Informatik 1, WS 2015/16 Universität Tübingen

Prof. Dr. Klaus Ostermann

November 6, 2017

mit Beiträgen von Jonathan Brachthäuser, Yufei Cai, Yi Dai und Tillmann Rendel

Große Teile dieses Skripts basieren auf dem Buch "How To Design Programs" von M. Felleisen, R.B. Findler, M. Flatt und S. Krishnamurthi.

Contents

1 Programmieren mit Ausdrücken

1.1 Programmieren mit arithmetischen Ausdrücken

Dieser Teil des Skripts basiert auf [HTDP/2e] Kapitel 1

Jeder von Ihnen weiß, wie man Zahlen addiert, Kaffee kocht, oder einen Schrank eines schwedischen Möbelhauses zusammenbaut. Die Abfolge von Schritten, die sie hierzu durchführen, nennt man *Algorithmus*, und Sie wissen wie man einen solchen Algorithmus ausführt. In diesem Kurs werden wir die Rollen umdrehen: Sie werden den Algorithmus programmieren, und der Computer wird ihn ausführen. Eine formale Sprache, in der solche Algorithmen formuliert werden können, heißt *Programmiersprache*. Die Programmiersprache, die wir zunächst verwenden werden, heißt *BSL*. BSL steht für "Beginning Student Language". Zum Editieren und Ausführen der BSL Programme verwenden wir *DrRacket*. DrRacket kann man unter der URL `http://racket-lang.org/` herunterladen. Bitte stellen Sie als Sprache "How To Design Programs - Anfänger" ein. Folgen Sie diesem Text am besten, indem Sie DrRacket parallel starten und immer mitprogrammieren.

Viele einfache Algorithmen sind in einer Programmiersprache bereits vorgegeben, z.B. solche zur Arithmetik mit Zahlen. Wir können "Aufgaben" stellen, indem wir DrRacket eine Frage stellen, auf die uns DrRacket dann im Ausgabefenster die Antwort gibt. So können wir zum Beispiel die Frage

```
(+ 1 1)
```

im *Definitionsbereich* (dem oberen Teil der DrRacket Oberfläche) stellen — als Antwort erhalten wir im *Interaktionsbereich* (dem Bereich unterhalb des Definitionsbereichs) bei Auswertung dieses Programms ("Start" Knopf) `2`. Im Definitionsbereich schreiben und editieren Sie ihre Programme. Sobald Sie hier etwas ändern, taucht der "Speichern" Knopf auf, mit dem Sie die Definitionen in einer Datei abspeichern können. Im Interaktionsbereich wird das Ergebnis einer Programmausführung angezeigt; außerdem können hier Ausdrücke eingegeben werden, die sofort ausgewertet werden aber nicht in der Datei mit abgespeichert werden.

Die Art von Programmen bzw. Fragen wie `(+ 1 1)` nennen wir *Ausdrücke*. In Zukunft werden wir solche Frage/Antwort Interaktionen so darstellen, dass wir vor die Frage das $>$ Zeichen setzen und in der nächsten Zeile die Antwort auf die Frage:

```
> (+ 1 1)
2
```

Operationen wie `+` nennen wir im Folgenden *Funktionen*. Die Operanden wie `1` nennen wir *Argumente*. Hier einige weitere Beispiele für Ausdrücke mit anderen Funktionen:

```
> (+ 2 2)
4
> (* 3 3)
9
> (- 4 2)
```

```
2
> (/ 6 2)
3
> (sqr 3)
9
> (expt 2 3)
8
> (sin 0)
0
```

Die Argumente dieser Funktionen sind jeweils Zahlen, und das Ergebnis ist auch wieder eine Zahl. Wir können auch direkt eine Zahl als Ausdruck verwenden. Zum Beispiel:

```
> 5
5
```

DrRacket zeigt als Ergebnis wieder genau dieselbe Zahl an. Eine Zahl, die direkt in einem Ausdruck verwendet wird, heißt auch Zahlen*literal*.

Für manche Ausdrücke kann der Computer das mathematisch korrekte Ergebnis nicht berechnen. Stattdessen erhalten wir eine Annäherung an das mathematisch korrekte Ergebnis. Zum Beispiel:

```
> (sqrt 2)
#i1.4142135623730951


> (cos pi)
#i-1.0
```

Das `i` im letzten Ergebnis steht für "inexact", also ungenau. In BSL kann man an diesem `i` sehen, ob eine Zahl ein exaktes Ergebnis oder nur ein angenähertes Ergebnis ist.

Programme beinhalten Ausdrücke. Alle Programme, die wir bisher gesehen haben, *sind* Ausdrücke. Jeder von Ihnen kennt Ausdrücke aus der Mathematik. Zu diesem Zeitpunkt ist ein Ausdruck in unserer Programmiersprache entweder eine Zahl, oder etwas, das mit einer linken Klammer "(" startet und mit einer rechten Klammer ")" endet. Wir bezeichnen Zahlen als *atomare Ausdrücke* und Ausdrücke, die mit einer Klammer starten, als *zusammengesetzte Ausdrücke*. Später werden andere Arten von Ausdrücken hinzukommen.

Programieren Sie einen Ausdruck, der die Summe der Zahlen 3, 5, 19, und 32 berechnet.

Wie kann man mehr als zwei Zahlen addieren? Hierzu gibt es zwei Möglichkeiten:

Durch Schachtelung:

```
> (+ 2 (+ 3 4))
9
```

oder durch Addition mit mehr als zwei Argumenten:

```
> (+ 2 3 4)
9
```

Immer wenn Sie in BSL eine Funktion wie `+` oder `sqrt` benutzen möchten, schreiben Sie eine öffnende Klammer, gefolgt vom Namen der Funktion, dann einem Lehrzeichen (oder Zeilenumbruch) und dann die Argumente der Funktion, also in unserem Fall die Zahlen, auf die die Funktion angewandt werden soll.

Programmieren Sie einen Ausdruck, der den Durchschnitt der Zahlen 3, 5, 19 und 32 berechnet.

Am Beispiel der Schachtelung haben Sie gesehen, dass auch zusammengesetzte Ausdrücke als Argumente zugelassen sind. Diese Schachtelung kann beliebig tief sein:

```
> (+ (* 5 5) (+ (* 3 (/ 12 4)) 4))
38
```

Das Ergebnis für einen solchen geschachtelten Ausdruck wird so berechnet, wie sie es auch auf einem Blatt Papier machen würden: Wenn ein Argument ein zusammengesetzter Ausdruck ist, so wird zunächst das Ergebnis für diesen Ausdruck berechnet. Dieser Unterausdruck ist möglicherweise selber wieder geschachtelt; in diesem Fall wird diese Berechnungsvorschrift auch auf diese Unterausdrücke wieder angewendet (*rekursive* Anwendung). Falls mehrere Argumente zusammengesetzte Ausdrücke sind, so werden diese in einer nicht festgelegten Reihenfolge ausgewertet. Die Reihenfolge ist nicht festgelegt, weil das Ergebnis nicht von der Reihenfolge abhängt — mehr dazu später.

Zusammengefasst ist Programmieren zu diesem Zeitpunkt das Schreiben von arithmetischen Ausdrücken. Ein Programm auszuführen bedeutet, den Wert der darin enthaltenen Ausdrücke zu berechnen. Ein Drücken auf "Start" bewirkt die Ausführung des Programms im Definitionsbereich; die Ergebnisse der Ausführung werden im Interaktionsbereich angezeigt.

Noch ein praktischer Hinweis: Wenn Sie dieses Dokument mit einem Webbrowser lesen, sollten alle Funktionen, die in den Beispielausdrücken vorkommen, einen Hyperlink zu ihrer Dokumentation enthalten. Beispielsweise sollte der Additionsoperator im Ausdruck `(+ 5 7)` einen solchen Hyperlink enthalten. Unter diesen Links finden Sie auch eine Übersicht über die weiteren Operationen, die sie verwenden können.

1.2 Arithmetik mit nicht-numerischen Werten

Wenn wir nur Programme schreiben könnten, die Zahlen verarbeiten, wäre Programmieren genau so langweilig wie Mathematik ;-) Zum Glück gibt es viele andere Arten von Werten, mit denen wir ganz analog zu Zahlen rechnen können, zum Beispiel Text, Wahrheitswerte, Bilder usw.

Zu jedem dieser sogenannten *Datentypen* gibt es *Konstruktoren*, mit denen man Werte dieser Datentypen konstruieren kann, sowie *Funktionen*, die auf Werte dieses Datentyps angewendet werden können und die weitere Werte des Datentyps konstruieren. Konstruktoren für numerische Werte sind zum Beispiel `42` oder `5.3` (also die Zahlen*literale*; Funktionen sind zum Beispiel `+` oder `*`.

Die Konstruktoren für Text (im folgenden auch *String* genannt) erkennt man an Anführungszeichen. So ist zum Beispiel

`"Konzepte der Programmiersprachen"`

ein Stringliteral. Eine Funktion auf diesem Datentyp ist `string-append`, zum Beispiel

```
> (string-append "Konzepte der " "Programmiersprachen")
"Konzepte der Programmiersprachen"
```

Es gibt weitere Funktionen auf Strings: Um Teile aus einem String zu extrahieren, um die Reihenfolge der Buchstaben umzukehren, um in Groß- oder Kleinbuchstaben zu konvertieren usw. Zusammen bilden diese Funktionen die *Arithmetik der Strings*.

