noch bevor sie mit ihren machbaren Teilen angefangen haben.

Jede Konstruktionsanleitung gibt konkrete Handlungsanweisungen für die Schritte des Arbeitsplans. Diese Anweisungen richten sich dabei stets nach den Daten, die an dem Problem beteiligt. Die Anleitungen sind also *datengesteuert*.

Kapitel 6

Einfache Funktionen mit Zahlen

In diesem Kapitel geht es um Funktionen, die aus Zahlen andere Zahlen nach Rechenvorschriften berechnen — also um programmierte Elemente einer Formelsammlung.

6.1 Konstruktionsanleitung

Die folgende Anleitung ist möglicherweise die erste, welche die Schüler kennenlernen. Darum enthält sie zu jedem Punkt Anleitungen — auch wenn diese Anleitungen sich nicht speziell auf Funktionen mit Zahlen beziehen.

6.1.1 Funktionen mit einer Eingabe

Datenanalyse Diese Anleitung ist für Aufgaben gedacht, in denen alle vorkommenden Größen Zahlen sind — gehe also sicher, daß in der Aufgabe nur Zahlen, aber keine Bilder vorkommen.

Beschreibung Beschreibe kurz Sinn und Zweck der Prozedur.

Vertrag Wähle einen Namen für die Funktion. Untersuche die Problembeschreibung nach Hinweisen, welche Größe die „Eingabe“ und welche Größe die „Ausgabe“ ist. Schreibe dann den Vertrag:

```
;; f: zahl -> zahl
```

Funktionsgerüst Wähle einen Namen für die Eingabe — den *Parameter* also. Schreibe dann ein Funktionsgerüst in der Form

```
(define (f p)
  ...)
```

Beispiele Suche in der Aufgabenstellung nach Beispielen für korrekte Ausgaben der Funktion oder konstruiere selbst welche. (Dafür mußt Du die Aufgabe erst einmal „von Hand“ lösen.) Gib diese als Testfälle ein.

Funktionsschablone Im Rumpf der Funktion muß der Parameter vorkommen. Schreibe ihn deshalb schon einmal herein:

```
(define (f p)
  ... p ...)
```

Rumpf Vervollständige die Funktionsschablone entsprechend der Aufgabenstellung.

Tests Drücke auf den Run-Knopf und stelle sicher, daß alle Testfälle erfolgreich durchlaufen. Falls nicht, lese Dein Programm noch einmal und finde heraus, wo das Problem herkommt. Dann behebe den Fehler.

6.1.2 Funktionen mit mehreren Eingaben

Diese Konstruktionsanleitung führt nur diejenigen Teile auf, die sich gegenüber der Anleitung für Funktionen mit einem Parameter geändert haben.

Vertrag Untersuche die Problembeschreibung nach Hinweisen, welche bzw. wieviele Größen durch die Aufgabenstellung gegeben sind — die „Eingaben" also, und welches die „Unbekannte" oder die „Ausgabe" ist. Schreibe dann den Vertrag:

```
;; f: zahl ... -> zahl
```

Achte darauf, daß vor dem `->` genauso oft `zahl` steht, wie die Funktion später Eingaben haben sollen.

Funktionsgerüst Wähle einen Namen für jeden Parameter. Schreibe dann ein Funktionsgerüst in der Form

```
(define (f p1 ... pn)
  ...)
```

Funktionsschablone Die Parameter müssen allesamt im späteren Funktionsrumpf vorkommen:

```
(define (f p1 ... pn)
  ... p1 ... ... ... pn ...)
```

6.2 Einfache Aufgaben

Gefragt ist eine Funktion, die für eine gegebene Anzahl Gummidrops den Preis ausrechnet. Ein Gummidrop kostet hier 5 Cents. Hier sind die einzelnen Schritte der dazu passenden Konstruktionsanleitung:

1. Die Datenanalyse ergibt, daß es ausschließlich um ganze Zahlen geht: die Anzahl der Gummidrops, und Preise in Cents.

2. Die Beschreibung steht in diesem Fall schon kompakt in der Aufgabenstellung: Gefragt sei eine Funktion, die für eine gegebene Anzahl Gummidrops den Preis ausrechnet.

3. Die Funktion wird die Anzahl der Gummidrops konsumieren und den Preis zurückgeben:

```
;; preis-gummidrops: zahl -> zahl
```

4. Das Funktionsgerüst ergibt sich direkt aus der Aufgabenbeschreibung und dem Vertrag:

```
(define (preis-gummidrops anzahl)
  ...)
```

5. Testfälle:

To test	(preis-gummidrops 0)
Expected	0

To test	(preis-gummidrops 1)
Expected	5

To test	(preis-gummidrops 5)
Expected	25

To test	(preis-gummidrops 10000000000000)
Expected	50000000000000

6. Die Funktionsschablone vervollständigt das Funktionsgerüst. In diesem Fall konsumiert die Funktion nur eine Zahl — ein Objekt eines „primitiven" Datentyps — die irgendwo im Rumpf vorkommen muß:

```
(define (preis-gummidrops anzahl)
  ... anzahl ...)
```

7. Der Funktionsrumpf schließlich vervollständigt die Schablone:

```
(define (preis-gummidrops anzahl)
  (* anzahl 5))
```

8. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

Hier das Endprodukt:

```
;; für eine gegebene Anzahl Gummidrops den Preis ausrechnen
;; preis-gummidrops: zahl -> zahl

(define (preis-gummidrops anzahl)
  (* anzahl 5))
```

To test	(preis-gummidrops 0)
Expected	0

To test	(preis-gummidrops 1)
Expected	5

To test	(preis-gummidrops 5)
Expected	25

To test	(preis-gummidrops 10000000000000)
Expected	50000000000000

Anmerkungen:

- Die meisten Schritte der Aufgabenlösung folgen rein systematischem Vorgehen und laufen immer auf die gleiche Art und Weise statt. Es ist wichtig, daß Schüler diese Vorgänge nicht nur selbst üben, sondern sich auch bewußt machen, daß es sich immer um die gleichen Vorgänge handelt.

- Die Konstruktion des Funktionsrumpfes ist der einzig wirklich „kreative“ Teil der Aufgabenlösung, da er nicht nur von der Form der Aufgabe abhängt, sondern von ihrem Inhalt: dieser Schritt ist vom Einsatz *bereichsspezifischen Wissens* abhängig. Dieses Wissen kommt dabei z.B. aus dem Alltag (wie in diesem Fall), dem Unterricht oder einer Formelsammlung.

- Der Name sollte die Aufgabe der Funktion möglichst direkt, prägnant und knapp wiedergeben. Die Auswahl passender Namen (für Funktionen, aber auch Parameter) sollte im Unterricht trainiert werden.

- DrScheme erlaubt es, den Vorgang des Testens zu automatisieren. Dazu gibt es die Möglichkeit, sogenannte *Test-Suiten* zu erzeugen, und alle Tests durch das Drücken eines einzelnen Knopfs durchlaufen und die Ergebnisse überprüfen zu lassen.

Aufgabe 6.1 Schreibe eine Funktion `DM->Euro`, die einen Geldbetrag in DM konsumiert, und den entsprechenden Euro-Betrag zurückliefert. Schreibe eine Funktion `Euro->DM`, welche die umgekehrte Aufgabe erledigt.

Aufgabe 6.2 In den USA wird eine andere Maßeinheit für Temperaturen verwendet, das *Fahrenheit*. Eine Temperatur in Fahrenheit wird in eine Grad-Celsius-Temperatur umgewandelt, indem von ihr 32 abgezogen wird, und das Ergebnis dann mit $\frac{5}{9}$ multipliziert wird. Schreibe eine Funktion `fahrenheit->celsius`, die eine Temperatur in Fahrenheit konsumiert und dine Tmeperatur in Grad Celsius zurückgibt.

Aufgabe 6.2 eignet sich für den Einsatz eines *Teachpacks*: ein Teachpack ist eine Sammlung vordefinierter Funktionen oder ein halbfertiges Programm, das durch die Lösung einer Aufgabe vervollständigt wird. In diesem Fall ist in DrScheme ein Teachpack namens `convert.ss` enthalten, das die Arbeitsweise der Funktion `fahrenheit->celsius` veranschaulicht. Nach Laden des Teachpacks führt die Auswertung von

```
(convert-gui fahrenheit->celsius)
```

zur Anzeige einer kleinen grafischen Benutzeroberfläche, mit der die Funktion ausprobiert werden kann.

Aufgabe 6.3 Schreibe eine Funktion `nettolohn`, welche den Nettolohn eines Arbeitnehmers ausrechnet, der stundenweise bezahlt wird. Die Funktion soll die Anzahl der Stunden konsumieren und den Nettolohn zurückgeben. Dabei zahlt die Firma einen Bruttolohn von 20€ pro Stunde. Der Arbeitnehmer muß aber 40% Steuern auf den Bruttolohn bezahlen.

6.3 Funktionen mit mehreren Eingaben

Auf einer Schulfeier verkaufen die Schüler Becher mit Punsch, um Geld für den Kauf von Material für die Verschönerung ihres Klassenraums zu sammeln. Sie versuchen

vorher auszurechnen, wieviel Profit ihnen der Verkauf bringen wird: Sie haben dabei bestimmte Fixkosten (den Topf für die Zubereitung zum Beispiel), bestimmte Herstellungskosten pro Becher (die Zutaten) und müssen beide durch den Verkauf wieder einspielen. Die Fixkosten sollen bei 20€, die Herstellungskosten pro Becher bei 0,20€ liegen. Gefragt ist eine Funktion `gewinn`, welche die Anzahl der verkauften Becher und den Verkaufspreis eines Bechers konsumiert, und den Gewinn aus dem Verkauf zurückgibt.

Beispiellösung:

1. Die Datenanalyse ist wiederum einfach: Es geht ausschließlich um Zahlen.

2. Die Kurzbeschreibung ist wiederum in der Aufgabenstellung enthalten: Es geht um den Gewinn aus dem Verkauf einer bestimmten Anzahl von Bechern Punsch zu einem bestimmten Preis pro Becher.

3. Die Funktion wird eine Anzahl von Bechern sowie den Preis pro Becher konsumieren und den Gewinn zurückgeben. Der Vertrag ist also:

```
;; gewinn: zahl zahl -> zahl
```

4. Das Funktionsgerüst ergibt sich wieder aus Aufgabenbeschreibung und Vertrag:

```
(define (gewinn anzahl-becher preis-pro-becher)
  ...)
```

5. Beispiele:

To test	`(gewinn 0 1)`
Expected	`-20`

To test	`(gewinn 1 1)`
Expected	`-19.2`

To test	`(gewinn 100 1)`
Expected	`60`

6. Wiederum geht es um (unstrukturierte) Zahlen, wichtig ist also lediglich, daß die beiden Parameter im Funktionsrumpf vorkommen. Hier ist die Schablone:

```
(define (gewinn anzahl-becher preis-pro-becher)
  ... anzahl-becher ... preis-pro-becher ...)
```

7. Der Funktionsrumpf vervollständigt die Schablone:

```
(define (gewinn anzahl-becher preis-pro-becher)
  (+ -20
     (* -0.2 anzahl-becher)
     (* preis-pro-becher anzahl-becher)))
```

8. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

Hier das Endprodukt:

```
;; für eine Anzahl von Bechern sowie den Preis pro
;; Becher den Gewinn zurückgeben

;; gewinn: zahl zahl -> zahl

(define (gewinn anzahl-becher preis-pro-becher)
  (+ -20
     (* -0.2 anzahl-becher)
     (* preis-pro-becher anzahl-becher)))
```

To test	`(gewinn 0 1)`
Expected	`-20`

To test	`(gewinn 1 1)`
Expected	`-19.2`

To test	`(gewinn 100 1)`
Expected	`60`

Anmerkungen:

- DrScheme repräsentiert in den HtDP-Modi alle Zahlen standardmäßig als präzise Brüche, die lediglich in Dezimalnotation ausgedruckt werden. Die üblichen unangenehmen Effekte, die in anderen Sprachen/Umgebungen durch die Verwendung von binärer Fließkommadarstellung auftreten, entfallen also.
- Die Signaturnotation wie in

  ```
  gewinn: zahl zahl -> zahl
  ```

  entspricht zwar nicht der matematischen Gepflogenheit, ein kartesisches Produkt („`zahl x zahl -> zahl`“) für die Notation von Funktionstypen für mehrere Parameter zu verwenden, ist dafür aber für Anfänger insbesondere in der Mittelstufe einfacher nachzuvollziehen.

Aufgabe 6.4 Zu einer dreistelligen Zahl gehören die Hunderter, Zehner und Einer. Schreibe eine Funktion `dreistellige-zahl`, welche drei Zahlen konsumiert, nämlich die Hunderter, Zehner und Einer einer Zahl, und die dazu passende Zahl zurückgibt.

6.4 Zusammengesetzte Aufgaben

Auszählungen bei der Schulfeier des Vorjahres und eine Umfrage haben ergeben, daß beim Schulfeier-Punsch-Verkauf die Anzahl der verkauften Becher mit dem Preis zusammenhängt. Bei 1€ pro Becher werden insgesamt 200 Becher verkauft. Für jede zusätzliche 5 Cents, die ein Becher kostet, werden zehn Becher weniger verkauft. Schreibe eine Funktion, die für den Verkaufspreis die Anzahl der verkauften Becher ausrechnet!