Die Namen aller dieser Funktionen muss man sich nicht merken; bei Bedarf können die zur Verfügung stehenden Funktionen für Zahlen, Strings und andere Datentypen in der DrRacket Hilfe nachgeschlagen werden unter: Hilfe -> How to Design Programs Languages -> Beginning Student -> Pre-defined Functions

Bisher haben wir nur Funktionen kennengelernt, bei denen alle Argumente und auch das Ergebnis zum selben Datentyp gehören müssen. Zum Beispiel arbeitet die Funktion `+` nur mit Zahlen, und die Funktion `string-append` arbeitet nur mit Strings. Es gibt aber auch Funktionen, die Werte eines Datentyps als Argument erwarten, aber Werte eines anderen Datentypes als Ergebnis liefern, zum Beispiel die Funktion `string-length`:

```
> (+ (string-length "Programmiersprachen") 5)
24
```

Programmieren Sie einen Ausdruck, der den String `"Der Durchschnitt ist ..."` erzeugt. Statt der drei Punkte soll der Durchschnitt der Zahlen 3, 5, 19 und 32 stehen. Verwenden Sie den Ausdruck, der diesen Durchschnitt berechnet, als Unterausdruck.

Das Ergebnis von `(string-length "Programmiersprachen")` ist die Zahl `19`, die ganz normal als Argument für die Funktion `+` verwendet werden kann. Sie können also Funktionen, die zu unterschiedlichen Datentypen gehören, in einem Ausdruck zusammen verwenden. Dabei müssen Sie allerdings darauf achten, daß jede Funktion Argumente des richtigen Datentyps bekommt. Es gibt sogar Funktionen, die Argumente unterschiedlicher Datentypen erwarten, zum Beispiel

```
> (replicate 3 "hi")
"hihihi"
```

Schließlich gibt es auch Funktionen, die Datentypen ineinander umwandeln, zum Beispiel

```
> (number->string 42)
"42"
> (string->number "42")
42
```

Dieses Beispiel illustriert, dass `42` und `"42"`, trotz ihres ähnlichen Aussehens, zwei sehr unterschiedliche Ausdrücke sind. Um diese zu vergleichen, nehmen wir noch zwei weitere Ausdrücke hinzu, nämlich `(+ 21 21)` und `"(+ 21 21)"`.

Zahlen	**Strings**
`42`	`"42"`
`(+ 21 21)`	`"(+ 21 21)"`

Der erste Ausdruck, `42`, ist ein Zahl; die Auswertung einer Zahl ergibt die Zahl selber. [1] Der dritte Ausdruck, `(+ 21 21)`, ist ein Ausdruck, der bei Auswertung ebenfalls den Wert 42 ergibt. Jedes Vorkommen des Ausdrucks `42` kann in einem Programm durch den Ausdruck `(+ 21 21)` ersetzt werden (und umgekehrt), ohne die Bedeutung des Programms zu verändern.

Der zweite Ausdruck, `"42"`, ist hingegen ein String, also eine Sequenz von Zeichen, die zufällig, wenn man sie in Dezimalnotation interpretiert, dem Wert 42 entspricht. Dementsprechend ist es auch sinnlos, zu `"42"` etwas hinzuaddieren zu wollen:

```
> (+ "42" 1)
+: expects a number as 1st argument, given "42"
```

Es gibt Programmiersprachen, die automatische Konvertierungen zwischen Zahlen und Strings, die Zahlen repräsentieren, unterstützen. Dies ändert nichts daran, dass Zahlen und Strings, die als Zahlen gelesen werden können, sehr unterschiedliche Dinge sind.

Der letzte Ausdruck, `"(+ 21 21)"`, ist auch eine Sequenz von Zeichen, aber sie ist nicht äquivalent zu `"42"`. Das eine kann also nicht durch das andere ersetzt werden ohne die Bedeutung des Programms zu verändern, wie dieses Beispiel illustriert:

```
> (string-length "42")
2


> (string-length "(+ 21 21)")
9
```

Nun zurück zu unserer Vorstellung der wichtigsten Datentypen. Ein weiterer wichtiger Datentyp sind Wahrheitswerte (Boolsche Werte). Die einzigen Konstruktoren hierfür sind die Literale `#true` und `#false`. Funktionen auf boolschen Werten sind zum Beispiel die aussagenlogischen Funktionen:

```
> (and #true #true)
#true
> (and #true #false)
#false
> (or #true #false)
#true
> (or #false #false)
#false
> (not #false)
#true
```

Kennen Sie den? Frage an die schwangere Informatikern: Wird es ein Junge oder ein Mädchen? Antwort: Ja!

Boolsche Werte werden auch häufig von Vergleichsfunktionen zurückgegeben:

```
> (> 10 9)
#true
```

[1] Wenn wir ganz präzise sein wollten, könnten wir noch unterscheiden zwischen dem Zahlenliteral `42` und dem mathematischen Objekt 42; ersteres ist nur eine Notation (Syntax) für letzteres. Wir werden jedoch die Bedeutung von Programmen rein syntaktisch definieren, daher ist diese Unterscheidung für uns nicht relevant.

```
> (< -1 0)
#true
> (= 42 9)
#false
> (string=? "hello" "world")
#false
```

Natürlich können Ausdrücke weiterhin beliebig verschachtelt werden, z.B. so:

```
> (and (or (= (string-length "hello world") (string-
>number "11"))
           (string=? "hello world" "good morning"))
       (>= (+ (string-length "hello world") 60) 80))
#false
```

Beachten Sie in diesem Beispiel wie die Einrückung des Textes hilft, zu verstehen, welcher Teilausdruck Argument welcher Funktion ist. Probieren Sie in DrRacket aus, wie die Funktionen "Einrücken" bzw. "Alles einrücken" im Menü "Racket" die Einrückung ihres Programms verändern.

Der letzte Datentyp den wir heute einführen werden, sind Bilder. In BSL sind Bilder "ganz normale" Werte, mit dazugehöriger Arithmetik, also Funktionen darauf. Existierende Bilder können per Copy&Paste oder über das Menü "Einfügen -> Bild" direkt in das Programm eingefügt werden. Wenn Sie dieses Dokument im Browser betrachten, können Sie das Bild dieser Rakete  mit Copy&Paste in ihr Programm einfügen. Genau wie die Auswertung einer Zahl die Zahl selber ergibt, ergibt die Auswertung des Bilds das Bild selber.

```
>
```


Wie auf anderen Datentypen sind auch auf Bildern eine Reihe von Funktionen verfügbar. Diese Funktionen müssen allerdings erst durch das Aktivieren eines "Teachpacks" zu BSL hinzugefügt werden. Aktivieren Sie in DrRacket das HtDP/2e Teachpack "image.ss", um selber mit den folgenden Beispielen zu experimentieren.

Achten Sie darauf, das Teachpack "image.ss" zu verwenden, das zu HtDP/2e gehört. Es steht im DrRacket-Teachpack-Dialog in der mittleren Spalte. Alternativ können Sie am Anfang ihrer Datei die Anweisung `"(require 2htdp/image)"` hinzufügen.

Beispielsweise kann die Fläche des Bildes durch diesen Ausdruck berechnet werden:

```
> (* (image-width  ) (image-height  ))
600
```

Statt existierende Bilder in das Programm einzufügen kann man auch neue Bilder konstruieren:

```
> (circle 10 "solid" "red")
```


```
> (rectangle 30 20 "solid" "blue")
```

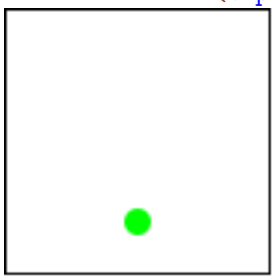

Die Arithmetik der Bilder umfasst nicht nur Funktionen um Bilder zu konstruieren, sondern auch um Bilder in verschiedener Weise zu kombinieren:

```
> (overlay (circle 5 "solid" "red")
           (rectangle 20 20 "solid" "blue"))
```

Programmieren Sie einen Ausdruck, der mehrere Kreise um denselben Mittelpunkt zeichnet. Programmieren Sie einen Ausdruck, der einen Kreis zeichnet, der durch eine schräge Linie geschnitten wird. Markieren Sie die Schnittpunkte durch kleine Kreise.

Benutzen Sie die Dokumentation von BSL (z.B. über die Links in der Browser-Version dieses Dokuments) wenn Sie wissen wollen welche weiteren Funktionen auf Bildern es gibt und welche Parameter diese erwarten. Zwei wichtige Funktionen die Sie noch kennen sollten sind `empty-scene` und `place-image`. Die erste erzeugt eine Szene, ein spezielles Rechteck in dem Bilder plaziert werden können. Die zweite Funktionen setzt ein Bild in eine Szene:

```
> (place-image (circle 5 "solid" "green")
               50 80
               (empty-scene 100 100))
```



1.3 Auftreten und Umgang mit Fehlern

Bei der Erstellung und Ausführung von Programmen können unterschiedliche Arten von Fehlern auftreten. Außerdem treten Fehler zu unterschiedlichen Zeitpunkten auf. Es ist wichtig, die Klassen und Ursachen dieser Fehler zu kennen.

Programmiersprachen unterscheiden sich darin, zu welchem Zeitpunkt Fehler gefunden werden. Die wichtigsten Zeitpunkte, die wir unterscheiden möchten, sind folgende: 1) *Vor* der Programmausführung (manchmal auch "Compile-Zeit" genannt), oder 2) *Während* der Programmausführung ("Laufzeit"). Im Allgemeinen gilt: Je früher desto besser! - allerdings muss diese zusätzliche Sicherheit häufig mit anderen Restriktionen erkauft werden, zum Beispiel der, dass einige korrekte Programme nicht mehr ausgeführt werden können.

Eine wichtige Art von Fehlern sind *Syntaxfehler*. Ein Beispiel für ein Programm mit einem Syntaxfehler sind die Ausdrücke (+ 2 3(oder (+ 2 3 oder (+ 2 (+ 3 4).

Syntaxfehler werden vor der Programmausführung von DrRacket geprüft und gefunden; diese Prüfung kann auch mit der Schaltfläche "Syntaxprüfung" veranlasst werden.

Ein Syntaxfehler tritt auf, wenn ein BSL Programm nicht zur BSL Grammatik passt. Später werden wir diese Grammatik genau definieren; informell ist eine Grammatik eine Menge von Vorschriften über die Struktur korrekter Programme.