1. Wieder sind nur (unstrukturierte) Zahlen an der Aufgabe beteiligt.
2. Gefordert ist eine Funktion, die einen Preis pro Becher konsumiert und die Anzahl der verkauften Becher zurückgibt.
3. Hier ist der Vertrag:

```
;; verkaufte-becher: zahl -> zahl
```

4. Das Funktionsgerüst ergibt sich direkt aus der Aufgabenbeschreibung und dem Vertrag:

```
(define (verkaufte-becher preis-pro-becher)
  ...)
```

5. Beispiele:

To test	`(verkaufte-becher 1)`
Expected	`200`

To test	`(verkaufte-becher 1.05)`
Expected	`190`

To test	`(verkaufte-becher 1.50)`
Expected	`100`

6. Die Funktionsschablone führt den Parameter im Rumpf auf:

```
(define (verkaufte-becher preis-pro-becher)
  ... preis-pro-becher ...)
```

7. Der Funktionsrumpf vervollständigt die Schablone:

```
(define (verkaufte-becher preis-pro-becher)
  (- 200
     (* (/ (- preis-pro-becher 1) 0.05)
        10)))
```

8. Die Tests bestätigen die Korrektheit der Lösung.

Hier die fertige Lösung:

```
;; aus dem Preis pro Becher die Anzahl der verkauften Becher berechnen

;; verkaufte-becher: zahl -> zahl

(define (verkaufte-becher preis-pro-becher)
  (- 200
     (* (/ (- preis-pro-becher 1) 0.05)
        10)))
```

To test	`(verkaufte-becher 1)`
Expected	`200`

To test	`(verkaufte-becher 1.05)`
Expected	`190`

To test	`(verkaufte-becher 1.50)`
Expected	`100`

Anmerkungen:

- Es lohnt sich, schon einmal darauf hinzuweisen bzw. die Schüler herausfinden zu lassen, daß `verkaufte-becher` für manche durchaus sinnvolle Eingaben vollkommen unsinnige Ergebnisse liefert.

Gefragt ist jetzt eine Funktion, die ausschließlich aus dem Preis eines Bechers Punsch den zu erwartenden Gewinn zurückgibt.

1. Es geht wieder ausschließlich um Zahlen.
2. Die Kurzbeschreibung ist wiederum in der Aufgabenstellung enthalten: Es geht um den Gewinn aus dem Verkauf von Bechern Punsch zu einem bestimmten Preis pro Becher.
3. Die Funktion wird den Preis pro Becher konsumieren und den Gewinn zurückgeben. Der Vertrag ist also:

```
;; gewinn-fuer-preis: zahl -> zahl
```

4. Das Funktionsgerüst ergibt sich wieder aus Aufgabenbeschreibung und Vertrag:

```
(define (gewinn-fuer-preis preis-pro-becher)
  ...)
```

5. Beispiele:

To test	(gewinn-fuer-preis 1.0)
Expected	140

To test	(gewinn-fuer-preis 1.5)
Expected	110

To test	(gewinn-fuer-preis 2.0)
Expected	-20

6. Wiederum geht es um (unstrukturierte) Zahlen, wichtig ist also lediglich, daß die beiden Parameter im Funktionsrumpf vorkommen. Hier ist die Schablone:

```
(define (gewinn-fuer-preis preis-pro-becher)
  ... preis-pro-becher ...)
```

7. Bei der Konstruktion des Rumpfes bietet es sich nun an, die bereits geschriebenen Funktionen `gewinn` und `verkaufte-becher` einzusetzen:

```
(define (gewinn-fuer-preis preis-pro-becher)
  (gewinn (verkaufte-becher preis-pro-becher)
          preis-pro-becher))
```

8. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

Hier die fertige Lösung:

```
;; gewinn-fuer-preis: zahl -> zahl

;; zu einem Preis pro Becher Punsch den Gewinn ausrechnen

(define (gewinn-fuer-preis preis-pro-becher)
  (gewinn (verkaufte-becher preis-pro-becher)
          preis-pro-becher))
```

To test	`(gewinn-fuer-preis 1.0)`
Expected	`140`

To test	`(gewinn-fuer-preis 1.5)`
Expected	`110`

To test	`(gewinn-fuer-preis 2.0)`
Expected	`-20`

„Durch die Hintertür“ taucht hier jetzt die erste Lösung einer Aufgabe auf, die aus mehreren Funktionen besteht. Das darunterliegende Prinzip ist das wichtigste Prinzip der Programmierung überhaupt und betrifft die Zerlegung einer komplexen Aufgabenstellung in mehrere Teile und damit ihrer Lösung in mehrere Funktionen. Es ist sinnvoll, schon hier auf den folgenden Zusammenhang hinzuweisen:

Der Gewinn zu einem bestimmten Preis pro Becher *hängt* von der Anzahl der verkauften Becher und dem Preis pro Becher *ab*. Die Anzahl der verkauften Becher ist also, weil sie erst noch errechnet werden muß, eine *Zwischengröße* der Aufgabe und verdient damit eine eigene Funktion.

Mantra 1 Zusammengesetzte Aufgaben enthalten *Zwischengrößen*: Sie sind daran zu erkennen, daß sie nicht direkt zur Eingabe gehören, also noch errechnet werden müssen, aber auch nicht direkt die Ausgabe des Programms sind. Schreibe für jede Zwischengröße eine eigene Funktion.

Es ist von Schülern nicht zu erwarten, daß sie nach diesem einen Beispiel bereits eigenständig zusammengesetzte Aufgabenstellungen in mehrere Funktionen zerlegen können. Dies läßt sich effektiver an ihren eigenen Lösungen aufzeigen — nämlich, wenn sie an der Komplexität der erarbeiteten Funktion scheitern. Dann ist es Zeit, das Mantra erneut zu behandeln und auf die Aufgabe anzuwenden.

Kapitel 7

Einfache Funktionen mit Bildern

HIER DIE KONSTRUKTIONSANLEITUNG IN REINFORM

7.1 Einfache Aufgaben

Gefragt ist eine Funktion, die für ein gegebenes Bild eines ausrechnet, das genauso aussieht, nur mit einem schwarzen Rahmen drumherum. Hier sind die einzelnen Schritte der dazu passenden Konstruktionsanleitung:

1. Die Datenanalyse ergibt, daß es ausschließlich um Bilder geht.

2. Die Beschreibung steht in diesem Fall schon kompakt in der Aufgabenstellung.

3. Die Funktion wird ein Bild konsumieren und ein weiteres zurückgeben:

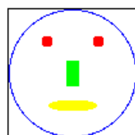
   ```
   ;; mit-rahmen: bild -> bild
   ```

4. Das Funktionsgerüst ergibt sich direkt aus der Aufgabenbeschreibung und dem Vertrag:

   ```
   (define (mit-rahmen bild)
     ...)
   ```

5. Beispiele (lassen sich am besten ausdrucken und austeilen oder auf Papier aufzeichnen):

   `(mit-rahmen gesicht/augen-nase-mund)` $\Longrightarrow$

6. Die Funktionsschablone vervollständigt das Funktionsgerüst. In diesem Fall konsumiert die Funktion nur ein Bild, das irgendwo im Rumpf mindestens einmal vorkommen muß:

   ```
   (define (mit-rahmen bild)
     ... bild ...)
   ```

7. Der Funktionsrumpf schließlich vervollständigt die Schablone. Dazu ist es wahrscheinlich am sinnvollsten, schrittweise und systematisch vorzugehen. Die Aufgabenstellung spricht von einem Rahmen — also einem Rechteck, das mindestens so groß ist wie das Bild selbst:

   ```
   (define (mit-rahmen bild)
     ... (outline-rect breite hoehe 'black) ...)
   ```

   Hier müssen natürlich noch `breite` und `hoehe` ausgefüllt werden.

   ```
   (define (mit-rahmen bild)
     ...
       (outline-rect (image-width bild)
                     (image-height bild)
                     'black)
     ...)
   ```

   Schließlich muß der Rahmen noch mit dem ursprünglichen Bild kombiniert werden:

   ```
   (define (mit-rahmen bild)
    (image+ (outline-rect (image-width bild)
                          (image-height bild)
                          'black)
            bild))
   ```

   Dies wäre der erste Anlauf. Er enthält noch Fehler, aber es ist an der Zeit, das Programm zu testen:

8. Der Test anhand des vorher erdachten Beispiels ergibt folgendes:

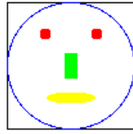
   (mit-rahmen gesicht/augen-nase-mund) $\Longrightarrow$

   Das sieht schon nichtr schlecht aus, aber unterscheidet sich in einem Detail vom ursprünglichen Ziel: Das Bild liegt an den Rändern über dem Rahmen, der Rahmen ist also noch — an jeder Seite um genau einen Punkt — zu klein. Neuer Versuch:

   ```
   (define (mit-rahmen bild)
    (image+ (outline-rect (+ 2 (image-width bild))
                          (+ 2 (image-height bild))
                          'black)
            bild))
   ```

   Ein erneuter Test ergibt:

   (mit-rahmen gesicht/augen-nase-mund) $\Longrightarrow$



Das ist immer noch nicht ganz richtig — das innere Bild muß gegenüber dem Rahmen noch um jeweils einen Punkt nach rechts und unten verschoben werden. Dazu gibts es die Funktion `offset-image+`:

```
(define (mit-rahmen bild)
  (offset-image+ (outline-rect (+ 2 (image-width bild))
                               (+ 2 (image-height bild))
                               'black)
                 1 1
                 bild))
```

Diese Version liefert schließlich das gewünschte Ergebnis.

Hier das Endprodukt:

```
;; für ein Bild ein ebensolches mit Rahmen drumherum liefern
;; mit-rahmen: bild -> bild

(mit-rahmen gesicht/augen-nase-mund) =>
```

```
(define (mit-rahmen bild)
  (offset-image+ (outline-rect (+ 2 (image-width bild))
                               (+ 2 (image-height bild))
                               'black)
                 1 1
                 bild))
```

Anmerkungen:

- Diese Aufgabe ist ein gutes Beispiel für eine Aufgabenlösung mit mehreren Schritten, von denen sich alle direkt an der Aufgabenstellung orientieren.

Aufgabe 7.1 Schreibe eine Funktion `mit-ellipsen-rahmen`, die ein Bild konsumiert und es — mit einem ellipsenförmigen Rahmen versehen — wieder zurückgibt.

7.2 Funktionen mit mehreren Argumenten

Aufgabe 7.2 Erweitere die Funktionen `mit-rahmen` so, daß sie folgendem Vertrag entspricht:

```
;; mit-rahmen: bild zahl symbol -> bild
```

Die neue Funktion soll also eine zusätzliche Zahl und ein Symbol konsumieren. Die Zahl soll dabei den Abstand des Rahmens um den Rest des Bilds bezeichnen, das Symbol dessen Farbe.

Aufgabe 7.3 Schreibe eine Funktion mit folgendem Vertrag

```
;; nebeneinander: bild bild -> bild
```

die zwei Bilder konsumiert und ein Bild zurückgibt, in dem beide Bilder nebeneinander stehen. Nimm dabei vereinfachend an, daß beide Bilder gleich hoch sind. (Zusatzaufgabe: Was für Probleme entstehen, wenn die Bilder nicht gleich hoch sind.)

Schreibe eine Funktion mit folgendem Vertrag

```
;; uebereinander: bild bild -> bild
```

die zwei Bilder konsumiert und ein Bild zurückgibt, in dem beide Bilder übereinander stehen. Nimm dabei vereinfachend an, daß beide Bilder gleich breit sind.

Zusatzaufgabe: Was für Probleme entstehen, wenn die Bilder nicht gleich hoch bzw. nicht gleich breit sind?

Kapitel 8

Fallunterscheidungen

8.1 Konstruktionsanleitung

Datenanalyse Wenn eine Aufgabenstellung bei einer Größe zwischen verschiedenen Klassen unterscheidet, in die der Wert der Größe fallen kann, mußt Du zunächst alle diese Klassen identifizieren. Die Datendefinition sollte diese Klassen darstellen.

Testfälle Es sollte mindestens einen Testfall für jede Klasse der Eingabewerte geben.

Funktionsschablone Der Funktionsrumpf besteht zunächst aus einem `cond`-Ausdruck, der *für jede Klasse der Eingabewerte einen Zweig* aufweist:

```
(define (f a)
  (cond
    [... ...]
    ...
    [... ...]))
```

Funktionsrumpf Als nächstes mußt Du für jeden Zweig eine *Bedingung* oder *Frage* aufschreiben, die genau dann `true` (bzw. „ja") liefert, wenn die Eingabe zur entsprechenden Klasse gehört.

Schließlich mußt Du für jeden Zweig einen Ausdruck angeben, der für den Fall zuständig ist, daß die Eingabe in die entsprechende Klasse fällt.

8.2 Beispiel

Gefragt ist eine Funktion, die feststellt, ob bei einer gegebenen Temperatur (in °C) feststellt, ob dabei Wasser gefriert.