Die Grammatik von dem Teil von BSL, den sie bereits kennen, ist sehr einfach:

- Ein BSL Programm ist eine Sequenz von Ausdrücken.
- Ein Ausdruck ist eine Zahl, ein Bild, ein Boolscher Wert, ein String, oder ein Funktionsaufruf.
- Ein Funktionsaufruf hat die Form `(f a1 a2 ...)` wobei `f` der name einer Funktion ist und die Argumente `a1`,`a2`,... Ausdrücke sind.

In BSL werden Syntaxfehler immer *vor* der Programmausführung erkannt. Auch einige andere Fehler, wie das Aufrufen einer Funktion, die nicht definiert ist, werden vor der Programmausführung erkannt, beispielsweise ein Aufruf `(foo "asdf")`. Dies ist jedoch kein Syntaxfehler (es ist leider etwas verwirrend dass diese Fehler dennoch durch Drücken des Knopfs "Check Syntax" gefunden werden).

Die folgenden Programme sind alle syntaktisch korrekt, allerdings lassen sich nicht alle diese Programme auswerten:

```
> (number->string "asdf")
number->string: expects a number, given "asdf"
> (string-length "asdf" "fdsa")
string-length: expects only 1 argument, but found 2
> (/ 1 0)
/: division by zero
> (string->number "asdf")
#false
```

Nicht jedes syntaktische korrekte Programm hat in BSL eine Bedeutung. *Bedeutung* heißt in diesem Fall dass das Programm korrekt ausgeführt werden kann und einen Wert zurückliefert. Die Menge der BSL Programme, die eine Bedeutung haben, ist nur eine *Teilmenge* der syntaktisch korrekten BSL Programme.

Die Ausführung des Programms `(number->string "asdf")` ergibt einen Laufzeitfehler, also ein Fehler der auftritt während das Programm läuft (im Unterschied zu Syntaxfehlern, die *vor* der Programmausführung erkannt werden). Wenn in BSL ein Laufzeitfehler auftritt, wird die Programmausführung abgebrochen und eine Fehlermeldung ausgegeben.

Dieser Fehler ist ein Beispiel für einen *Typfehler*: Die Funktion erwartet, dass ein Argument einen bestimmten Typ hat, diesem Fall 'Zahl', aber tatsächlich hat das Argument einen anderen Typ, in diesem Fall 'String'.

Ein anderer Fehler, der auftreten kann, ist der, dass die Anzahl der angegebenen Argumente nicht zu der Funktion passt (ein *Aritätsfehler*). Im Programm `(string-length "asdf" "fdsa")` tritt dieser Fehler auf.

Manchmal stimmt zwar der Datentyp des Arguments, aber trotzdem 'passt' der Argument in irgendeiner Weise nicht. Im Beispiel `(/ 1 0)` ist es so, dass die Divionsfunktion als Argumente Zahlen erwartet. Das zweite Argument ist eine Zahl, trotzdem

resultiert die Ausführung in einer Fehlermeldung, denn die Division durch Null ist nicht definiert.

Typfehler, Aritätsfehler und viele andere Arten von Fehlern werden erst zur Laufzeit erkannt. Eigentlich wäre es viel besser, auch diese Fehler schon vor der Programmausführung zu entdecken.

Recherchieren Sie, was das Theorem von Rice aussagt.

Zwar gibt es Programmiersprachen, die mehr Fehlerarten bereits vor der Programmausführung erkennen (insbesondere solche mit sogenanntem "statischen Typsystem"), aber es gibt einige fundamentale Grenzen der Berechenbarkeit, die dafür sorgen, dass es in jeder ausreichend mächtigen ("Turing-vollständigen") Sprache immer auch Fehler gibt, die erst zur Laufzeit gefunden werden.

Nicht alle Laufzeitfehler führen zu einem Abbruch der Programmausführung. Beispielsweise ergibt die Ausführung des Programms `(string->number "asdf")` den Wert `#false`. Dieser Wert signalisiert, dass der übergebene String keine Zahl repräsentiert. In diesem Fall tritt also *kein* Laufzeitfehler auf, sondern die Ausführung wird fortgesetzt. Der Aufrufer von `(string->number "asdf")` hat dadurch die Möglichkeit, abzufragen, ob die Umwandlung erfolgreich war oder nicht, und je nachdem das Programm anders fortzusetzen. Das Programm ist also aus BSL-Sicht wohldefiniert. Die Funktion `string->number` hätte alternativ aber auch so definiert werden können, dass sie in dieser Situation einen Laufzeitfehler auslöst.

1.4 Kommentare

Ein Programm kann neben dem eigentlichen Programmtext auch Kommentare enthalten. Kommentare haben keinen Einfluss auf die Bedeutung eines Programms und dienen nur der besseren Lesbarkeit eines Programms. Insbesondere wenn Programme größer werden können Kommentare helfen das Programm schneller zu verstehen. Kommentare werden durch ein Semikolon eingeleitet; alles was in einer Zeile nach einem Semikolon steht ist ein Kommentar.

```
; Berechnet die Goldene Zahl
(/ (+ 1 (sqrt 5)) 2)
```

Es kann auch manchmal hilfreich sein, ganze Programme "auszukommentieren". Programme, welche in Kommentaren enthalten sind werden von DrRacket ignoriert und nicht ausgewertet.

```
; (/ (+ 1 (sqrt 5)) 2)
; (+ 42
```

Im obigen Beispiel würde kein Ergebnis ausgegeben werden, da die Berechnungen in einem Kommentar steht. Fehlerhafte Teile eines Programms (wie jenes in der zweiten Kommentarzeile), können auskommentiert werden, um den Rest des Programms in DrRacket ausführen zu können.

Wir werden später mehr dazu sagen, wo, wie und wie ausführlich Programme kommentiert werden sollten.

1.5 Bedeutung von BSL Ausdrücken

Fassen wir nochmal den jetzigen Stand zusammen: Programmieren ist das Aufschreiben arithmetischer Ausdrücke, wobei die Arithmetik Zahlen, Strings, Boolsche Werte und Bilder umfasst. Programme sind syntaktisch korrekt wenn Sie gemäß der Regeln aus dem vorherigen Abschnitt konstruiert wurden. Nur syntaktisch korrekte Programme (aber nicht alle) haben in BSL eine Bedeutung. Die Bedeutung eines Ausdrucks in BSL ist ein Wert, und dieser Wert wird durch folgende Auswertungsvorschriften ermittelt:

- Zahlen, Strings, Bilder und Wahrheitswerte sind Werte. Wir benutzen im Rest dieser Vorschrift Varianten des Buchstaben *v* als Platzhalter für Ausdrücke die Werte sind und Varianten des Buchstaben *e* für beliebige Ausdrücke (Merkhilfe: Wert = *v*alue, Ausdruck = *e*xpression).
- Ist der Ausdruck bereits ein Wert so ist seine Bedeutung dieser Wert.
- Hat der Ausdruck die Form `(f` e_1 `...` e_n`)`, wobei `f` ein Funktionsname und e_1,..., e_n Ausdrücke sind, so wird dieser Ausdruck wie folgt ausgewertet:
  - Falls e_1,..., e_n bereits Werte v_1,...,v_n sind und `f` auf den Werten v_1,...,v_n definiert ist, so so ist der Wert des Ausdrucks die Anwendung von `f` auf v_1,...,v_n
  - Falls e_1,...,e_n bereits Werte v_1,...,v_n sind aber `f` ist *nicht* auf den Werten v_1,...,v_n definiert, dann wird die Auswertung mit einer passenden Fehlermeldung abgebrochen.
  - Falls mindestens eines der Argumente noch kein Wert ist, so werden durch Anwendung der gleichen Auswertungsvorschriften die Werte aller Argumente bestimmt, so dass dann die vorherige Auswertungsregel anwendbar ist.

Diese Vorschriften können wir als Anleitung zur schrittweisen *Reduktion* von Programmen auffassen. Wir schreiben $e \rightarrow e'$ um zu sagen, dass e in einem Schritt zu e' reduziert werden kann. Werte können nicht reduziert werden. Die Reduktion ist wie folgt definiert:

- Falls der Ausdruck die Form `(f` v_1 `...` v_n`)` hat und die Anwendung von `f` auf `v1`,...,`vN` den Wert `v` ergibt, dann `(f` v_1 `...` v_n`)` $\rightarrow$ `v`.
- Falls ein Audruck e_1 einen Unterausdruck e_2 in einer *Auswertungsposition* enthält, der reduziert werden kann, also $e_2 \rightarrow e_2'$, dann gilt $e_1 \rightarrow e_1'$, wobei e_1' aus e_1 entsteht indem e_2 durch e_2' ersetzt wird.

Experimentieren Sie in DrRacket mit dem "Stepper" Knopf: Geben Sie einen komplexen Ausdruck in den Definitionsbereich ein, drücken Sie auf "Stepper" und dann auf die "Schritt nach rechts" Taste und beobachten was passiert.

Die letzte Regel nennen wir die *Kongruenzregel*. In dem Teil der Sprache, den Sie bereits kennengelernt haben, sind *alle* Positionen von Unterausdrücken Auswertungspositionen. Da bisher nur Funktionsaufrufe Unterausdrücke haben, daher gilt in diesem Fall folgender Spezialfall der Kongruenzregel: Fall $e_i \rightarrow e_i'$, dann `(f` e_1 `...` e_n`)` $\rightarrow$ `(f` e_1 `...` e_{i-1} e_i' e_{i+1} `...`).

Wir benutzen folgende Konventionen:

- $e_1 \rightarrow e_2 \rightarrow e_3$ ist eine Kurzschreibweise für $e_1 \rightarrow e_2$ und $e_2 \rightarrow e_3$
- Wenn wir schreiben $e \rightarrow^* e$' so bedeutet dies dass entweder $e = e$' gilt, oder es ein n $\geqslant 0$ und e_1, ...,e_n gibt so dass $e \rightarrow e_1 \rightarrow ... \rightarrow e_n \rightarrow e$' gilt. Insbesondere gilt also $e \rightarrow^* e$ (Reflexivität), aus $e \rightarrow^* e$' und e' $\rightarrow^* e$'' folgt $e \rightarrow^* e$'' (Transitivität), und aus $e \rightarrow e$' folgt $e \rightarrow^* e$' (Inklusion). Man sagt, $\rightarrow^*$ ist der reflexive und transitive Abschluss von $\rightarrow$.