Dazu muß die Temperatur mit Null verglichen werden. Ob eine Zahl größer oder kleiner als Null ist, ist eine *Ja/Nein-Frage*. Viele Funktionen für Ja/Nein-Fragen sind in Scheme schon eingebaut:

```
(> 10 5) ⟹ true
(< 10 5) ⟹ false
(<= 5 5) ⟹ true
(<= 5 6) ⟹ true
(<= 6 5) ⟹ false
```

Damit steht `true` für „ja“ oder „wahr“ und `false` für „nein“ oder „falsch“.

Anmerkungen:

- In den HtDP-Modi von DrScheme werden die boolesche Werte als `true` und `false` ausgedruckt. Die Literale dazu sind ebenfalls `true` und `false`. Dies unterscheidet die HtDP-Modi von R^{5}RS-Scheme, wo `#t` für „true“ und `#f` für „false“ steht.

Zurück zur ursprünglichen Frage. Es handelt sich dabei um eine *Fallunterscheidung*: alle Zahlen, die Temperaturen darstellen, fallen in zwei Gruppen — die Temperaturen, bei denen Wasser gefriert, und diejenigen, bei denen Wasser nicht gefriert. Die Funktion soll eine Temperatur konsumieren und, je nachdem, in welche Gruppe sie fällt, `"gefriert"` oder `"gefriert nicht"` zurückgeben.

1. Die Datenanalyse besagt, daß bei der Eingabe eine Fallunterscheidung stattfindet, und zwar zwischen Zahlen, die kleiner oder gleich Null sind, und allen anderen Zahlen.

   Die Ausgabe ist eine Zeichenkette, und zwar `"gefriert"` oder `"gefriert nicht"`.

2. Die Funktion soll eine Zahl konsumieren und, wenn diese Zahl größer als 0 ist, `"gefriert"` zurückgeben, sonst `"gefriert nicht"`.

3. Vertrag:

```
;; wasser-gefriert-bei: zahl -> {"gefriert" oder "gefriert nicht"}
```

4. Das Funktionsgerüst ergibt (wie immer) sich aus dem Vertrag und der Aufgabenbeschreibung:

```
(define (wasser-gefriert-bei temperatur)
  ...)
```

5. Beispiele:

To test	(wasser-gefriert-bei -10)
Expected	"gefriert"

To test	(wasser-gefriert-bei 0)
Expected	"gefriert"

To test	(wasser-gefriert-bei 0.1)
Expected	"gefriert nicht"

To test	(wasser-gefriert-bei 1000)
Expected	"gefriert nicht"

6. Die Funktionsschablone ergibt sich aus der Fallunterscheidung in der Eingabe — da es zwei Fälle gibt, muß im Rumpf ein `cond` mit zwei Fällen stehen, jeweils mit einer Frage, die mit `true` beantwortet wird, falls die Eingabe in die entsprechende Gruppe gehört:

```
(define (wasser-gefriert-bei temperatur)
  (cond
    [(<= temperatur 0) ...]
    [(>  temperatur 0) ...]))
```

7. In diesem Fall ist es einfach, die Ellipsen in der Schablone zum Funktionsrumpf zu vervollständigen:

```
(define (wasser-gefriert-bei temperatur)
  (cond
    [(<= temperatur 0) "gefriert"]
    [(>  temperatur 0) "gefriert nicht"]))
```

8. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

Die fertige Lösung der Aufgabe:

```
;; für eine Temperatur "gefriert" zurückgeben, falls sie kleiner oder
;; gleich 0 ist, sonst "gefriert nicht"

;; wasser-gefriert-bei: zahl -> "gefriert" oder "gefriert nicht"

(define (wasser-gefriert-bei temperatur)
  (cond
    [(<= temperatur 0) "gefriert"]
    [(>  temperatur 0) "gefriert nicht"]))
```

To test	(wasser-gefriert-bei -10)
Expected	"gefriert"

To test	(wasser-gefriert-bei 0)
Expected	"gefriert"

To test	(wasser-gefriert-bei 0.1)
Expected	"gefriert nicht"

To test	(wasser-gefriert-bei 1000)
Expected	"gefriert nicht"

- Bei der Fallunterscheidung geht es in der rein funktionalen Programmierung im Gegensatz zur imperativen Programmierung nicht darum, abhängig vom Ausgang der Bedingung etwas „zu tun“. Stattdessen führen unterschiedliche (Gruppen von) Eingaben zu unterschiedlichen Rückgabewerten.

- `Cond` entspricht der Fallunterscheidung in der Mathematik, nur mit vertauschten Fragen und Antworten. In mathematischer Notation sähe die Funktion von oben zum Beispiel so aus:

$$g(t) := \begin{cases} \texttt{"gefriert"} & \text{falls } t \leq 0 \\ \texttt{"gefriert nicht"} & \text{falls } t > 0 \end{cases}$$

- In R^5RS-Scheme gibt es nur runde Klammern, keine eckigen. Bei `cond` allerdings hat sich gezeigt, daß Anfänger oft von den zwei aufeinanderfolgenden öffnenden runden Klammern bei den `cond`-Fällen verwirrt werden. Die eckigen Klammern erleichtern die Sache.

- In diesem Beispiel soll bewußt kein Boolean zurückgegeben werden. Dies vermeidet einerseits leichte Irritationen, weil Booleans dann an zwei unterschiedlichen Stellen der Aufgabe auftauchen würden. Andererseits erleichtert die Verwendung von Zeichenketten eine spätere Erweiterung; siehe Aufgabe 8.1.

8.3 Übungsaufgaben

Aufgabe 8.1 Schreibe eine Funktion `wasser-aggregatzustand`, die eine Temperatur konsumiert, und den dazugehörigen Aggregatzustand von Wasser zurückgibt, also `"fest"`, `"flüssig"` oder `"gasförmig"`.

Kapitel 9

Zusammengesetzte Daten

Realistische Programme müssen häufig Dinge darstellen, die aus mehreren Bestandteilen bestehen:

- Ein Schokokeks besteht aus einem Schokolade- und einem Keks-Anteil.
- Eine kartesische Koordinate in der Ebene besteht aus einer X- und Y-Komponente.
- Ein ausgiebiges Essen besteht aus Vorspeise, Hauptgang und Nachspeise.

Mit anderen Worten: mehrere Dinge werden *zu einem* zusammengesetzt. Eine andere Betrachtungsweise ist, daß ein einzelnes Ding mehrere Eigenschaften hat.

9.1 Konstruktionsanleitung

9.1.1 Funktionen, die zusammengesetzte Daten konsumieren

Datenanalyse Bevor Du eine Funktion entwickelst, die ein Problem löst, in dem zusammengesetzte Daten vorkommen, mußt Du Dir erst überlegen, wie diese Daten dargestellt werden sollen und dann jeweils eine entsprechende *Struktur-* und eine *Datendefinition* aufschreiben — es reicht nicht mehr aus, die in DrScheme eingebauten Datendefinitionen zu benutzen.

Falls also ein Objekt in der Aufgabenstellung aus mehreren — n — Bestandteilen besteht, benötigst Du eine Strukturdefinition mit n Bestandteilen. Überlege Dir einen Namen N für die Klasse der zusammengesetzten Objekte sowie jeweils einen Namen b_i für jedes Bestandteil und schreibe die entsprechende `define-struct`-Form auf:

```
(define-struct N (b1 ... bn))
```

Natürlich mußt Du Dir überlegen, zu welcher Klasse jedes der Bestandteile gehört.

Schreibe nun die Datendefinition in folgender Form auf:

```
;; Ein N ist eine Struktur (make-N x1 ... xn)
;; wobei ...
;; x1 ein ...,
;; ...
;; xn ein ...
;; ist.
```

Vertrag Für die Klassen der zusammengesetzten Objekte benutze jeweils den Namen der Strukturdefinition.

Funktionsgerüst Für die Namen der Parameter, die später zusammengesetzte Objekte aufnehmen sollen, hänge ein `ein-` oder `eine-` vor den jeweiligen Namen, damit es keine Mißverständnisse zwischen dem Namen der Strukturdefinition und dem Namen im Funktionsgerüst geben kann.

Testfälle Um Testfälle aufzustellen, benutze die `make-`N-Funktion, um zusammengesetzte Objekte zu erzeugen.

Funktionsschablone Die meisten Funktionen, die zusammengesetzte Daten konsumieren, ermitteln ihr Resultat aus den Komponenten. Deswegen kommen die Komponenten meist im Rumpf der fertigen Funktion vor. Darum gehören die entsprechenden Aufrufe der Selektoren auch zur Funktionsschablone.

9.1.2 Funktionen, die zusammengesetzte Daten zurückliefern

Datenanalyse, Vertrag, Testfälle siehe oben.

Funktionsschablone Der Rumpf einer Funktion, die einen zusammengesetzten Wert zurückgeben soll, muß aus einem Aufruf des entsprechenden Konstruktors bestehen.

Funktionsrumpf Schreibe für jede der Eingaben des Konstruktors einen Ausdruck auf, der als Ergebnis den entsprechenden Bestandteil der Struktur liefert.

9.2 Beispiel

Gefragt ist eine Funktion, die das Gesamtgewicht eines Schokokekses ausrechnet.

1. Die Datenanalyse ergibt, daß ein Schokokeks aus einem Schoko- und einem Keks-Anteil besteht — damit sind zusammengesetzte Daten und eine entsprechende Strukturdefinition im Spiel:

   ```
   (define-struct schokokeks (schoko keks))
   ```

   Die dazugehörige Datendefinition sieht so aus:

   ```
   ;; Ein schokokeks ist eine Struktur (make-schokokeks S K)
   ;; wobei
   ;; S eine Zahl,
   ;; K eine Zahl
   ;; ist.
   ```

   Diese Strukturdefinition definiert eine Reihe von Funktionen:

   Konstruktor Die Funktion `make-schokokeks` mit folgendem Vertrag:

   ```
   ;; make-schokokeks: zahl zahl -> schokokeks
   ```

   Diese Funktion konsumiert also zwei Zahlen — das Gewicht des Schoko- und das Gewicht des Keks-Anteils, und liefert als Ergebnis ein `schokokeks`-Objekt.

   Selektoren Zwei Funktion extrahieren jeweils den Schoko- und den Keks-Anteil aus einem `schokokeks`-Objekt:

```
;; schokokeks-schoko: schokokeks -> zahl
;; schokokeks-keks: schokokeks -> zahl
```

Zwei Beispiele verdeutlichen die Funktionsweise:

```
(schokokeks-schoko (make-schokokeks 5 10)) ⟹ 5
(schokokeks-keks (make-schokokeks 5 10)) ⟹ 10
```

Prädikat Die Funktion `schokokeks?` unterscheidet `schokokeks`-Objekte von anderen Objekten:

```
;; schokokeks? : objekt -> bool
(schokokeks? 5) ⟹ false
(schokokeks? "Ich bin kein Schokokeks") ⟹ false
(schokokeks? "Ich bin ein Schokokeks") ⟹ false
(schokokeks? (make-schokokeks 5 10)) ⟹ true
```

2. Die Funktion soll ein `schokokeks`-Objekt konsumieren und dessen Gesamtgewicht zurückgeben.

3. Vertrag:

```
;; schokokeks-gewicht: schokokeks -> zahl
```

4. Das Funktionsgerüst ergibt sich aus dem Vertrag und der Aufgabenbeschreibung:

```
(define (schokokeks-gewicht ein-schokokeks)
  ...)
```

5. Beispiele:

To test	(schokokeks-gewicht (make-schokokeks 5 10))
Expected	15

To test	(schokokeks-gewicht (make-schokokeks 0 10))
Expected	10

To test	(schokokeks-gewicht (make-schokokeks 7 0))
Expected	7

6. Hier ist die Funktionsschablone:

```
(define (schokokeks-gewicht ein-schokokeks)
  ... (schokokeks-schoko ein-schokokeks) ...
  ... (schokokeks-keks ein-schokokeks) ...)
```

7. Von da ist es nicht mehr weit zur fertigen Definition:

```
(define (schokokeks-gewicht ein-schokokeks)
  (+ (schokokeks-schoko ein-schokokeks)
     (schokokeks-keks ein-schokokeks)))
```

8. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

Anmerkungen:

- Anfänger haben erfahrungsgemäß oft Schwierigkeiten mit der Idee „mehrere Dinge werden zu einem“. Darum empfiehlt es sich, die Idee erst einmal an einer möglichst langen Reihe von Beispielen zügig mental durchzuspielen. (Siehe den Aufgabenabschnitt.)
- Anfänger — und auch besonders Schüler, die schon einmal Java oder C programmiert haben — haben Schwierigkeiten mit der Idee der Selektorfunktion. Sie haben das Gefühl, es wäre etwas „doppelt“ bei einem Ausdruck wie:

  ```
  (schokokeks-schoko ein-schokokeks)
  ```

  Aus diesem Grund ist es sinnvoll, einen Parameter für Schokokekse auch `ein-schokokeks` und nicht einfach `schokokeks` zu benennen, um den Grad der Verwirrung etwas zu senken.

  In jeden Fall ist es auch hier entscheidend, am Anfang noch einmal besonders darauf zu bestehen, daß die Schüler die Funktionsschablone stets niederschreiben, bevor sie damit fortfahren, eine Aufgabe zu lösen..

9.3 Beispiel

Gefragt ist eine Funktion, die aus einem Schokokeks einen noch schmachkhafteren Schokokeks macht — natürlich dadurch, daß sie seinen Schoko-Anteil verdoppelt.