Beispiele:

- `(+ 1 1)` $\rightarrow$ `2`.
- `(+ (* 2 3) (* 4 5))` $\rightarrow$ `(+ 6 (* 4 5))` $\rightarrow$ `(+ 6 20)` $\rightarrow$ `26`.
- `(+ (* 2 3) (* 4 5))` $\rightarrow$ `(+ (* 2 3) 20)` $\rightarrow$ `(+ 6 20)` $\rightarrow$ `26`.
- `(+ (* 2 (+ 1 2)) (* 4 5))` $\rightarrow$ `(+ (* 2 3) (* 4 5))`.
- `(+ (* 2 3) (* 4 5))` $\rightarrow^*$ `26` aber nicht `(+ (* 2 3) (* 4 5))` $\rightarrow$ `26`.
- `(+ 1 1)` $\rightarrow^*$ `(+ 1 1)` aber nicht `(+ 1 1)` $\rightarrow$ `(+ 1 1)`.
- `(overlay (circle 5 "solid" "red") (rectangle 20 20 "solid" "blue"))` $\rightarrow$ `(overlay`  `(rectangle 20 20 "solid" "blue"))` $\rightarrow$ `(overlay`  `)` $\rightarrow$  .

Im Allgemeinen kann ein Ausdruck mehrere reduzierbare Unterausdrücke haben, also die Kongruenzregeln an mehreren Stellen gleichzeitig einsetzbar sein. In den Beispielen oben haben wir zum Beispiel `(+ (* 2 3) (* 4 5))` $\rightarrow$ `(+ (* 2 3) 20)` aber auch `(+ (* 2 3) (* 4 5))` $\rightarrow$ `(+ 6 (* 4 5))`. Es ist jedoch nicht schwer zu sehen, dass immer wenn wir die Situation $e_1 \rightarrow$ e2 und $e_1 \rightarrow e_3$ haben, dann gibt es ein e_4 so dass gilt $e_2 \rightarrow^* e_4$ und $e_3 \rightarrow^* e_4$. Diese Eigenschaft nennt man *Konfluenz*. Reduktionen die auseinanderlaufen können also immer wieder zusammengeführt werden; der Wert den man am Ende erhält ist auf jeden Fall eindeutig.

Auf Basis dieser Reduktionsregeln können wir nun definieren, unter welchen Umständen zwei Programme äquivalent sind: Zwei Ausdrücke e_1 und e_2 sind äquivalent, geschrieben $e_1 \equiv e_2$, falls es einen Ausdruck e gibt so dass $e_1 \rightarrow^* e$ und $e_2 \rightarrow^* e$.

Beispiele:

- `(+ 1 1)` $\equiv$ `2`.
- `(+ (* 2 3) 20)` $\equiv$ `(+ 6 (* 4 5))` $\equiv$ `26`.
- `(overlay`  `(rectangle 20 20 "solid" "blue"))` $\equiv$ `(overlay (circle 5 "solid" "red")`  `)`.

Die Rechtfertigung für diese Definition liegt in folgender wichtiger Eigenschaft begründet: Wir können innerhalb eines großen Programms Teilausdrücke beliebig durch äquivalente Teilausdrücke ersetzen, ohne die Bedeutung des Gesamtprogramms zu verändern. Etwas formaler können wir das so ausdrücken:

Sei e_1 ein Unterausdruck eines größeren Ausdrucks e_2 und $e_1 \equiv e_3$. Ferner sei e_2' eine Kopie von e_2 in dem Vorkommen von e_1 durch e_3 ersetzt wurden. Dann gilt: $e_2 \equiv e_2$'.

Diese Eigenschaft folgt direkt aus der Kongruenzregel und der Definition von $\equiv$. Dieser Äquivalenzbegriff ist identisch mit dem, den Sie aus der Schulmathematik kennen, wenn Sie Gleichungen umformen, zum Beispiel wenn wir a + a - b umformen zu 2a - b weil wir wissen dass a + a = 2a.

Die Benutzung von $\equiv$ um zu zeigen dass Programme die gleiche Bedeutung haben nennt man auch *equational reasoning*.

2 Programmierer entwerfen Sprachen!

Dieser Teil des Skripts basiert auf [HTDP/2e] Kapitel 1

Die Programme, die Sie bisher geschrieben haben, waren im Wesentlichen Berechnungen, wie man sie auch auf einem Taschenrechner durchführen könnte — allerdings mit dem großen Unterschied dass BSL die Arithmetik von vielen Arten von Werten beherrscht und nicht nur die der Zahlen. Bei den Funktionen, die Sie in diesen Ausdrücken verwenden können, können Sie aus einer festen Menge vordefinierter Funktionen wählen. Das *Vokabular* (Menge der Funktionen), welches sie verwenden können, ist also fix.

Eine echte Programmiersprache unterscheidet sich von einem Taschenrechner dadurch, dass Sie selber, auf Basis der bereits bestehenden Funktionen, neue Funktionen definieren können und diese danach in Ausdrücken und Definitionen neuer Funktionen benutzen können. Im Allgemeinen können Sie *Namen* definieren und damit das Vokabular, welches Ihnen (und, bei Veröffentlichung ihrer Programme auch anderen) zur Verfügung steht, selber erweitern. Ein Programm ist also mehr als eine Menge von Maschineninstruktionen - es definiert auch eine *Sprache* für die Domäne des Programms. Sie werden sehen, dass die Möglichkeit, neue Namen zu definieren, das Schlüsselkonzept ist, um mit der Komplexität großer Softwaresysteme umzugehen.

2.1 Funktionsdefinitionen

Hier ist die Definition einer Funktion die den Durchschnitt von zwei Zahlen berechnet.

```
(define (average x y) (/ (+ x y) 2))
```

Wenn diese Funktionsdefinition in den Definitionsbereich geschrieben und dann auf "Start" gedrückt wird, passiert — nichts. Eine Funktionsdefinition ist *kein* Ausdruck. Der Effekt dieser Definition ist der, dass der Name der neuen Funktion nun in Ausdrücken verwendet werden kann:

```
> (average 12 17)
14.5
> (average 100 200)
150
```

Diese Werte hätten natürlich auch ohne die vorherige Funktionsdefinition berechnet werden können:

```
> (/ (+ 12 17) 2)
14.5
> (/ (+ 100 200) 2)
150
```

Allerdings sehen wir, dass die zweite Variante redundant ist: Der Algorithmus zur Berechnung des Durchschnitts wurde zweimal repliziert. Sie ist auch weniger abstrakt und weniger leicht verständlich, denn wir müssen erst verstehen, dass der Algorithmus den Durchschnitt zweier Zahlen berechnet, während wir in der ersten Version dem Algorithmus einen *Namen* gegeben haben, der die Details des Algorithmus verbirgt.

Gute Programmierer versuchen im Allgemeinen, jede Art von Redundanz in Programmen zu vermeiden. Dies wird manchmal als das DRY (Don't repeat yourself) Prinzip bezeichnet. Der Grund dafür ist nicht nur, dass man Schreibarbeit spart, sondern auch dass redundante Programme schwerer zu verstehen und zu warten sind: Wenn ich einen Algorithmus später ändern möchte, so muss ich in einem redundanten Programm erst alle Kopien dieses Algorithmus finden und jede davon ändern. Daher ist Programmieren niemals eine monotone repetitive Tätigkeit, denn wiederkehrende Muster können in Funktionsdefinitionen (und anderen Formen der Abstraktion die sie noch kennenlernen werden) gekapselt und wiederverwendet werden.

Im Allgemeinen Fall haben Funktionsdefinitionen diese Form:

```
(define (FunctionName InputName1 InputName2 ...) BodyExpression)
```

Funktionsdefinitionen sind *keine* Ausdrücke sondern eine neue Kategorie von Programmen. Funktionsdefinitionen dürfen also beispielsweise nicht als Argument von Funktionen verwendet werden. Eine Funktionsdefinition startet mit dem Schlüsselwort `define`. Der einzige Zweck dieses Schlüsselworts ist der, Funktionsdefinitionen von Ausdrücken unterscheiden zu können. Insbesondere darf es also keine Funktionen geben, die `define` heißen. `FunctionName` ist der Name der Funktion. Diesen benötigt man, um die Funktion in Ausdrücken benutzen (oder: *aufrufen*) zu können. `InputName1`, `InputName2` und so weiter sind die *Parameter* der Funktion. Die Parameter repräsentieren die Eingabe der Funktion, die erst bekannt wird wenn die Funktion aufgerufen wird. Die `BodyExpression` ist ein Ausdruck der die Ausgabe der Funktion definiert. Innerhalb der `BodyExpression` werden in der Regel die Parameter der Funktion benutzt. Wir nennen `BodyExpression` die *Implementation* der Funktion oder den *Body* der Funktion.

Funktionsaufrufe haben die Form:

```
(FunctionName ArgumentExpression1 ArgumentExpression1 ...)
```

Ein Funktionsaufruf einer mit `define` definierten (*benutzerdefinierte*) Funktion sieht also genau so aus wie das Benutzen einer fest eingebauten (*primitiven*) Funktion. Dies ist kein Zufall. Dadurch, dass man nicht sehen kann, ob man gerade eine primitive Funktion oder eine benutzerdefinierte Funktion aufruft, ist es leichter, die Programmiersprache selber zu erweitern oder zu verändern. Zum Beispiel kann aus einer primitiven Funktion eine benutzerdefinierte Funktion gemacht werden, oder ein Programmierer kann Erweiterungen definieren die so aussehen, als wäre die Sprache um primitive Funktionen erweitert worden.