1. Die Datenanalyse ergibt, daß die gefragte Funktion einen Schokokeks konsumiert und auch wieder einen zurückliefert. Die Daten- und Strukturdefinitionen wurden schon für vorherige Aufgaben aufgeschrieben. Es kommen beide Konstruktionsanleitungen für zusammengesetzte Daten zum Einsatz.
2. Der Vertrag der Funktion sieht so aus:

   ```
   ;; leckerer-schokokeks: schokokeks -> schokokeks
   ```

3. Das Funktionsgerüst folgt daraus — wie immer — direkt:

   ```
   (define (leckerer-schokokeks ein-schokokeks)
     ...)
   ```

4. Beispiele:

   ```
   To test   (leckerer-schokokeks (make-schokokeks 5 10))
   Expected  %
   (make-schokokeks 10 10)
   To test   (leckerer-schokokeks (make-schokokeks 0 10))
   Expected  %
   (make-schokokeks 0 10
   To test   (leckerer-schokokeks (make-schokokeks 7 0))
   Expected  %
   (make-schokokeks 14 0)
   ```

5. Für die Funktionsschablone werden die Konstruktionsanleitungen für Funktionen kombiniert, die zusammengesetzte Daten konsumieren, und solche, die zusammengesetzte Daten zurückgeben sollen:

```
(define (leckerer-schokokeks ein-schokokeks)
  (make-schokokeks
    ... (schokokeks-schoko ein-schokokeks) ...
    ... (schokokeks-keks ein-schokokeks) ...))
```

6. Für die Konstruktion des Rumpfes müssen Ausdrücke aufgeschrieben werden — natürlich unter Verwendung der Bestandteile der Eingabe, die schon in der Funktionsschablone stehen:

   (a) Der Schoko-Anteil des leckereren Schokokekses.

   (b) Der Keks-Anteil des leckereren Schokokekses.

   Der erste Ausdrück muß den doppelten Schoko-Anteil der Eingabe liefern:

```
(define (leckerer-schokokeks ein-schokokeks)
  (make-schokokeks
    (* 2 (schokokeks-schoko ein-schokokeks))
    ... (schokokeks-keks ein-schokokeks) ...))
```

   Der zweite Ausdruck muß den Keks-Anteil der Eingabe unverändert liefern:

```
(define (leckerer-schokokeks ein-schokokeks)
  (make-schokokeks
    (* 2 (schokokeks-schoko ein-schokokeks))
    (schokokeks-keks ein-schokokeks)))
```

9.4 Aufgaben

Aufgabe 9.1

- Eine Uhrzeit hat wieviele Komponenten?
- Ein Datum hat wieviele Komponenten?
- Ein Erdbeerkuchen hat wieviele Komponenten?
- usw.

Schreibe jeweils passende Daten- und Strukturdefinitionen.

Aufgabe 9.2 Betrachte folgende Strukturdefinitionen:

```
(define-struct film (titel jahr))

(define-struct cd (name interpret/in))

(define-struct mineralwasser (natrium calcium magnesium))

(define-struct brief (absender addressat))
```

1. Wie heißen jeweils der Konstruktor, das Prädikat und die Selektoren der Strukturdefinitionen?
2. Schreibe Datendefinitionen, die zu den obigen Strukturdefinitionen passen.
3. Schreibe Funktionsschablonen für Funktionen, die Objekte der Strukturdefinitionen konsumieren.

Aufgabe 9.3 Schreibe Struktur- und Daten-Definitionen für

1. kartesiche Koordinaten in der Ebene,
2. Essen aus Vorspeise, Hauptgang und Nachspeise,
3. Uhrzeiten und Datumsangaben.

Aufgabe 9.4 Mit Hilfe der Definitionen aus der vorigen Aufgabe:

1. Schreibe eine Funktion, die eine kartesiche Koordinatenangabe konsumiert, und deren Abstand vom Ursprung berechnet.
2. Schreibe eine Funktion, die eine Uhrzeit konsumiert, und die zugehörige Anzahl der Minuten seit Mitternacht zurückgibt.
3. Schreibe eine Funktion, die eine Datumsangabe konsumiert, und die Anzahl der Tage seit Jahresbeginn zurückgibt.

Aufgabe 9.5

1. Schreibe eine Struktur- und eine Datendefinition für Schlangen auf; eine Schlange wird dabei durch ihren Namen, ihr Gewicht (in kg) und ihre Lieblingsnahrung beschrieben.
2. Schreibe eine Funktion `schlange-duenn?`, die eine Schlange konsumiert, und `true` liefert, wenn die Schlange weniger als 5kg wiegt, sonst `false`.
3. Schreibe eine Funktion `fuettere-schlange`, die eine Schlange konsumiert und eine Schlange mit dem gleichen Namen und der gleichen Lieblingsnahrung zurückliefert, die aber um 2kg schwerer ist.

Kapitel 10

Gemischte Daten

Manche Funktionen müssen Daten verarbeiten können, die zu einer von mehreren Typen von Klassen gehören. Zum Beispiel könnte es darum gehen, den Flächeninhalt einer geometrischen Form zu berechnen. Eine geometrische Form könnte dabei z.B. ein Kreis, ein Rechteck etc. sein — die entsprechende Funktion muß in der Lage sein, alle diese Objekte zu konsumieren, mithin *gemischte Daten* zu verarbeiten.

10.1 Konstruktionsanleitung

Datenanalyse Falls die Aufgabenstellung verschiedene Klassen von Daten behandelt, die gemeinsam behandelt werden, handelt es sich um gemischte Daten. Damit die noch zu entwickelnde Datendefinition sinnvoll wird, mußt Du die verschiedenen Klassen so wählen, daß sie eindeutig unterscheidbar sind — Du mußt also für ein gegebenes Objekt immer sagen könnten, zu welcher Klasse es gehört.

Denke Dir einen Namen für die Vereinigung der verschiedenen Klassen aus, sagen wir X. Schreibe dann eine Datendefinition in folgender Form auf:

```
;; Ein X ist ...
;; - ein A,
;; ...
;; oder ein Z.
```

Für jede Klasse mußt Du dann eine eigene, separate Datenanalyse durchführen.

Vertrag Für die Klassen der Objekte aus gemischten Daten benutze den gewählten Namen für die Vereinigung.

Hilfsfunktionen Schreibe nun für jede Klasse eine separate Funktion, welche das Problem *ausschließlich für Werte dieser Klasse löst.* Folge dabei der jeweiligen Konstruktionsanleitung.

Funktionsgerüst Für die Namen der Parameter, die später zusammengesetzte Objekte aufnehmen sollen, hänge ein `ein-` oder `eine-` vor den jeweiligen Namen, damit es keine Mißverständnisse zwischen dem Namen der Vereinigung und dem Namen im Funktionsgerüst geben kann.

Funktionsschablone Eine Funktion, die gemischte Daten konsumiert, muß meistens zwischen den verschiedenen Klassen unterscheiden, zu denen die Eingabe gehört. Deswegen besteht der Rumpf einer solchen Funktion in diesem Fall aus einem `cond`-Ausdruck:

```
(define (f ein-X)
  (cond
    [... ...]
    ...
    [... ...]))
```

Der `cond`-Ausdruck mußt genauso viele Zweige enthalten, wie es Klassen in der Datendefinition gibt.

Dann schreibst Du die Bedingungen dazu: Jede Bedigung muß prüfen, ob die Eingabe zu der ensprechenden Klasse der Datendefinition gehört.

Funktionsrumpf Für jeden Zweig des `cond`-Ausdrucks kannst Du nun die Hilfsfunktion aufrufen, die Du eigens dafür vorher geschrieben hast.

10.2 Beispiel

In Kapitel 9 ging es um das Gewicht eines Schokokekses. Für viele ist das Gewicht ja nicht nur bei Schokokeksen relevant, sondern auch bei, sagen wir, Schoko-Milchshakes. Exakt nach dem gleichen Muster wie bei der Funktion `schokokeks-gewicht` aus diesem Kapitel entsteht auch die Strukturdefinition

```
(define-struct schoko-milchshake (schoko milch))
```

sowie die Datendefinition

```
;; Ein schoko-milchshake ist eine Struktur (make-schoko-milchshake M S)
;; wobei
;; M eine Zahl
;; S eine Zahl,
;; ist.
```

und eine passende Funktion

```
;; schoko-milchshake-gewicht: schoko-milchshake -> zahl

(define (schoko-milchshake-gewicht ein-schoko-milchshake)
  (+ (schoko-milchshake-schoko ein-schoko-milchshake)
     (schoko-milchshake-keks ein-schoko-milchshake)))
```

Gefragt ist nun eine Funktion, die für ein Grundnahrungsmittel — also einen Schokokeks oder einen Schoko-Milchshake — das Gewicht berechnet.

1. Es ist eine Funktion gefragt, die sowohl Schokokekse als auch Schoko-Milchshakes (als *Grundnahrungsmittel*) akzeptiert. Es handelt sich also um gemischte Daten, und die Datendefinition dazu ist die folgende:

   ```
   ;; Ein grundnahrungsmittel ist
   ;; - ein schokokeks oder
   ;; - ein schoko-milchshake
   ```

2. Vertrag:

   ```
   ;; grundnahrungsmittel-gewicht: grundnahrungsmittel -> zahl
   ```

3. Das Funktionsgerüst entsteht wieder aus dem Vertrag und der Aufgabenbeschreibung:

```
(define (grundnahrungsmittel-gewicht ein-grundnahrungsmittel)
  ...)
```

4. Beispiele:

To test	(grundnahrungsmittel-gewicht (make-schokokeks 5 10))
Expected	15

To test	(grundnahrungsmittel-gewicht (make-schokokeks 0 10))
Expected	10

To test	(grundnahrungsmittel-gewicht (make-schokokeks 7 0))
Expected	7

To test	(grundnahrungsmittel-gewicht (make-schoko-milchshake 12 3))
Expected	15

To test	(grundnahrungsmittel-gewicht (make-schoko-milchshake 0 7))
Expected	7

To test	(grundnahrungsmittel-gewicht (make-schoko-milchshake 8 0))
Expected	8

5. Hier ist die Funktionsschablone — da es für ein Grundnahrungsmittel *zwei* Möglichkeiten gibt, muß in der Schablone ein `cond`-Ausdruck mit ebenfalls *zwei* Zweigen stehen. Erster Schritt:

```
(define (grundnahrungsmittel-gewicht ein-grundnahrungsmittel)
  (cond
    [... ...]
    [... ...]))
```

Im zweiten Schritt müssen noch die Fragen ergänzt werden — glücklicherweise haben die Strukturdefinitionen für Schokokekse und Schoko-Milchshakes jeweils auch Prädikate `schokokeks?` und `schoko-milchshake?` zur Verfügung:

```
(define (grundnahrungsmittel-gewicht ein-grundnahrungsmittel)
  (cond
    [(schokokeks? ein-grundnahrungsmittel) ...]
    [(schoko-milchshake? ein-grundnahrungsmittel) ...]))
```

Bei den langen Zeilen empfehlen sich wahrscheinlich Zeilenumbrüche:

```
(define (grundnahrungsmittel-gewicht ein-grundnahrungsmittel)
  (cond
    [(schokokeks? ein-grundnahrungsmittel)
     ...]
    [(schoko-milchshake? ein-grundnahrungsmittel)
     ...]))
```

6. Der Rumpf ist jetzt mit Hilfe der vorher definierten Funktionen `schokokeks-gewicht` und `schoko-milchshake-gewicht` einfach auszufüllen:

```
(define (grundnahrungsmittel-gewicht ein-grundnahrungsmittel)
  (cond
    [(schokokeks? ein-grundnahrungsmittel)
     (schokokeks-gewicht ein-grundnahrungsmittel)]
    [(schoko-milchshake? ein-grundnahrungsmittel)
     (schoko-milchshake-gewicht ein-grundnahrungsmittel)]))
```

10.3 Beispiel

Gefragt ist eine Funktion, die geometrische Formen — Kreise und Rechtecke — jeweils den Flächeninhalt ausrechnet.

1. Die Datenanalyse muß mit etwas vagen Informationen seitens der Aufgabenstellung auskommen: Es geht um Kreise und Rechtecke.