2.2 Funktionen die Bilder produzieren

Selbstverständlich können in BSL Funktionen nicht nur Zahlen sondern beliebige Werte als Eingabe bekommen oder als Ausgabe zurückliefern. In Abschnitt §1.2 "Arithmetik mit nicht-numerischen Werten" haben Sie gesehen, wie man mit `place-image` ein Bild in einer Szene plaziert. Zum Beispiel erzeugen die drei Ausdrücke

```
(place-image  50 20 (empty-scene 100 100))
(place-image  50 40 (empty-scene 100 100))
(place-image  50 60 (empty-scene 100 100))
```

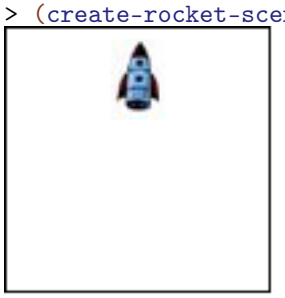
die Bilder

Offensichtlich sind diese drei Ausdrücke zusammen redundant, denn sie unterscheiden sich nur in dem Parameter für die Höhe der Rakete. Mit einer Funktionsdefinition können wir das Muster, welches diese drei Ausdrücke gemein haben, ausdrücken:

```
(define (create-rocket-scene height)
  (place-image  50 height (empty-scene 100 100)))
```

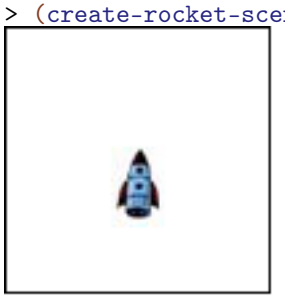
Die drei Bilder können nun durch Aufrufe der Funktion erzeugt werden.

```
> (create-rocket-scene 20)
```



```
> (create-rocket-scene 40)
```

```
> (create-rocket-scene 60)
```



Sie können den Höhenparameter auch als Zeitparameter auffassen; die Funktion bildet also Zeitpunkte auf Bilder ab. Solche Funktionen können wir auch als Film oder Animation auffassen, denn ein Film ist dadurch charakterisiert, dass es zu jedem Zeitpunkt ein dazugehöriges Bild gibt.
Im Teachpack `universe` gibt es eine Funktion, die den Film, der zu einer solchen Funktion korrespondiert, auf dem Bildschirm vorführt. Diese Funktion heißt `animate`. Die Auswertung des Ausdrucks

Ein Teachpack ist eine Bibliothek mit Funktionen die sie in ihrem Programm verwenden können. Sie können ein Teachpack aktivieren, indem Sie `(require 2htdp/universe)` an den Anfang Ihrer Datei hinzufügen.

```
(animate create-rocket-scence)
```

bewirkt, dass ein neues Fenster geöffnet wird in dem eine Animation zu sehen ist, die zeigt, wie sich die Rakete von oben nach unten bewegt und schließlich verschwindet. Wenn sie das Fenster schließen wird eine Zahl im Interaktionsbereich angezeigt; diese Zahl steht für die aktuelle Höhe der Rakete zu dem Zeitpunkt als das Fenster geschlossen wurde.

Die `animate` Funktion bewirkt folgendes: Eine Stoppuhr wird mit dem Wert 0 initialisiert; 28 mal pro Sekunde wird der Zählerwert um eins erhöht. Jedesmal wenn der Zähler um eins erhöht wird, wird die Funktion `create-rocket-scene` ausgewertet und das resultierende Bild in dem Fenster angezeigt.

2.3 Bedeutung von Funktionsdefinitionen

Um die Bedeutung von Funktionsdefinitionen zu definieren, müssen wir sagen, wie Funktionsdefinitionen und Funktionsaufrufe ausgewertet werden. Durch Funktionsdefinitionen können Ausdrücke nicht mehr isoliert (ohne Berücksichtigung des Rests des Programms) ausgewertet werden; sie werden im *Kontext* einer Menge von Funktionsdefinitionen ausgewertet. Dieser Kontext umfasst die Menge aller Funktionsdefinitionen, die im Programmtext *vor* dem auszuwertenden Ausdruck stehen. Um unsere

formale Notation nicht unnötig schwergewichtig zu machen, werden wir diesen Kontext nicht explizit zur Reduktionsrelation hinzufügen; stattdessen gehen wir einfach davon aus, dass es einen globalen Kontext mit einer Menge von Funktionsdefinitionen gibt.

Die Auswertungsregeln aus §1.5 “Bedeutung von BSL Ausdrücken” werden nun zur Berücksichtigung von Funktionsdefinitionen wie folgt erweitert:

- Falls der Ausdruck die Form `(f` v_1 `...` v_n`)` hat und `f` eine primitive (eingebaute) Funktion ist und die Anwendung von `f` auf v_1,...,v_n den Wert `v` ergibt, dann `(f` v_1 `...` v_n`)` → `v`.
- Falls der Ausdruck die Form `(f` v_1 `...` v_n`)` hat und `f` *keine* primitive Funktion ist und der Kontext die Funktionsdefinition

  ```
  (define (f x1 ... xn) BodyExpression)
  ```

  enthält, dann `(f` v_1 `...` v_n`)` → `NewBody`, wobei `NewBody` aus `BodyExpression` entsteht, indem man alle Vorkommen von x_i durch v_i ersetzt (für i=1...n).

Wie wird der Ausdruck `(* (- 1 1) (+ 2 3))` ausgewert? Wieviele Schritte werden benötigt? Wie wird der Ausdruck `(and (= 1 2) (= 3 3))` ausgewertet? Wieviele Schritte werden benötigt? Können Sie eine Funktion `mul` programmieren, die dasselbe wie `*` berechnet, aber (ähnlich wie `and`) weniger Schritte benötigt?

Unverändert gilt die Kongruenzregel aus §1.5 “Bedeutung von BSL Ausdrücken”.

Beispiel: Unser Programm enthält folgende Funktionsdefinitionen.

```
(define (g y z) (+ (f y) y (f z)))
(define (f x) (* x 2))
```

Dann `(g (+ 2 3) 4)` → `(g 5 4)` → `(+ (f 5) 5 (f 4))` → `(+ (* 5 2) 5 (f 4))` → `(+ 10 5 (f 4))` → `(+ 10 5 8)` → `23`

Beachten Sie, dass während der gesamten Reduktion der Kontext sowohl `f` als auch `g` enthält, daher ist ist kein Problem, dass im Body von `g` die Funktion `f` aufgerufen wird, obwohl `f` erst *nach* `g` definiert wird. Die Auswertung des Programms

```
(define (g y z) (+ (f y) y (f z)))
(g (+ 2 3) 4)
(define (f x) (* x 2))
```

schlägt hingegen fehl, weil der Kontext bei der Auswertung von `(g (+ 2 3) 4)` nur `g` aber nicht `f` enthält.

2.4 Konditionale Ausdrücke

In der Animation aus dem letzten Abschnitt verschwindet die Rakete einfach irgendwann nach unten aus dem Bild. Wie können wir es erreichen, dass die Rakete stattdessen auf dem Boden der Szene "landet"?

2.4.1 Motivation

Offensichtlich benötigen wir hierzu in unserem Programm eine Fallunterscheidung. Fallunterscheidungen kennen sie aus zahlreichen Beispielen des realen Lebens.

Beispielsweise ist das Bewertungsschema für eine Klausur, welches jeder Punktzahl eine Note zuordnet, eine Funktion die die unterschiedlichen Grenzen für die resultierenden Noten voneinander unterscheidet. In BSL können wir ein Notenschema, bei dem man mindestens 90 Punkte für eine 1 benötigt und alle 10 Punkte darunter eine Note heruntergegangen wird, wie folgt definieren:

```
(define (note punkte)
  (cond
    [(>= punkte 90) 1]
    [(>= punkte 80) 2]
    [(>= punkte 70) 3]
    [(>= punkte 60) 4]
    [(>= punkte 50) 5]
    [(< punkte 50) 6]))
```

Einige Beispiele für die Benutzung des Notenschemas:

```
> (note 95)
1
> (note 73)
3
> (note 24)
6
```

2.4.2 Bedeutung konditionaler Ausdrücke

Im allgemeinen Fall sieht ein konditionaler Ausdruck wie folgt aus:

```
(cond
  [ConditionExpression1 ResultExpression1]
  [ConditionExpression2 ResultExpression2]
  ....
  [ConditionexpressionN ResultExpressionN])
```

Ein konditionaler Ausdruck startet also mit einer öffnenden Klammer und dem Schlüsselwort `cond`. Danach folgen beliebig viele Zeilen, von denen jede zwei Ausdrücke beinhaltet. Der linke Ausdruck wird die *Bedingung* oder *Kondition* und der rechte das *Resultat* genannt.

Ein `cond` Ausdruck wird wie folgt ausgewertet. DrRacket wertet zunächst die erste Bedingung `ConditionExpression1` aus. Ergibt diese Auswertung den Wert `#true`, so ist der Wert des gesamten `cond` Ausdrucks der Wert von `ResultExpression1`. Ergibt diese Auswertung hingegen den Wert `#false`, so wird mit der zweiten Zeile fortgefahren und genau so verfahren wie mit der ersten Zeile. Wenn es keine nächste Zeile mehr gibt — also alle Bedingungen zu `#false` ausgewertet wurden — so wird mit einer Fehlermeldung abgebrochen. Ebenso ist es ein Fehler, wenn die Auswertung einer Bedingung nicht `#true` oder `#false` ergibt:

```
> (cond [(< 5 3) 77]
        [(> 2 9) 88])
cond: all question results were false
> (cond [(+ 2 3) 4])
cond: question result is not true or false: 5
```

Die Reduktionsregeln für BSL müssen wir zur Berücksichtigung konditionaler Ausdrücke um folgende beiden Regeln ergänzen:

(cond [#false e] [e_2 e_3] .... [e_{n-1} e_n]) → (cond [e_2 e_3] .... [e_{n-1} e_n])

und

(cond [#true e] [e_2 e_3] .... [e_{n-1} e_n]) → e

Außerdem ergänzen wir die Auswertungspositionen, die in der Kongruenzregel verwendet werden können wie folgt: In einem Ausdruck der Form

```
(cond [e0 e1]
      [e2 e3]
      ....
      [en-1 en])
```

ist der Ausdruck e_0 in einer Auswertungsposition, aber nicht e_1,...,e_n .