   Kreise zuerst: Ein Kreis ist charakterisiert durch seinen Mittelpunkt und seinen Radius. Hier eine passende Strukturdefinition:

   ```
   (define-struct kreis (mittelpunkt radius))
   ```

   Hier die dazugehörige Datendefinition:

   ```
   ;; Ein kreis ist eine Struktur (make-kreis M R), wobei
   ;; - M ein posn,
   ;; - R eine Zahl ist.
   ```

   Hier ist die Strukturdefinition für Rechtecke:

   ```
   (define-struct rechteck (unten-links breite hoehe))
   ```

   ... und hier die dazugehörige Datendefinition:

   ```
   ;; Ein rechteck ist eine Struktur (make-rechteck UL B H), wobei
   ;; - UL ein posn,
   ;; - und B und H Zahlen sind.
   ```

   Für jede einzelne von ihnen ist es einfach, die passende Funktion zu schreiben, die den Flächeninhalt berechnet. Zunächst der Kreis:

   ```
   (define (kreis-flaeche ein-kreis)
     (* 2 pi
        (quadrat (kreis-radius ein-kreis))))

   (define (quadrat zahl)
     (* zahl zahl))
   ```

   (Das Quadrat einer Zahl ist eine häufig vorkommende Zwischengröße, die nach Mantra 1 eine eigene Funktion verdient.) Als nächstes das Rechteck:

   ```
   (define (rechteck-flaeche rechteck)
     (* (rechteck-breite rechteck)
        (rechteck-hoehe rechteck)))
   ```

   Nun wäre es schön, wenn eine einzige Funktion (z.B. mit dem einfachen Namen `form-flaeche`) sowohl Kreise als auch Rechtecke konsumieren kann und für beide jeweils die richtige Fläche berechnet. Dazu gehört eine Datendefinition:

```
;; Eine Form ist:
;; - ein Kreis
;; - oder ein Rechteck
```

Der Vertrag

```
;; form-flache : form -> zahl
```

Hier das Funktionsgerüst:

```
(define (form-flaeche eine-form)
  ...)
```

Da es sich bei der Eingabe — `eine-form` — um eine Form und damit um eine von zwei Sorten Formen handelt, enthält die Funktionsschablone einen `cond`-Ausdruck mit zwei Zweigen: es

```
(define (form-flaeche eine-form)
  (cond
    [... ...]
    [... ...]))
```

Es gilt jetzt Fragen für die beiden `cond`-Zweige aufzuschreiben, welche die beiden Sorten identifizieren:

```
(define (form-flaeche eine-form)
  (cond
    [(kreis? eine-form)
     ...]
    [(rechteck? eine-form)
     ...]))
```

Zuletzt fehlen nur noch die Antworten für beiden Zweige:

```
(define (form-flaeche eine-form)
  (cond
    [(kreis? eine-form)
     (kreis-flaeche eine-form)]
    [(rechteck? eine-form)
     (rechteck-flaeche eine-form)]))
```

10.4 Aufgaben

Aufgabe 10.1 Werte die folgenden Ausdrücke von Hand aus:

```
(number? (make-schokokeks 5 10))
(schokokeks? 5)
(schokokeks? (make-schokokeks 5 10))
(schokokeks? (make-schoko-milchshake 12 3))
(schoko-milchshake? (make-schokokeks 5 10))
```

Aufgabe 10.2 In manchen Teilen Deutschlands gilt auch Bier als Grundnahrungsmittel — je nach Bier mit bestimmten Kohlensäure- und Alkoholanteilen.

- Schreibe eine Strukturdefinition und eine Datendefinition für Bier.
- Schreibe eine erweiterte Datendefinition für Grundnahrungsmittel.
- Schreibe eine Funktion `grundnahrungsmittel-schoko`, die ein Grundnahrungsmittel konsumiert, und dessen Schokoladen-Anteil zurückgibt.
- Ist es möglich, die Funktion `grundnahrungsmittel-gewicht` so zu ändern, daß sie auch mit der neuen Datendefinition funktioniert?

Aufgabe 10.3

- Schreibe eine Struktur- und eine Datendefinition für Quadrate auf.
- Erweitere die Datendefinition für Formen um Quadrate.
- Schreibe eine Funktion `form-umfang`, die den Umfang einer Form berechnet.
- Erweitere die Definition von `form-flaeche`, so daß sie für die neue Datendefinition funktioniert.

Aufgabe 10.4

1. Schreibe eine Struktur- und eine Datendefinition für Ameise. Eine Ameise wird durch ihr Gewicht (in g) und ihr Alter (in Jahren) beschrieben.
2. Schreibe eine Struktur und eine Datendefinition für Nacktmulle auf. Ein Nacktmull wird durch sein Gewicht (in g) und die Länge seiner Schneidezähne (in cm) beschrieben.
3. Schreibe eine Funktion `fuettere-ameise`, die eine Ameise konsumiert, und eine um 1g schwerere Ameise zurückgibt.
4. Schreibe eine Funktion `fuettere-nacktmull`, die einen Nacktmull konsumiert und einen um 100g schwereren Nacktmull zurückgibt.
5. Schreibe eine Datendefinition für *Tiere*, wobei ein Tier eine Schlange (siehe Aufgabe 9.5), eine Ameise oder ein Nacktmull sein kann.
6. Schreibe eine Funktion `fuettere-tier`, die ein Tier füttert.

Kapitel 11

Einführung in das Arbeiten mit Listen

Listen sind zusammengesetzte Datenstrukturen, bei denen, anders als bei Strukturen, die Länge nicht fest bestimmt sind ist. Listen sind ein aus dem Alltag bekanntest Konzept — es gibt Einkaufslisten, Weihnachts-Wunschlisten, schwarze Listen, Gästelisten bei Feiern undsoweiter undsoweiter. Sie alle haben gemeinsam, daß die Länge in der Regel nicht im vorhinein bekannt ist. In Computerprogrammen sind Listen mit Abstand die nützlichste Datenstruktur; in Scheme-Programmen sind sie allgegenwärtig.

Bei Listen handelt es sich um *selbst-referentielle* (oder *induktive*) Datenstrukturen. Damit sind sie das ideale Einstiegsmittel in die Programmiertechnik Rekursion. (Und nicht etwa Zahlen, deren selbst-referentielle Struktur nicht der Intuition von Schülern entspricht.) Da Selbstreferentialität für viele etwas gewöhnungsbedürftig ist, empfiehlt sich ein sanfter Einstieg mit Listen begrenzter Länge — davon handelt dieses Kapitel.

11.1 Listen konstruieren

Listen fangen — wie im richtigen Leben — immer kurz an und werden dann länger. Die kürzeste Liste ist die *leere Liste* und heißt in Scheme

```
empty
```

Eine Liste mit einem Element wird aus der leeren Liste und dem Element erzeugt:

```
(cons "Tuttifrutti" empty)
```

Das `cons` steht für „construct“ oder „konstruiere“. Der folgende Ausdruck konstruiert eine Liste mit zwei Elementen:

```
(cons "Waldmeister" (cons "Tuttifrutti" empty))
```

Mit anderen Worten hängt `cons` an eine Liste ein Element *vorn* an — das unterscheidet Listen in Scheme von den Listen aus dem Alltag, bei denen neue Elemente meist hinten angehängt werden.

Drei Elemente gehen auch:

```
(cons "Eierlikör" (cons "Waldmeister" (cons "Tuttifrutti" empty)))
```

Das Prinzip ist immer das gleiche — an den Beispielen läßt sich darum auch ein vorläufiger Vertrag für `cons` ablesen:

```
;; cons : zeichenkette liste -> liste
```

Es ist etwas schwierig, in einem Ausdruck wie

```
(cons "Eierlikör" (cons "Waldmeister" (cons "Tuttifrutti" empty)))
```

die Liste zu erkennen, wegen der vielen Klammern und `cons`. Darum ist es hier an der Zeit, eine Sprachebene höher zu schalten, zu „Anfänger mit Listen-Abkürzungen". An den Listen selbst hat sich nichts geändert, aber sie werden jetzt anders ausgedruckt. Der obige Ausdruck wird dann ausgedruckt als

```
(list "Eierlikör" "Waldmeister" "Tuttifrutti")
```

Tatsächlich ist `list` auch eine eingebaute Funktion, die Programme verwenden können —

```
(list "Eierlikör" "Waldmeister" "Tuttifrutti")
```

ist ein Scheme-Ausdruck und hat die gleiche Bedeutung wie

```
(cons "Eierlikör" (cons "Waldmeister" (cons "Tuttifrutti" empty)))
```

`List` ist also eine Abkürzung für mehrere verschachtelte Anwendungen von `cons`.

Es empfiehlt sich, den Umgang mit `list` und den Zusammenhang mit `cons` etwas zu trainieren:

```
(cons "Eierlikör" (list "Waldmeister" "Tuttifrutti"))
  ⟹ (list "Eierlikör" "Waldmeister" "Tuttifrutti")
(cons "Eierlikör" (list)) ⟹ (list "Eierlikör"")
```

Die Listenelemente sind nicht auf Zeichenketten beschränkt. Es gibt auch Listen aus Zahlen:

```
(cons 1 (cons 2 (cons 3 empty)))
```

Eine Liste kann auch Elemente verschiedener Arten enthalten:

```
(cons 1 (cons "Tuttifrutti" (cons 2 (cons "Waldmeister empty))))
```

Damit muß der Vertrag für `cons` revidiert werden:

```
;; cons : objekt liste -> liste
```

11.2 Mit Listen rechnen

Eine mögliche Liste könnte gerade aus den Preisen der drei Artikel einer Einkaufsliste bestehen:

```
(list 1.99 2.99 0.50)
```

Eine typische Aufgabe ist es, die Elemente der Liste zu addieren um den Gesamtpreis der Einkaufsladung zu ermitteln. Es geht also um eine Funktion, die eine Liste mit drei Elementen — allesamt Zahlen — konsumiert und eine Zahl zurückliefert.

Hier ist der Vertrag:

```
;; addiere-einkaufsliste-3 : liste-mit-3-zahlen -> zahl
```

Damit steht auch das Funktionsgerüst schnell:

```
(define (addiere-einkaufsliste-3 l)
  ...)
```

Beispiele sind schnell aufgeschrieben:

To test	(addiere-einkaufsliste-3 (list 1.99 2.99 0.50))
Expected	5.48

To test	(addiere-einkaufsliste-3 (list 0.99 50.33 99.99))
Expected	151.31

To test	(addiere-einkaufsliste-3 (list 30.00 5000.00 0.01))
Expected	5030.01

Hier ist der erste Anlauf für eine Funktionsschablone:

```
(define (addiere-einkaufsliste-3 l)
  ... erstes Element von l ...
  ... zweites Element von l ...
  ... drittes Element von l)) ...
```

Um diese Funktionsschablone zu vervollständigen, fehlen Ausdrücke, die aus einer Liste die einzelnen Elemente herausholen — so etwas wie die Selektoren für die Strukturen. Dazu gibt es zwei Funktionen `first` und `rest`. Ihre Funktionsweise läßt sich am einfachsten anhand einer Reihe von Beispielen verstehen:

```
(first (list 1.99 2.99 0.50)) ⟹ 1.99
(first (list 0.99 50.33 99.99)) ⟹ 0.99
(first (list 30.00 5000.00 0.01)) ⟹ 30

(rest (list 1.99 2.99 0.50)) ⟹ (list 2.99 0.50)
(rest (list 0.99 50.33 99.99)) ⟹ (list 50.33 99.99)
(rest (list 30.00 5000.00 0.01)) ⟹ (list 5000.00 0.01)
```

Mit anderen Worten: `first` konsumiert eine Liste (die nicht `empty` sein darf) und liefert deren erstes Element. Die `rest`-Funktion konsumiert ebenfalls eine nicht-leere Liste und liefert wieder eine Liste zurück, bei der gegenüber der Eingabeliste das erste Element fehlt.

Die Funktionsweise von `rest` mag manchen etwas merkwürdig erscheinen — sie ist vielleicht deutlicher zu erkennen, wenn die Eingabelisten nicht mit `list` sondern mit direkt mit `cons` konstruiert werden und DrScheme noch einmal vorübergehend auf die Anfänger-Sprachebene ohne Listenabkürzungen umgeschaltet wird:

```
(first (cons 1.99 (cons 2.99 (cons 0.50 empty)))) ⟹ 1.99
(first (cons 0.99 (cons 50.33 (cons 99.99 empty)))) ⟹ 0.99
(first (cons 30.00 (cons 5000.00 (cons 0.01 empty)))) ⟹ 30

(rest (cons 1.99 (cons 2.99 (cons 0.50 empty)))) ⟹ (cons 2.99 (cons 0.50 empty))
(rest (cons 0.99 (cons 50.33 (cons 99.99 empty)))) ⟹ (cons 50.33 (cons 99.99 empty))
(rest (cons 30.00 (cons 5000.00 (cons 0.01 empty)))) ⟹ (cons 5000.00 (cons 0.01 empty))
```

Es gelten also folgende Regeln:

```
(first (cons a b)) ⟹ a
(rest (cons a b)) ⟹ b
```

Mit Kombinationen von `first` und `rest` läßt sich jedes beliebige Element aus einer Liste extrahieren:

```
(first (list 1.99 2.99 0.50)) ⟹ 1.99
(first (rest (list 1.99 2.99 0.50))) ⟹ 2.99
(first (rest (rest (list 1.99 2.99 0.50)))) ⟹ 0.5
```

11.3 Anmerkungen

- Obwohl Liste natürlich beliebig viele Elemente enthalten können, empfiehlt es sich, erst einmal mit Listen fester Länge „aufzuwärmen“ — dadurch kann eine zu steile Lernkurve bei den rekursiven Funktionen vermieden werden.
- In Standard-Scheme haben `first` und `rest` die historischen Namen `car` und `cdr`, die sich auf Befehlsnamen auf der IBM 704 beziehen — für Programmieranfänger ergeben diese Abkürzungen keinen Sinn.
- In Standard-Scheme ist der Ausdruck für die leere Liste `'()` — damit wird das Sprachelement `quote` verwendet, daß zu diesem Zeitpunkt in der Ausbildung noch schwer zu erklären ist. Darum gibt es in den Lehrsprachen `empty`.
- In Standard-Scheme heißt `empty?` `null?` und `cons?` `pair?`.
- In Standard-Scheme kann `cons` auch für die Bildung von *Paaren* verwendet werden, die keine Listen sind. Z.B. ist `(cons 1 2)` möglich. Solche Ausdrücke sind in den Lehrsprachen unzulässig.