Beispiel: Betrachten wir den Aufruf (note 83) in dem Beispiel oben. Dann (note 83) → (cond [(>= 83 90) 1] [(>= 83 80) 2] ...) → (cond [#false 1] [(>= 83 80) 2] ...) → (cond [(>= 83 80) 2] ...) → (cond [#true 2] ...) → 2

2.4.3 Beispiel

Zurück zu unserer Rakete. Offensichtlich müssen wir hier zwei Fälle unterscheiden. Während die Rakete noch oberhalb des Bodens der Szene ist, soll sie wie gehabt sinken. Wenn die Rakete allerdings bereits auf dem Boden angekommen ist, soll die Rakete nicht mehr weiter sinken.

Da die Szene 100 Pixel hoch ist, können wir die Fälle unterscheiden, dass die aktuelle Höhe kleiner oder gleich 100 ist und dass die aktuelle Höhe größer als 100 ist.

Für die Varianten der `create-rocket-scence` Funktion verwenden wir die Namenskonvention dass wir den Varianten die Suffixe `-v2`, `-v3` usw. geben.

```
(define (create-rocket-scene-v2 height)
  (cond
    [(<= height 100)
     (place-image  50 height (empty-scene 100 100))]
    [(> height 100)
     (place-image  50 100 (empty-scene 100 100))]))
```

2.4.4 Etwas syntaktischer Zucker...

Zwei Spezialfälle konditionaler Ausdrücke sind so häufig, dass es in BSL eine eigene Syntax dafür gibt, die für diese Spezialfälle optimiert ist.

Der erste Spezialfall ist der, dass man einen Zweig der Kondition haben möchte, der immer dann genommen wird, wenn alle anderen Zweige nicht anwendbar sind. In diesem Fall kann man statt der Kondition das Schlüsselword `else` verwenden. Das Beispiel von oben könnten wir daher auch so formulieren:

```
(define (note punkte)
  (cond
    [(>= punkte 90) 1]
    [(>= punkte 80) 2]
    [(>= punkte 70) 3]
    [(>= punkte 60) 4]
    [(>= punkte 50) 5]
    [else 6]))
```

Die `else` Klausel allerdings darf nur im letzten Zweig eines `cond` Ausdrucks verwendet werden:

```
> (cond [(> 3 2) 5]
        [else 7]
        [(< 2 1) 13])
```
cond: found an else clause that isn't the last clause in its cond expression

Der `else` Zwei ist äquivalent zu einem Zweig mit der immer erfüllten Bedingung `#true`, daher ist im Allgemeinen Fall die Bedeutung von

```
(cond [e0 e1]
      [e2 e3]
      ....
      [else en])
```

definiert als die Bedeutung von

```
(cond [e0 e1]
      [e2 e3]
      ....
      [#true en])
```

Wir geben also in diesem Fall keine Reduktionsregeln für dieses Sprachkonstrukt an, sondern stattdessen eine Transformation, die die Bedeutung transformiert. Wenn Sprachkonstrukte "nichts neues" hinzufügen sondern lediglich eine Abkürzung für eine bestimmte Benutzung bestehender Sprachkonstrukte sind, so nennt man solche Sprachkonstrukte auch *syntaktischen Zucker*.

Ein anderer Spezialfall konditionaler Ausdrücke ist der, dass es nur eine Bedingung gibt, die überprüft werden soll, und je nachdem ob diese Bedingung wahr oder falsch ist soll ein anderer Ausdruck ausgewählt werden. Für diesen Fall gibt es das `if` Konstrukt.

Beispiel:

```
(define (aggregatzustand temperatur)
  (if (< temperatur 0) "gefroren" "flüssig"))


> (aggregatzustand -5)
"gefroren"
```

Im Allgemeinen hat ein `if` Ausdruck folgende Form:

```
(if CondExpression ThenExpression ElseExpression)
```

Ein `if` Ausdruck ist syntaktischer Zucker; die Bedeutung wird durch die Transformation in diesen Ausdruck festgelegt:

```
(cond [CondExpression ThenExpression]
      [else ElseExpression])
```

Im Allgemeinen eignet sich `if` für Situationen, in denen wir so etwas wie "entweder das eine oder das andere" sagen wollen. Die `cond` Ausdrücke eignen sich dann, wenn man mehr als zwei Situationen unterscheiden möchten.

Obwohl es zunächst so aussieht, als sei `if` ein Spezialfall von `cond`, kann man allerdings auch jeden `cond` Ausdruck durch einen geschachtelten `if` Ausdruck ersetzen. Beispielsweise kann die Funktion von oben auch so geschrieben werden:

```
(define (note punkte)
  (if (>= punkte 90)
      1
     (if (>= punkte 80)
         2
         (if (>= punkte 70)
             3
             (if (>= punkte 60)
                 4
                 (if (>= punkte 50)
                     5
                     6))))))
```

In solchen Fällen ist offensichtlich das `cond` Konstrukt besser geeignet, weil man keine tief geschachtelten Ausdrücke benötigt. Dennoch kann man festhalten, dass `cond` und `if` gleichmächtig sind, weil das eine in das andere so transformiert werden kann, dass die Bedeutung gleich bleibt.

2.4.5 Auswertung konditionaler Ausdrücke

In §2.4.2 “Bedeutung konditionaler Ausdrücke” haben wir definiert, dass in einem konditionalen Ausdruck

```
(cond [e0  e1]
      [e2  e3]
      ....
      [en-1  en])
```

nur der Ausdruck e_0 in einer Auswertungsposition ist, aber nicht e_1,...,e_n . Wieso diese Einschränkung — wieso nicht auch die Auswertung von e_1,...,e_n erlauben? Betrachten Sie folgendes Beispiel:

```
(cond [(= 5 7) (/ 1 0)]
      [(= 3 3) 42]
      [(/ 1 0) 17])
```

Gemäß unserer Auswertungsregeln gilt:

```
(cond [(= 5 7) (/ 1 0)]
      [(= 3 3) 42]
      [(/ 1 0) 17])

      →

(cond [#false (/ 1 0)]
      [(= 3 3) 42]
      [(/ 1 0) 17])

      →

(cond [(= 3 3) 42]
      [(/ 1 0) 17])

      →

(cond [#true 42]
      [(/ 1 0) 17])

      →

42
```

Wenn es erlaubt wäre, auch auf den anderen Positionen auszuwerten, müssten wir gemäß unserer Regeln die Berechnung in dem Beispiel mit einem Fehler abbrechen, sobald wir einen der `(/ 1 0)` Ausdrücke auswerten. Gemäß der Terminologie aus §1.5 “Bedeutung von BSL Ausdrücken” geht uns die Konfluenz-Eigenschaft verloren und der Wert eines Ausdrucks ist nicht mehr eindeutig.

Das Beispiel oben ist sehr künstlich, aber wir werden später sehen, dass konditionale Ausdrücke häufig verwendet werden, um sicherzustellen, dass eine Funktion terminiert – die Auswertung also nicht endlos andauert. Dafür ist es essentiell, dass nicht auf allen Positionen ausgewertet werden darf. Operatoren wie `cond`, bei denen die Auswertung beliebiger Argumente nicht erlaubt ist, nennt man auch *nicht-strikt*. Normale Funktionen, bei deren Aufruf alle Argumente ausgewertet werden bevor die Funktion angewendet wird, nennt man hingegen *strikt*.

2.4.6 In der Kürze liegt die Würze

Aus der Bedeutung konditionaler Ausdrücke können wir eine Reihe von Refactorings herleiten, die verwendet werden können (und sollten), um konditionale Ausdrücke zu vereinfachen. Unnötig lange Ausdrücke sind auch ein Verstoss gegen das DRY-Prinzip, denn auch diese sind eine Form von Redundanz. Bezüglich der Refactorings, die aus einem konditionalen Ausdruck einen boolschen Ausdruck machen, werden wir später (§8.9.4 "Bedeutung boolscher Ausdrücke") sehen, dass diese Refactorings auch bezüglich des Striktheitsverhaltens übereinstimmen.

Ausdruck		**Vereinfachter Ausdruck**	**Bedingung**
`(if e #true #false)`	≡	`e`	*e* ≡ `#true` *oder* *e* ≡ `#fa`
`(if e #false #true)`	≡	`(not e)`	–
`(if e e #false)`	≡	`e`	*e* ≡ `#true` *oder* *e* ≡ `#fa`
`(if (not e-1) e-2 e-3)`	≡	`(if e-1 e-3 e-2)`	–
`(if e-1 #true e-2)`	≡	`(or e-1 e-2)`	*e-2* ≡ `#true` *oder* *e-2* ≡
`(if e-1 e-2 #false)`	≡	`(and e-1 e-2)`	*e-2* ≡ `#true` *oder* *e-2* ≡
`(if e-1 (f ... e-2 ...) (f ... e-3 ...))`	≡	`(f ... (if e-1 e-2 e-3) ...)`	–

Ein Beispiel für das letzte Refactoring werden wir in §2.6.2 "DRY Redux" diskutieren. Später werden wir zeigen, wie man die Korrektheit dieser Refactorings beweisen kann. Informell können Sie die Äquivalenzen nachvollziehen, wenn Sie einfach mal die möglichen Fälle durchspielen und Ausdrücke durch `#true` oder `#false` ersetzen.

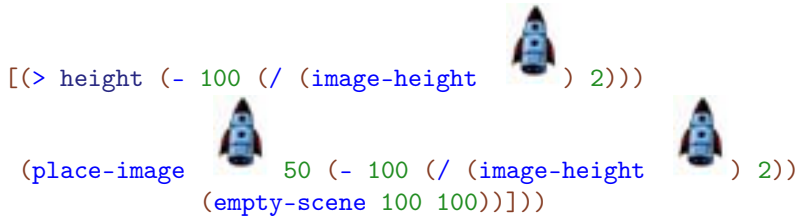
2.5 Definition von Konstanten

Wenn wir uns `(animate create-rocket-scene-v2)` anschauen, stellen wir fest, dass die Animation noch immer nicht befriedigend ist, denn die Rakete versinkt halb im Boden. Der Grund dafür ist, dass `place-image` das Zentrum des Bildes an dem vorgegebenen Punkt plaziert. Damit die Rakete sauber auf dem Boden landet, muss das Zentrum also überhalb des Bodens sein. Mit etwas Überlegung wird schnell klar, dass die Rakete nur bis zu der Höhe

```
(- 100 (/ (image-height  ) 2))
```

absinken sollte. Das bedeutet, dass wir unsere `create-rocket-scene-v2` Funktion wie folgt modifizieren müssen:

```
(define (create-rocket-scene-v3 height)
  (cond
    [(<= height (- 100 (/ (image-height  ) 2)))
     (place-image  50 height (empty-scene 100 100))]
```



2.6 DRY: Don't Repeat Yourself!

Ein Aufruf von `(animate create-rocket-scene-v3)` illustriert, dass die Rakete nun wie von uns gewünscht landet. Allerdings ist offensichtlich, dass `create-rocket-scene-v3` gegen das im Abschnitt §2.1 "Funktionsdefinitionen" angesprochene Prinzip verstößt, dass gute Programme keine Redundanz enthalten. Im Programmiererjargon wird dieses Prinzip auch häufig DRY-Prinzip — Don't Repeat Yourself — genannt.

2.6.1 DRY durch Konstantendefinitionen

Konstante Werte wie `100` in Programmtexten werden von Programmierern häufig abfällig als *magic numbers* bezeichnet.

Eine Art von Redundanz, die in `create-rocket-scene-v3` auftritt, ist die, dass die Höhe und Breite der Szene sehr häufig wiederholt wird. Stellen Sie sich vor, sie möchten statt einer 100 mal 100 Szene eine 200 mal 400 Szene haben. Zu diesem Zweck müssen Sie alle Vorkommen der alten Höhe und Breite finden, jeweils herausfinden ob sie für die Breite oder die Höhe oder noch etwas anderes stehen (deshalb ist das Problem auch nicht mit maschineller Textersetzung lösbar), und je nachdem durch den neuen Wert ersetzen. Der Aufwand ist bei `create-rocket-scene-v3` zwar noch überschaubar, aber wenn Sie Programme mit vielen tausend Codezeilen betrachten wird schnell klar, dass dies ein großes Problem ist.

Idealerweise sollte die Beziehung zwischen den Anforderungen an ein Programm und dem Programmtext *stetig* sein: Ein kleiner Änderungswunsch an den Anforderungen für ein Programm sollte auch nur eine kleine Änderung am Programmtext erfordern. In unserem konkreten Beispiel können wir dieses Problem mit `define` lösen. Mit `define` können nämlich nicht nur Funktionen, sondern auch *Konstanten* definiert werden. Beispielsweise können wir in unser Programm diese Definition hineinschreiben:

```
(define HEIGHT 100)
```

Für die Bedeutung des Programms spielt es keine Rolle dass der Name der Konstanten nur aus Großbuchstaben besteht. Dies ist lediglich eine Namenskonvention, anhand derer Programmierer leicht erkennen können, welche Namen sich auf Konstanten beziehen.

Die Bedeutung einer solchen Definition ist, dass im Rest des Programms `HEIGHT` ein gültiger Ausdruck ist, der bei Auswertung den Wert `100` hat. Wenn wir im Programm alle Vorkommen von `100`, die für die Höhe stehen, durch `HEIGHT` ersetzen, und das gleiche für `WIDTH` machen, erhalten wir diese Variante von `create-rocket-scene`:

```
(define WIDTH 100)
(define HEIGHT 100)
(define (create-rocket-scene-v4 height)
```

```
(cond
  [(<= height (- HEIGHT (/ (image-height  ) 2)))
   (place-image  50 height (empty-scene WIDTH HEIGHT))]
  [(> height (- HEIGHT (/ (image-height  ) 2)))
   (place-image  50 (- HEIGHT (/ (image-
height  ) 2))
                (empty-scene WIDTH HEIGHT))]))
```

Testen Sie durch `(animate create-rocket-scene-v4)` dass das Programm weiterhin funktioniert. Experimentieren Sie mit anderen Werten für `WIDTH` und `HEIGHT` um zu sehen, dass diese kleine Änderung wirklich genügt um die Größe der Szene zu ändern.

Im Programmiererjargon nennen sich Programmänderungen, die die Struktur des Programms verändern ohne sein Verhalten zu verändern, *Refactorings*. Häufig werden Refactorings durchgeführt um die Wartbarkeit, Lesbarkeit, oder Erweiterbarkeit des Programms zu verbessern. In unserem Fall haben wir sowohl Wartbarkeit als auch Lesbarkeit durch dieses Refactoring verbessert. Die verbesserte Wartbarkeit haben wir bereits illustriert; die verbesserte Lesbarkeit rührt daher, dass wir an Namen wie `WIDTH` die Bedeutung der Konstanten ablesen können, während wir bei magic numbers wie `100` diese Bedeutung erst durch genaue Analyse des Programms herausfinden müssen (im Programmiererjargon auch *reverse engineering* genannt).

Es spielt übrigens keine Rolle, ob die Definitionen der Konstanten oberhalb oder unterhalb der `create-rocket-scene` Definition stehen. Die Konstanten sind innerhalb der gesamten Programmdatei sichtbar. Man sagt, die Konstanten haben *globalen Scope*. Um die Definitionen der Konstanten nicht im Programmtext suchen zu müssen, ist es sinnvoll, diese immer an der gleichen Stelle zu definieren. In vielen Programmiersprachen gibt es die Konvention, dass Konstantendefinitionen immer am Anfang des Programmtextes stehen, daher werden auch wir uns an diese Konvention halten.

Allerdings verstößt `create-rocket-scene-v4` immer noch gegen das DRY-Prinzip. Beispielsweise kommt der Ausdruck

```
(- HEIGHT (/ (image-height  ) 2))
```

mehrfach vor. Diese Redundanz kann ebenfalls mit `define` beseitigt werden, denn der Wert, mit dem eine Konstante belegt wird, kann durch einen beliebig komplexen

Ausdruck beschrieben werden. Im Allgemeinen haben Konstantendefinitionen die folgende Form:

```
(define CONSTANTNAME CONSTANTExpression)
```

Im vorherigen Beispiel können wir die Konstante zum Beispiel `ROCKET-CENTER-TO-BOTTOM` nennen. Beachten Sie, wie durch die Wahl guter Namen die Bedeutung des Programms viel offensichtlicher wird. Ohne diesen Namen müssten wir jedesmal, wenn wir den komplexen Ausdruck oben lesen und verstehen wollen, wieder herausfinden, dass hier die gewünschte Distanz des Zentrums der Rakete zum Boden berechnet wird.
Das gleiche können wir mit dem mehrfach vorkommenden Ausdruck `(empty-scene WIDTH HEIGHT)` machen. Wir geben ihm den Namen `MTSCN`.

In der Tradition der Familie von Programmiersprachen, aus der BSL stammt, ist es üblich, die englische Aussprache der Buchstaben des Alphabets zu verwenden um Namen abzukürzen. `MTSCN` spricht man daher "empty scene".

Auch die Zahl `50` im Programmtext ist eine *magic number*, allerdings hat sie eine andere Qualität als `WIDTH` und `HEIGHT`: Sie ist nämlich abhängig von dem Wert anderer Konstanten, in diesem Fall `WIDTH`. Da diese Konstante für die horizontale Mitte steht, definieren wir sie als `(define MIDDLE (/ WIDTH 2))`.

Die letzte Art der Redundanz, die nun noch vorkommt, ist, dass die Rakete selber mehrfach im Programmtext vorkommt. Die Rakete ist zwar kein Zahlenliteral und daher keine *magic number*, aber ein *magic image* — mit genau den gleichen Nachteilen wie *magic numbers*. Daher definieren wir auch für das Bild eine Konstante `ROCKET`. Das Programm, welches alle diese *Refactorings* beinhaltet, sieht nun so aus:

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH 2))
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET <image>)
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v5 height)
  (cond
    [(<= height ROCKET-CENTER-TO-BOTTOM)
     (place-image ROCKET MIDDLE height MTSCN)]
    [(> height ROCKET-CENTER-TO-BOTTOM)
     (place-image ROCKET MIDDLE ROCKET-CENTER-TO-
BOTTOM MTSCN)]))
```

2.6.2 DRY Redux

Halt! Auch `create-rocket-scene-v5` verstößt noch gegen das DRY-Prinzip. Allerdings werden wir die verbliebenen Redundanzen nicht durch Funktions- oder Konstantendefinitionen eliminieren.

Eine Redundanz ist die, dass die Kondition `(> height ROCKET-CENTER-TO-BOTTOM)` genau dann wahr ist wenn `(<= height ROCKET-CENTER-TO-BOTTOM)`

falsch ist. Diese Information steht jedoch nicht direkt im Programmtext; stattdessen wird die Kondition wiederholt und negiert. Eine Möglichkeit wäre, eine Funktion zu schreiben, die diese Kondition abhängig vom `height` Parameter berechnet und diese Funktion dann in beiden Zweigen der Kondition aufzurufen (und einmal zu negieren). In diesem Fall bietet sich allerdings eine einfachere Lösung an, nämlich statt `cond` `if` zu verwenden. Damit können wir diese Redundanz eliminieren:

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH 2))
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET  )
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v6 height)
  (if
    (<= height ROCKET-CENTER-TO-BOTTOM)
    (place-image ROCKET MIDDLE height MTSCN)
    (place-image ROCKET MIDDLE ROCKET-CENTER-TO-
BOTTOM MTSCN)))
```