11.4 Aufgaben

Aufgabe 11.1 Sei l die Liste

```
(cons "Charlie Chaplin" (cons "Orson Welles" (cons "Dorothy Parker" empty)))
```

Was ist jeweils der Wert der folgenden Ausdrücke?

- `(rest` *l*`)`
- `(first (rest` *l*`))`
- `(first (rest (rest` *l*`)))`
- `(rest (rest (rest` *l*`)))`

Aufgabe 11.2 Schreibe drei verschiedene Ausdrücke auf, die als Wert

```
(cons "Charlie Chaplin" (cons "Orson Welles" (cons "Dorothy Parker" empty)))
```

haben.

Aufgabe 11.3

- Entwickle eine Funktion, die eine Liste mit zwei Zahlen konsumiert und deren Summe zurückliefert.
- Entwickle eine Funktion, die eine Liste mit vier Zahlen konsumiert und deren Summe zurückliefert.
- Entwickle eine Funktion, die eine Liste mit vier Zahlen konsumiert und deren Produkt zurückliefert.

Aufgabe 11.4 Nimm an, daß eine Liste mit drei Zahlen eine Koordinatenangabe im dreidimensionalen Raum ist. Entwickle eine Funktion `abstand-zum-ursprung`, die eine solche Koordinatenangabe konsumiert und deren Abstand zum Ursprung zurückliefert.

Aufgabe 11.5 Welche der folgenden Ausdrücke sind sinnlos? Warum?

- `(first empty)`

- `(first (cons 1 empty))`
- `(first (cons 1 2))`
- `(first (rest (cons 1 empty)))`
- `(rest (first (cons 1 empty)))`
- `(rest (rest (cons 1 empty)))`
- `(rest (rest (cons 1 (cons 2 empty))))`
- `(rest empty)`

Kapitel 12

Listen

12.1 Konstruktionsanleitung

Datenanalyse Wenn in der Aufgabenstellung Listen vorkommen — also Daten, die sich in einer Reihe aus gleichartigen Bestandteilen zusammensetzen und beliebige Länge haben können, dann sind Listen im Spiel. Zu Listen gehört eine selbstreferentielle Datendefinition — hier für Listen von Zeichenketten:

```
;; Eine Liste von Zeichenketten ist:
;; - die leere Liste
;; - oder ein cons aus einer Zeichenkette und einer Liste von Zeichenketten
```

(Es empfiehlt sich einen Pfeil zwischen den beiden Vorkommen der Phrase „Liste von Symbolen“ zu zeichnen.)

Vertrag Für die Klassen der Listen benutze die Notation `list` (s), wobei s die Klasse der Elemente der Liste ist.

Funktionsschablone Eine Funktion, die Listen konsumiert, hat als Schablone eine Kombination der Schablonen für gemischte und zusammengesetzte Daten:

```
(define (f eine-liste)
  (cond
    [(empty? eine-liste)
     ...]
    [(cons? eine-liste)
     ... (first eine-liste) ...
     ... (f (rest eine-liste)) ...]))
```

Funktionsrumpf Fülle zunächst den ersten `cond`-Zweig für die leere Liste aus. Dann überlege Dir, wie Du aus dem gewüschten Rückgabewert für den Rest der Liste und dem ersten Element den gewüschten Rückgabewert für die gesamte Liste konstruieren kannst. Benutze dazu Beispiele. Denke *nicht* darüber nach, *wie* der Rückgabewert für den Rest der Liste berechnet wird.

12.2 Hinführung

In diesem Kapitel geht es um Listen mit variabler Länge. Bei einem Eis zum Beispiel kann es sein, daß vorher noch gar nicht klar ist, wieviele Kugeln dazugehören sollen:

```
empty
(cons "Tuttifrutti" empty)
(cons "Waldmeister" (cons "Tuttifrutti" empty))
(cons "Eierlikör" (cons "Waldmeister" (cons "Tuttifrutti" empty)))
(cons "Stracciatella"
      (cons "Eierlikör"
            (cons "Waldmeister"
                  (cons "Tuttifrutti" empty))))
```

Es ist nicht möglich, die Datendefinition für Listen mit den bisher gelernten Mitteln aufzuschreiben oder Funktionen dafür zu entwickeln: Eine Funktion, die eine Liste mit Eiskugeln konsumiert, muß für null, eine, zwei, drei, aber auch für fünfhundert Kugeln funktionieren. Genauso muß die Datendefinition für Listen von Zeichenketten für Listen von Zeichenketten beliebiger Länge funktionieren.

Bei der Datendefinition wird aus den ersten beiden Zeilen klar, daß es sich um gemischte Daten handelt:

```
;; Eine Liste von Zeichenketten ist:
;; - die leere Liste
;; - oder ein cons aus einer Zeichenkette und ?
```

Die jeweils zweiten Argumente von `cons` haben allesamt gemeinsam, daß es sich ebenfalls um Listen von Zeichenketten handelt. Die vollständige Datendefinition lautet also folgendermaßen:

```
;; Eine Liste von Zeichenketten ist:
;; - die leere Liste
;; - oder ein cons aus einer Zeichenkette und eine Liste von Zeichenketten
```

Damit sind Listen zu allererst gemischte Daten mit zwei Fällen. Damit gilt die folgende Funktionsschablone:

```
(define (f eine-liste)
  (cond
    [(empty? eine-liste)
     ...]
    [(cons? eine-liste)
     ...]))
```

Für den zweiten Fall handelt es sich außerdem um zusammengesetzte Daten. Damit ist es möglich, die Funktionsschablone zu erweitern:

```
(define (f eine-liste)
  (cond
    [(empty? eine-liste)
     ...]
    [(cons? eine-liste)
     ... (first eine-liste) ...
     ... (rest eine-liste) ...]))
```

Tatsächlich ist es möglich, die Funktionsschablone noch weiter zu vervollständigen, da `cons`-Listen *selbstreferentiell* sind, und damit auch Funktionen auf Listen in aller Regel *selbstaufrufend* oder *rekursiv* sind:

```
(define (f eine-liste)
  (cond
    [(empty? eine-liste)
```

```
    ...]
   [(cons? eine-liste)
    ... (first eine-liste) ...
    ... (f (rest eine-liste)) ...]))
```

12.3 Beispiel

Gefragt ist nach einer Funktion, die eine Liste von Zahlen konsumiert (sagen wir, die Liste der Preise aller Geschenke auf der Weihnachtswunschliste) und deren Summe zurückliefert.

1. Die Datenanalyse ergibt, daß es um Listen von Zahlen geht. Die dazu passende Datendefinition ergibt sich aus der Definition für Listen von Zeichenketten durch einfaches Ersetzen:

   ```
   ;; Eine Liste von Zahlen ist:
   ;; - die leere Liste
   ;; - oder ein cons aus einer Zahl und einer Liste von Zahlen
   ```

2. Der Vertrag sieht so aus:

   ```
   ;; listensumme : list (zahl) -> zahl
   ```

3. Das Funktionsgerüst ergibt sich aus dem Vertrag:

   ```
   (define (listensumme zahlen)
     ...)
   ```

4. Bei den Beispielen ist vielleicht der `empty`-Fall der schwierigste. Aber die Wunschliste eines wunschlosen Kindes ist kostenlos:

To test	(listensumme empty)
Expected	0

To test	(listensumme (list 9.95))
Expected	9.95

To test	(listensumme (list 1.22 3.50 2.23))
Expected	6.95

To test	(listensumme (list 1000 2000 3000))
Expected	6000

5. Hier die Funktionsschablone:

   ```
   (define (listensumme zahlen)
     (cond
       [(empty? zahlen)
        ...]
       [(cons? zahlen)
        ... (first zahlen) ...
        ... (listensumme (rest zahlen)) ...]))
   ```

6. Im Funktionsrumpf muß zuerst der `empty`-Fall ausgefüllt werden. Der ergibt sich aus den Überlegungen zur leeren Wunschliste:

```
(define (listensumme zahlen)
  (cond
    [(empty? zahlen)
     0]
    [(cons? zahlen)
     ... (first zahlen) ...
     ... (listensumme (rest zahlen)) ...]))
```

   Für den `cons`-Fall führt folgende Überlegung zum Ziel: Wenn die Summe der *restlichen* Zahlen bekannt ist sowie die *erste* Zahl, dann ergibt sich die Summe *aller* Zahlen durch Addition der beiden:

```
(define (listensumme zahlen)
  (cond
    [(empty? zahlen)
     0]
    [(cons? zahlen)
     (+ (first zahlen)
        (listensumme (rest zahlen)))]))
```

7. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

Anmerkungen:

- Es empfiehlt sich, möglichst viele Beispiele im Stepper durchzuspielen.

12.4 Beispiel

Gefragt ist eine Funktion, die für eine Liste von Eissorten feststellt, ob Waldmeister darin enthalten ist. Dieses Beispiel erfordert die Kombination der Konstruktionsanleitung für Listen mit der Anleitung für Fallunterscheidungen.

1. Die Datenanalyse ist einfach: die Aufgabenstellung enthält bereits das Wort „Liste". Jede Eissorte wird durch eine Zeichenkette repräsentiert — es handelt sich bei der Eingabe um Listen von Zeichenketten. Zur Erinnerung:

```
;; Eine Liste von Zeichenketten ist:
;; - die leere Liste
;; - oder ein cons aus einer Zeichenkette und einer Liste von Zeichenketten
```

2. Der Vertrag der Funktion sieht so aus:

```
;; enthaelt-waldmeister? : list (zeichenkette) -> bool
```

3. Das Funktionsgerüst ergbibt sich — wie immer — aus dem Vertrag:

```
(define (enthaelt-waldmeister? eissorten)
  ...)
```

4. Beispiele:

To test	(enthaelt-waldmeister? empty)
Expected	false

To test	(enthaelt-waldmeister? (list "Tuttifrutti"))
Expected	false

To test	(enthaelt-waldmeister? (list "Waldmeister"))
Expected	true

To test	(enthaelt-waldmeister? (list "Tuttifrutti" "Waldmeister"))
Expected	true

To test	(enthaelt-waldmeister? (list "Tuttifrutti" "Waldemeister"))
Expected	false

To test	(enthaelt-waldmeister? (list "Tuttifrutti" "Waldmeister" "Vanille"))
Expected	true

5. Hier ist die Funktionsschablone:

```
(define (enthaelt-waldmeister? eissorten)
  (cond
    [(empty? eissorten)
     ...]
    [(cons? eissorten)
     ... (first eissorten) ...
     ... (enthaelt-waldmeister? (rest eissorten)) ...]))
```

6. Im Funktionrumpf muß erst einmal der `empty`-Fall ausgefüllt werden. Der ergibt sich aus dem ersten Testfall:

```
(define (enthaelt-waldmeister? eissorten)
  (cond
    [(empty? eissorten)
     false]
    [(cons? eissorten)
     ... (first eissorten) ...
     ... (enthaelt-waldmeister? (rest eissorten)) ...]))
```

Als nächstes ist der `cons`-Fall an der Reihe — hier trifft die Konstruktionsanleitung für Fallunterscheidungen zu: Wenn `(first eissorten)` kann `"Waldmeister"` sein — oder nicht.

```
(define (enthaelt-waldmeister? eissorten)
  (cond
    [(empty? eissorten)
     false]
    [(cons? eissorten)
     (cond
       [(string=? (first eissorten) "Waldmeister")
         ... (enthaelt-waldmeister? (rest eissorten)) ...]
       [else
         ... (enthaelt-waldmeister? (rest eissorten)) ...])]))
```

Wieder ist der erste Fall einfach — das Material `(enthaelt-waldmeister? (rest eissorten))` aus der Vorlage wird gar nicht benötigt:

```
(define (enthaelt-waldmeister? eissorten)
  (cond
    [(empty? eissorten)
     false]
    [(cons? eissorten)
     (cond
       [(string=? (first eissorten) "Waldmeister")
        true]
       [else
         ... (enthaelt-waldmeister? (rest eissorten)) ...])]))
```

Schließlich ist der letzte Fall an der Reihe: Wenn `"Waldmeister"` in den restlichen Eissorten enthalten ist, dann muß `true` herauskommen. Andernfalls muß `false` herauskommen, da ja schon der Fall ausgeschlossen ist, daß das erste Element `"Waldmeister"` ist.

Mit anderen Worten: Wenn `(enthaelt-waldmeister? (rest eissorten))` den Wert `true` zurückliefert, dann muß `true` herauskommen, und wenn `(enthaelt-waldmeister? (rest eissorten))` den Wert `false` zurückliefert, muß `false` herauskommen. Es reicht also, einfach `(enthaelt-waldmeister? (rest eissorten))` zurückzuliefern und die Ellipsen zu entfernen:

```
(define (enthaelt-waldmeister? eissorten)
  (cond
    [(empty? eissorten)
     false]
    [(cons? eissorten)
     (cond
       [(string=? (first eissorten) "Waldmeister")
        true]
       [else
         (enthaelt-waldmeister? (rest eissorten))])]))
```

7. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

12.5 Funktionen, die Listen produzieren

Gefragt ist eine Funktion, die von den Preisen der Liste einer Weihnachtswunschliste jeweils 10% (wegen eines Spezialrabatts) abzieht.

1. Es ist sofort klar, daß die geforderte Funktion eine Liste von Zahlen konsumiert. Beim Rückgabewert benutzt die Aufgabenstellung eine nachlässige aber häufig vorkommende Formulierung: herauskommen soll eine Liste der reduzierten Preise, also wiederum eine Liste von Zahlen. Die Datendefinition für Listen von Zahlen übernehmen wir vom ersten Beispiel.

   Es handelt sich also einerseits um eine Funktion, die sowohl eine Liste konsumiert — und damit ist die Konstruktionsanleitung für derartige Funktionen anwendbar — als auch eine zurückliefern soll.

2. Der Vertrag sieht so aus:

   ```
   ;; rabattiere-liste : list (zahl) -> list (zahl)
   ```

3. Funktionsgerüst:

```
(define (rabattiere-liste zahlen)
  ...)
```

4. Beispiele:

To test	(rabattiere-liste empty)
Expected	empty

To test	(rabattiere-liste (list 10 100))
Expected	(list 9 90)

To test	(rabattiere-liste (list 1.22 3.50 2.23))
Expected	(list 1.098 3.15 2.007)

Es fällt auf, daß die herauskommenden Zahlen keine ganzen Cent-Beträge sind — aber nach denen hat die Aufgabenstellung auch nicht gefragt.

5. Hier ist die Funktionsschablone für Funktionen, die Listen konsumieren:

```
(define (rabattiere-liste zahlen)
  (cond
    [(empty? zahlen)
     ...]
    [(cons? zahlen)
     ... (first zahlen) ...
     ... (rabattiere-liste (rest zahlen)) ...]))
```

6. Im Funktionsrumpf wird der `empty`-Fall zuerst ausgefüllt. Er ergibt sich wieder aus dem ersten Testfall:

```
(define (rabattiere-liste zahlen)
  (cond
    [(empty? zahlen)
     empty]
    [(cons? zahlen)
     ... (first zahlen) ...
     ... (rabattiere-liste (rest zahlen)) ...]))
```

Es ist wieder sinnvoll, die Bedeutung der beiden Fragmente aus der Schablone aufzuschreiben:

- `(first zahlen)` ist das erste Element der Liste, also die erste zu rabattierende Zahl.
- Bei `(rabattiere-liste (rest zahlen))` ist bekannt, daß `rabattiere-liste` eine Liste mit den rabattierten Elementen von `(rest zahlen)` zurückliefert — also von allen Zahlen der Liste bis auf die erste. Es lohnt sich, das anhand eines Beispiels aufzuschreiben. Wenn `zahlen` gerade

  ```
  (list 1.22 3.50 2.23)
  ```

  (der erste Testfall) ist, so muß `(rabattiere-liste (rest zahlen))` gerade

```
(list 3.15 2.007)
```

sein.

Die Frage ist jetzt also, wie sich aus der ersten Zahl der Liste (in diesem Fall `1.22`) sowie den rabattierten restlichen Zahlen `(list 3.15 2.007)` das gewünschte Resultat —

```
(list 1.098 3.15 2.007)
```

berechnen läßt. Die `1.098` entsteht durch die Multiplikation von `1.22` mit `0.9`, es mußt also

```
(* 0.9 (first zahlen))
```

vorkommen. Daraus und aus der Rest-Liste wird das gewünschte Ergebnis mit `cons` weil gilt:

```
(cons (* 0.9 1.22) (list 3.15 2.007)) ⟹ (list 1.098 3.15 2.007)
```

Damit ist es jetzt möglich, den Rumpf zu vervollständigen:

```
(define (rabattiere-liste zahlen)
  (cond
    [(empty? zahlen)
     empty]
    [(cons? zahlen)
     (cons (* 0.9 (first zahlen))
           (rabattiere-liste (rest zahlen)))]))
```

7. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

12.6 Funktionen, die mehrere Listen konsumieren

Gefragt ist eine Funktion, die zwei Listen aus Zahlen verkettet.

1. Die Datenanalyse ist einfach: es geht ausschließlich um Listen von Zahlen.

2. Der Vertrag ergibt sich direkt aus der Aufgabenstellung:

```
;; verkette : list (zahl) -> list (zahl)
```

3. Ebenso das Funktionsgerüst:

```
(define (verkette zahlen-1 zahlen-2)
  ...)
```

4. Beispiele:

To test	`(verkette empty empty)`
Expected	`empty`

To test	`(verkette (list 1 2 3) (list 4 5 6)`
Expected	`(list 1 2 3 4 5 6)`

To test	(verkette (list 1 2 3) empty
Expected	(list 1 2 3)

To test	(verkette empty (list 1 2 3)
Expected	(list 1 2 3)

5. Bei der Funktionsschablone tritt ein Problem auf: diese ist nur darauf ausgerichtet, daß die Funktion *eine* Liste konsumiert. Es gibt also zwei Möglichkeiten, die Funktionsschablone zu konstruieren. Hier die erste, die sich nach `zahlen-1` richtet:

```
(define (verkette zahlen-1 zahlen-2)
  (cond
    [(empty? zahlen-1)
     ...]
    [(cons? zahlen-1)
     ... (first zahlen-1) ...
     ... (verkette (rest zahlen-1) zahlen-2) ...]))
```

Die zweite richtet sich stattdessen nach `zahlen-2`:

```
(define (verkette zahlen-1 zahlen-2)
  (cond
    [(empty? zahlen-2)
     ...]
    [(cons? zahlen-2)
     ... (first zahlen-2) ...
     ... (verkette zahlen-1(rest zahlen-2)) ...]))
```

Für solche Fälle gibt es kein allgemeingültiges Rezept — es ist nötig, beide Fälle zu verfolgen um zu sehen, welche zum Erfolg führt. Im Fall von `verkette`, ist es möglich, die Entscheidung anhand eines Beispiels zu treffen, wenn es mit `cons` statt mit `list` geschrieben wird:

To test	(verkette (cons 1 (cons 2 empty)) (cons 3 (cons 4 empty)))
Expected	(cons 1 (cons 2 (cons 3 (cons 4 empty))))

Hier ist sichtbar, daß die zweite Liste `(cons 3 (cons 4 empty))` vollkommen unverändert im Resultat auftaucht — sie ist also nur „Beifahrer“ bei der Berechnung, und die Schablone richtet sich nach der ersten Liste.

6. Beim Rumpf ist wieder der `empty`-Fall zuerst an der Reihe — er ergibt sich anhand eines Testfalls:

```
(define (verkette zahlen-1 zahlen-2)
  (cond
    [(empty? zahlen-1)
     zahlen-2]
    [(cons? zahlen-1)
     ... (first zahlen-1) ...
     ... (verkette (rest zahlen-1) zahlen-2) ...]))
```

Die zweite Variante läßt sich wieder am einfachsten anhand eines Beispiels entscheiden. Sei `zahlen-1 (list 1 2 3)` sowie `zahlen-2 (list 4 5 6)`. Dann ist `(first zahlen-1)` gerade 1, `(rest zahlen-1)` gerade `(list 2 3)` und

```
(verkette (rest zahlen-1) zahlen-2)
```

gerade

```
(list 2 3 4 5 6)
```

Da am Ende `(list 1 2 3 4 5 6)` herauskommen soll, muß der vollständige Funktionsrumpf folgendermaßen lauten:

```
(define (verkette zahlen-1 zahlen-2)
  (cond
    [(empty? zahlen-1)
     zahlen-2]
    [(cons? zahlen-1)
     (cons (first zahlen-1)
           (verkette (rest zahlen-1) zahlen-2))]))
```

7. Die Tests bestätigen die erwarteten Ergebnisse aus Schritt 5.

Anmerkungen:

- Bei `verkette` ist es nicht nötig, die „analytische“ Überlegung anzustellen, um zu entscheiden, ob die Schablone sich nach der ersten oder zweiten Liste richten soll — Probieren ist ein absolut zulässiges Mittel, um diese Frage zu entscheiden.
- Bei der Demonstration im Unterricht bietet es sich an, die Klasse abstimmen zu lassen, welche Alternative zuerst ausprobiert werden soll. Erfahrungsgemäß stimmen meist auch einige von den „Cracks“ für die falsche Variante.

12.7 Aufgaben

Aufgabe 12.1 Schreibe eine Funktion, die eine Liste von Zahlen konsumiert und deren Produkt zurückliefert.

Anmerkung: Der schwierigste Zeil der Aufgabe ist wahrscheinlich der `empty`-Fall.

Aufgabe 12.2 Schreibe eine Funktion `ohne-mehrwertsteuer-16%`, die eine Liste von Preisen einer Einkaufsliste konsumiert und eine Liste der Preise jeweils ohne die Mehrwertsteuer zurückliefert.

Schreibe eine Aufgabe `ohne-mehrwertsteuer`, die eine Liste von Preisen einer Einkaufsliste sowie einen Mehrwertsteuersatz in Prozent konsumiert und eine Liste der Preise jeweils ohne die Mehrwertsteuer zurückliefert.

Aufgabe 12.3 Schreibe eine Funktion `schokokeks-gewichte`, die eine Liste von Schokokeksen konsumiert und eine Liste ihrer Gewichte zurückliefert.

Aufgabe 12.4 Anschluß an Aufgabe 9.5:

- Schreibe eine Funktion `fuettere-schlangen`, die eine Liste von Schlangen konsumiert und alle Schlangen in der Liste füttert.

- Schreibe eine Funktion `duenne-schlangen`, die eine Liste von Schlangen konsumiert und eine Liste der dünnen Schlangen in der Liste zurückliefert.

Aufgabe 12.5 Schreibe eine Funktion, die eine Liste von Namen von Weihnachtsgeschenken passend für Jungs macht, also alle Vorkommen von `"Barbie"` durch `"Ken"` ersetzt.

Verallgemeinere die Funktion zu einer, die eine Weihnachtsgeschenkliste sowie die Namen zweier Weihnachtsgeschenke konsumiert, und die in der Liste alle Vorkommen des ersten Geschenks durch das zweite ersetzt.

Aufgabe 12.6 Schreibe eine Funktion `friss-aepfel`, die eine Liste von Zeichenketten konsumiert und eine Liste zurückliefert, in der alle Äpfel fehlen.

Aufgabe 12.7 Schreibe eine Funktion `streiche-zu-teuer`, die eine Liste von Weihnachtswunschlistenpreisen sowie einen Höchstbetrag konsumiert und eine Liste der Elemente der Weihnachtswunschliste zurückliefert, bei der alle Preise, die den Höchstbetrag überschreiten, fehlen.

Aufgabe 12.8 Schreibe eine Funktion `verkette-listen`, die eine Liste von Listen von Zahlen konsumiert und eine Liste aller Zahlen der Teillisten zurückliefert.

Aufgabe 12.9 Funktionieren die Funktionen `verkette` und `verkette-listen` auch mit Listen von Zeichenketten? Listen von Schlangen? Erfinde eine bessere Schreibweise für die Verträge der beiden Funktionen!

Aufgabe 12.10 Schreibe eine Funktion `snoc`, die eine Liste und eine Zahl konsumiert und die Zahl hinten an die Liste anhängt.

Anhang A

Kurzüberblick Scheme

A.1 Wörter

A.1.1 Zahlen

ganze Zahlen `23`, `42`, `189473248732424`

Brüche `1/3`, `2390874/132948576`
inkorrekt: `2390874 /132948576` (enthält Leerzeichen)

Fließkomma `0.3`, `3.1415926`, `6.02e23`, `-.738E-17`

Hinweis: Solche Zahlen werden im allgemeinen nur mit beschränkter Genauigkeit dargestellt.

A.1.2 Leerzeichen und Zeilenumbrüche

Ein Leerzeichen oder ein Zeilenumbruch ist notwendig, um zwei Wörter, die nebeneinander stehen, voneinander zu trennen.

Sie sind außerdem zulässig (aber nicht notwendig) vor und nach Klammern. Ansonsten haben zusätzliche Leerzeichen und Zeilenumbrüche keine Auswirkungen auf die Bedeutung eines Programms.

A.1.3 Klammern

Linke Klammern und rechte Klammern müssen in Zahl und Art übereinstimmen.

Hinweis: Klammern haben *immer* Bedeutung in einem Programm. Es nicht erlaubt, Klammernpaare hinzuzufügen oder wegzulassen, auch wenn die Bedeutung vorher und nachher „intuitiv“ gleich ist.

A.1.4 Namen

Namen können aus folgenden Bestandteilen bestehen:

- Buchstaben
- Bindestriche
- Unterstriche
- die Zeichen `+`, `*`, `/`, `=`, `>`, `<`, `?`, `&`, `:`

- Ziffern, sofern sie nicht an erster Stelle vorkommen

Folgende Bestandteile dürfen nicht vorkommen:

- Klammern (egal welcher Art)
- Semikolon
- Komma
- Anführungszeichen (egal welcher Art)
- Leerzeichen

Korrekte Beispiele:

- `+`, `*`, `/`, usw.
- `sqrt` (vordefiniert)
- `true` (vordefiniert)
- `a-number` (Komposita werden konventionsmäßig mit Bindestrichen geschrieben)
- `aNumber` (wenn auch konventionswidrig)
- `a_number` (wenn auch konventionswidrig)
- `number->string` (vordefiniert)
- `cons?` (vordefiniert)

Inkorrekte Beispiele:

- `define`, `cond`, `define-struct`, `else` (weil Schlüsselwörter)
- `1+` (weil beginnend mit einer Ziffer)

A.1.5 Kommentare

Alles, was auf einer Zeile nach einem Semikolon (das nicht innerhalb einer Zeichenkette steht) folgt, wird von Scheme ignoriert und kann damit als Kommentar dienen.

Beispiel:

```
(define pi 3.14159265) ; wichtige Konstante
```

A.1.6 Zeichenketten

Eine Zeichenkette repräsentiert ein Stück beliebigen Text. Der Text wird dafür zwischen doppelte Anführungszeichen gesetzt. Eventuell darin vorkommende doppelte Anführungszeichen werden als `\"` dargestellt; eventuell vorkommende `\` werden als `\\` dargestellt. Beispiele:

- `"foo"`
- `"Das Prinzip \"Hoffnung\""`
- `"Kommt ein Backslash geflogen, \\ ..."`

A.2 Syntax

A.2.1 Funktionsaufrufe

Der Ausdruck

`(`f e_0 e_1 `...` e_n`)`

ist eine Anwendung (oder ein Aufruf) der Funktion f auf die Ergebnisse der Ausdrücke e_0, e_1, …, e_n. Beispiele:

- `(+ 1 2)` ist die Anwendung der Funktion `+` auf die Werte 1 und 2 — das Ergebnis ist die Summe der beiden Zahlen.
- `(* (+ 3 4) 5)` ist die Anwendung der Funktion `*` auf zwei Zahlen: das Ergebnis von `(+ 3 4)` (7) und die Zahl 5, also insgesamt 35.
- `(= x y)` ist die Anwendung der Funktion `=` auf die Werte der Variablen `x` und `y`. Dabei kommt `true` oder `false` heraus, je nachdem, ob die beiden Werte (numerisch) gleich sind oder nicht.

Hinweise:

- Die Funktion bzw. der Operator steht *immer* vorn, nie dazwischen wie bei vielen Operationen in der Mathematik.
- Die Klammern sind zwingend erforderlich.
- Zusätzliche Klammern sind nicht zulässig.

Inkorrekte Beispiele:

- `(1 + 2)` (Der Operator steht nicht vorn)
- `+ 1 2` (Klammern fehlen)
- `((+ 1 2))` (zusätzliche Klammern)

A.2.2 Variablendefinitionen

`(define` v e`)`

definiert die Variable v auf den Wert des Ausdrucks e, so daß zukünftige Vorkommen von v durch den Wert von e ersetzt werden.

Beispiele:

- `(define number 121243234)`
  Nach dieser Definition wird `number` durch die Zahl 121243234 ersetzt.

A.2.3 Funktionsdefinitionen

`(define (`f p_1 p_1 `...` p_n`)`
  e`)`

definiert die Funktion f. Jeder Parameter p_i ist ein Variablenname, und diese Variablen können innerhalb des *Rumpfes* e verwendet werden.

Beispiele:

- `(define (square n) (* n n))`

  oder

```
(define (square n)
  (* n n))
```

Nach dieser Definition liefert z.B. `(square 5)` den Wert 25: `(square 5)` wird durch den Rumpf `(* n n)` ersetzt, wobei 5 für `n` eingesetzt wird.

A.2.4 Fallunterscheidungen

```
(cond [q1 a1]
      [q2 a2]
      ...
      [qn an])
```

Dieser Ausdruck gibt den Wert einer der a_i zurück, je nachdem welcher der Ausdrücke q_i (die jeweils einen Wahrheitswert ergeben müssen) der erste ist, der `true` ergibt. Statt der letzten Frage q_n kann auch das Schlüsselwort `else` erscheinen, was besagt, daß der Ausdruck a_n ergeben soll, wenn keine der Fragen `true` ergeben hat.

Beispiel:

```
(cond [(> temperatur 35) "heiss"]
      [(> temperatur 25) "warm"]
      [(> temperatur 15) "mittel"]
      [else 'kalt])
```

Hinweise:

- Jede Klausel muß von einem Klammernpaar umschlossen werden.
- Jede Klausel muß genau zwei Ausdrücke (Frage und Antwort) enthalten.

Inkorrekte Beispiele:

```
(cond (< number 4) 5
      (>= number 4) 6)
```

Hier fehlen Klammern um die Klauseln.

```
(cond [(+ bignum 4) 5]
      [(- bignum 4) 6])
```

In diesem Fall ergeben die Fragen keine Wahrheitswerte.

A.2.5 Strukturdefinitionen

`(define-struct` n `(`f_1 f_2 `...` f_n`))`

Dabei ist n der Name der Struktur und die f_i (ihrerseits Namen) sind *Felder*. Diese Strukturdefinition definiert mehrere Funktionen auf einmal:

- `(make-`n v_1 v_2 `...`v_n`)`
  erzeugt ein Objekt der Struktur-Klasse n, in dem die Felder f_1, f_2, ..., f_n mit den Werten v_1, v_2, ..., v_n besetzt sind.
- `(`n`?` v`)` stellt fest, ob das (beliebige) Objekt v eine Struktur der Klasse n ist oder nicht, und gibt entsprechend `true` oder `false` zurück.
- `(`v`-`f_i s`)` liefert für eine Struktur s der Klasse n den Inhalt von Feld f_i.

Die Funktion `make-`*n* heißt *Konstruktor*, die Funktion *n*`?` heißt *Prädikat*, und die die v-f_i heißen *Selektoren*. Beispiel:

```
;; Ein Name besteht aus zwei Zeichenketten - Vor- und Nachname
(define-struct name (personal family))
;; make-name: zeichenkette x zeichenkette -> name
;; name?: object -> boolean
;; name-personal: name -> zeichenkette
;; name-family: name -> zeichenkette
```

A.3 Vokabular

A.3.1 Schlüsselwörter

Ein syntaktisches Schlüsselwort nach einer öffnenden Klammer bedeutet, daß zwischen den Klammern kein Funktionsaufruf, sondern eine *Spezialform* steht, bei der besondere Regeln gelten. Die syntaktischen Schlüsselwörter in rudimentärem Scheme sind `define`, `cond`, `else` und `define-struct`. (Erläuterungen im vorangegangenen Abschnitt.)

A.3.2 Arithmetik

- `+`, `-`, `*`, `/` (...)
- `sqrt` liefert die Wurzel einer Zahl
- `remainder` liefert den Divisionsrest zweier Zahlen

A.3.3 Numerische Konstanten

- `pi` (3.1415...)
- `e` (2.7182...)

A.3.4 Wahrheitswerte und Prädikate

- `true` steht für „wahr“, `false` für „falsch“
- `<`, `<=`, `>`, `>=`, `=` akzeptieren jeweils zwei Zahlen als Eingabe und liefern einen Wahrheitswert
- `zero?`, `positive?`, `negative?` akzeptieren jeweils eine Zahl als Eingabe und liefern einen Wahrheitswert.
- `equal?` nimmt zwei Objekte (gleich welcher Klasse) als Eingaben und liefert einen Wahrheitswert
- `and`, `or`, `not` sind die entsprechenden Operationen auf Wahrheitswerten

Beispiele:

```
(or (<= 1 3) (zero? 5)) => true
(and (<= 1 3) (zero? 5)) => false
(and (<= 1 3) (not (zero? 5))) => true
```

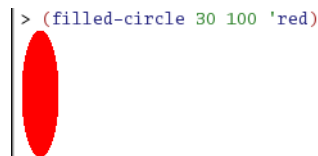
A.3.5 Bilder

Die hier beschriebenen Funktionen zur Erzeugung von Bildern sind in allen HtDP-Sprachebenen verfügbar.

Die Funktionen, die direkt Bilder mit Rechtecken oder Ellipsen erzeugen, konsumieren eine Zeichenkette, die eine Farbe bezeichnet. Folgende Farben stehen (u.a.) zur Auswahl: `"red"`, `"pink"`, `"brown"`, `"gold"`, `'yellow"`, `"green"`, `"blue"`, `"violet"`, `"black"`, `"white"`, `"gray"`, `"silver"`. Eine vollständige Liste findet sich im Help Desk unter dem Stichwort `color-database<%>`.

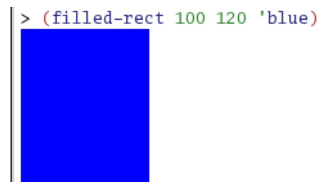
- `filled-circle: zahl zahl zeichenkette -> bild`

  Diese Funktion liefert liefert ein Bild mit einer ausgefüllten Ellipse, gegeben deren Breite, Höhe und Farbe.

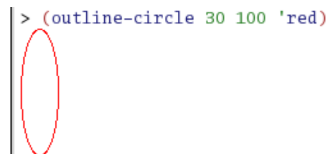
- `filled-rect: zahl zahl zeichenkette -> bild`

  Diese Funktion liefert liefert ein Bild mit einem ausgefüllten Rechteck, gegeben dessen Breite, Höhe und Farbe.

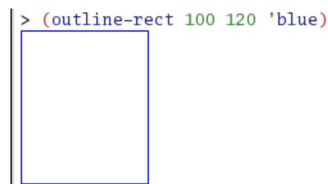
- `outline-circle: zahl zahl zeichenkette -> bild`

  Diese Funktion liefert liefert ein Bild mit dem Umriß einer Ellipse, gegeben dessen Breite, Höhe und Farbe des Umrisses.

- `outline-rect: zahl zahl zeichenkette -> bild`

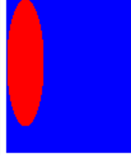
  Diese Funktion liefert liefert ein Bild mit dem Umriß eines Rechtecks, gegeben dessen Breite, Höhe und Farbe.

- `image+: bild bild -> bild`

  Erzeugt aus zwei Bildern einen neues, indem auf das erste konsumierte Bild alle nichtweißen Punkte des zweiten in die obere linke Ecke gezeichnet werden. („Überschüssige“ Punkte im zweiten Bild werden ignoriert.)

```
> (image+ (filled-rect 100 120 'blue)
          (filled-circle 30 100 'red))
```

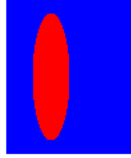


Erzeugt aus zwei Bildern einen neues, indem auf das erste konsumierte Bild alle nichtweißen Punkte des zweiten in die obere linke Ecke gezeichnet werden. („Überschüssige“ Punkte im zweiten Bild werden ignoriert.)

- `offset-image+: bild zahl zahl bild -> bild`

  Erzeugt aus zwei Bildern einen neues, indem auf das erste konsumierte Bild alle nichtweißen Punkte des zweiten an die durch die beiden konsumierten Zahlen bezeichnetetn Koordinaten gezeichnet werden. Dabei bezeichnet die erste Zahl die X-Koordinate von links, die zweite Zahl die Y-Koordinate von oben. („Überschüssige“ Punkte im zweiten Bild werden ignoriert.)

```
> (offset-image+ (filled-rect 100 120 'blue)
                 20 10
                 (filled-circle 30 100 'red))
```



- `image-height : bild -> zahl`

  Diese Funktion konsumiert ein Bild und liefert dessen Höhe.

- `image-width : bild -> zahl`

  Diese Funktion konsumiert ein Bild und liefert dessen Breite.

- `image=? : bild bild -> boolean`

  Diese Funktion konsumiert zwei Bilder und liefert `true`, falls diese identisch sind und `false`, falls nicht.

- `image? : objekt -> boolean`

  `Image?` ist das Prädikat für Bilder; es konsumiert ein beliebiges Objekt und liefert `true`, falls es sich dabei um ein Bild handelt und `false`, falls nicht.

A.3.6 Listen

- `empty` ist ein vordefinierter Name für die leere Liste
- `cons` nimmt ein beliebiges Objekt und eine Liste als Eingaben, und liefert eine neue Liste zurück, deren erstes Element das Objekt, und dessen Rest die Eingabeliste ist.
- `first` nimmt eine nichtleere Liste als Eingabe und liefert deren erstes Element.
- `rest` nimmt eine nichtleere Liste als Eingabe und liefert deren Rest, also eine Liste, bei der gegenüber der Eingabe das erste Element fehlt.
- `empty?` akzeptiert ein Objekt als Eingabe und liefert einen Wahrheitswert, der angibt, ob es sich dabei um die leere Liste handelt.

- `cons?` akzeptiert ein Objekt als Eingabe und liefert einen Wahrheitswert, der angibt, ob es sich dabei um eine nichtleere Liste handelt.

Beispiele:

```
(cons "element" empty) => (cons "element" empty)

(define pizza-toppings
  (cons "anchovies" (cons "salami" (cons "gummi" empty))))

pizza-toppings => (cons "anchovies" (cons "salami" (cons "gummi" empty)))

(first pizza-toppings) => "anchovies"

(rest pizza-toppings) => (cons "salami" (cons "gummi" empty))

(first (rest pizza-toppings)) => "salami"

(empty? pizza-toppings) => false

(cons? pizza-toppings) => true

(empty? (rest (rest (rest pizza-toppings)))) => true
```

A.3.7 Sonstiges

- `error` akzeptiert eine Zeichenkette als Eingabe und bricht die Programmausführung unter Anzeige des enthaltenen Textes an.