Die letzte Redundanz, die wir in `create-rocket-scene-v6` eliminieren wollen, ist die, dass die beiden Aufrufe von `place-image` bis auf einen Parameter identisch sind. Falls in einem konditionalen Ausdruck die Bodies aller Zweige bis auf einen Unterausdruck identisch sind, können wir die Kondition in den Audruck *hineinziehen*, und zwar so:

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH 2))
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET  )
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v7 height)
  (place-image
    ROCKET
    MIDDLE
    (if (<= height ROCKET-CENTER-TO-BOTTOM)
        height
        ROCKET-CENTER-TO-BOTTOM) MTSCN))
```

2.7 Bedeutung von Funktions- und Konstantendefinitionen

Wir haben oben gesagt, dass es keine Rolle spielt, ob die Konstanten oberhalb oder unterhalb der Funktionsdefinition definiert werden. Allerdings spielt es sehr wohl eine Rolle, in welcher Reihenfolge diese Konstanten definiert werden. Wie sie sehen, verwenden einige der Konstantendefinitionen andere Konstanten. Zum Beispiel verwendet die Definition von `MTSCN WIDTH`. Dies ist auch sinnvoll, denn andernfalls hätte man weiterhin die Redundanz die man eigentlich eliminieren wollte.

DrRacket wertet ein Programm von oben nach unten aus. Wenn es auf eine Konstantendefinition trifft, so wird sofort der Wert des Ausdrucks, an den der Name gebunden werden soll (die `CONSTANTExpression`), berechnet. Wenn in diesem Ausdruck eine Konstante vorkommt, die DrRacket noch nicht kennt, so gibt es einen Fehler:

```
> (define A (+ B 1))
B: this variable is not defined
> (define B 42)
```

Daher dürfen in Konstantendefinitionen nur solche Konstanten (und Funktionen) verwendet werden, die oberhalb der Definition bereits definiert wurden.

Tritt dieses Problem auch bei Funktionen auf? Hier ein Versuch:

```
(define (add6 x) (add3 (add3 x)))
(define (add3 x) (+ x 3))


> (add6 5)
11
```

Der Grund, wieso die Reihenfolge von Funktionsdefinitionen nicht wichtig ist, ist der, dass DrRacket bei Auswertung einer Funktionsdefinition lediglich registriert, dass es eine neue Funktion des angegebenen Namens gibt, jedoch im Unterschied zu Konstantendefinitionen die `BodyExpression` der Funktion nicht auswertet.

Etwas formaler können wir die Bedeutung von Programmen mit Funktions- und Konstantendefinitionen so definieren:

- Ein Programm ist eine Sequenz von Ausdrücken, Konstantendefinitionen und Funktionsdefinitionen. Diese können in beliebiger Reihenfolge auftreten.
- Ein Kontext ist eine Menge von Funktions- und Konstantendefinitionen. Der Kontext ist am Anfang der Programmausführung leer.
- Ein Programm wird von links nach rechts (bzw. oben nach unten) ausgewertet. Hier sind nun drei Fälle zu unterscheiden.
  - Ist das nächste Programmelement ein Ausdruck, so wird dieser gemäß der bekannten Reduktionsregeln im aktuellen Kontext zu einem Wert ausgewertet. Für die Auswertung von Konstanten gilt hierbei `x` $\rightarrow$ `v`, falls der Kontext die Definition `(define x v)` enthält.

- Ist das nächste Programmelement eine Funktionsdefinition, so wird diese Funktionsdefinition dem aktuellen Kontext hinzugefügt.
- Ist das nächste Programmelement eine Konstantendefinition `(define CONSTANTNAME CONSTANTExpression)`, so wird `CONSTANTExpression` im aktuellen Kontext zu einem Wert `v` ausgewertet und zum Kontext die Definition `(define CONSTANTNAME v)` hinzugefügt.

Der aktuelle Kontext wird im Stepper von DrRacket angezeigt, und zwar als die Menge der Funktions- und Konstantendefinitionen, die oberhalb des aktuell zu reduzierenden Ausdrucks stehen. Bitte benutzen Sie den Stepper um die Reduktion des folgenden Programms zu visualisieren. Am besten versuchen Sie erst auf einem Blatt Papier vorherzusagen, welches die Reduktionsschritte sein werden und kontrollieren dann mit dem Stepper.

```
(define WIDTH 100)
(define HEIGHT 100)
(define MIDDLE (/ WIDTH 2))
(define MTSCN (empty-scene WIDTH HEIGHT))

(define ROCKET  )
(define ROCKET-CENTER-TO-BOTTOM (- HEIGHT (/ (image-
height ROCKET) 2)))
(define (create-rocket-scene-v7 height)
  (place-image
    ROCKET
    MIDDLE
    (if (<= height ROCKET-CENTER-TO-BOTTOM)
        height
        ROCKET-CENTER-TO-BOTTOM) MTSCN))
(create-rocket-scene-v7 42)
```

Randnotiz: Zählen Sie einmal die Anzahl der Reduktionsschritte, die Sie pro Aufruf von `create-rocket-scene-v7` zusätzlich benötigen (also wenn Sie noch weitere Aufrufe zum Programm hinzufügen). Wieviele zusätzliche Schritte benötigen Sie, wenn Sie stattdessen `create-rocket-scene-v2` verwenden? Wie kommt es zu den Unterschieden und was bedeuten sie?

2.8 Programmieren ist mehr als das Regelverstehen!

Ein guter Schachspieler muss die Regeln des Schachspiels verstehen. Aber nicht jeder, der die Schachregeln versteht ist auch ein guter Schachspieler. Die Schachregeln verraten nichts darüber, wie man eine gute Partie Schach spielt. Das Verstehen der Regeln ist nur ein erster kleiner Schritt auf dem Weg dahin.

Jeder Programmierer muss die "Mechanik" der Programmiersprache beherrschen: Was gibt es für Konstrukte in der Programmiersprache und was bedeuten sie? Was gibt es für vordefinierte Funktionen und Bibliotheken?

Trotzdem ist man dann noch lange kein guter Programmierer. Viele Anfängerbücher (und leider auch viele Anfängerkurse an Universitäten) fürs Programmieren sind so gehalten, dass sie sich *nur* auf diese mechanischen Aspekte der Programmierung fokussieren. Noch schlimmer, sie lernen nicht einmal, was genau ihre Programme bedeuten, sondern sie lernen im Wesentlichen nur die Syntax einer Programmiersprache und einige ihrer Bibliotheken.

Das liegt daran, dass einige Programmiersprachen, die in Anfängerkursen verwendet werden, so kompliziert sind, dass man den Großteil des Semesters damit verbringt, nur die Syntax der Sprache zu lernen. Unsere Sprache, BSL, ist so einfach, dass Sie bereits jetzt die Mechanik dieser Sprache verstehen — ein Grund dafür, wieso wir uns für diese Sprache entschieden haben. Wir werden zwar noch einige weitere Sprachfeatures einführen, aber im größten Teil dieser Vorlesung geht es um den interessanteren Teil der Programmierung: Wie kommt man von einer Problembeschreibung systematisch zu einem guten Programm? Was für Arten der Abstraktion gibt es und wie kann man sie einsetzen? Wie gehe ich mit Fehlern um? Wie strukturiere ich mein Programm so, dass es lesbar, wartbar und wiederverwendbar ist? Wie kann ich die Komplexität sehr großer Programme beherrschen?

Die Antworten, die wir auf diese Fragen geben, werden Ihnen in *allen* Programmiersprachen, die sie verwenden werden, nutzen. Darum wird es in diesem Kurs gehen.

3 Systematischer Programmentwurf

Dieser Teil des Skripts basiert auf [HTDP/2e] Kapitel I und dem Artikel "On Teaching How to Design Programs" von Norman Ramsey.

In diesem Kapitel werden keine neuen Sprachfeatures vorgestellt. Stattdessen geht es in diesem Kapitel um die *Methodik* des Programmierens: Wie kann systematisch eine Problembeschreibung in ein qualitativ hochwertiges Programm umgeformt werden?

3.1 Funktionale Dekomposition

Programme bestehen nur in den seltensten Fällen aus einer einzelnen Funktion. Typischerweise bestehen Programme aus vielen Funktionsdefinitionen, die teilweise dadurch voneinander abhängig sind, dass sie sich untereinander aufrufen. Betrachten Sie das folgende Programm zur Erstellung etwas plumper "Nigerian-Scam" Briefe:

```
(define (letter fst lst signature-name)
  (string-append
    (opening lst)
    "\n"
    (body fst lst)
    "\n"
    (closing signature-name)))

(define (opening lst)
  (string-append "Sehr geehrte(r) Herr/Frau " lst ","))

(define (body fst lst)
  (string-append
   "Herr Gadafi aus Libyen ist gestorben und hat Sie, "
   fst
   ", in seinem Testament als Alleinerben eingesetzt.\n"
   "Lieber " fst lst ", gegen eine kleine Bearbeitungsgebühr
überweise ich das Vermögen."))

(define (closing signature-name)
  (string-append
   "Mit freundlichen Grüßen,"
   "\n"
   signature-name))
```

Diese Definitionen können wir nun benutzen, um mit wenig Aufwand viele solcher Briefe zu erstellen:

```
> (letter "Tillmann" "Rendel" "Klaus Ostermann")
"Sehr geehrte(r) Herr/Frau Rendel,\nHerr Gadafi aus Libyen
ist gestorben und hat Sie, Tillmann, in seinem Testament als
Alleinerben eingesetzt.\nLieber TillmannRendel, gegen eine
kleine Bearbeitungsgebühr überweise ich das Vermögen.\nMit
freundlichen Grüßen,\nKlaus Ostermann"
```

Das Ergebnis ist ein langer String. Das \n in dem String steht für einen Zeilenumbruch. Sobald dieser String zum Beispiel in eine Datei geschrieben wird, wird aus dem \n ein echter Zeilenumbruch.

Im Allgemeinen sollte ein Programm so gestaltet sein, daß es eine Funktion pro Aufgabe gibt, die das Programm erledigen soll. Aufgaben sollten hierbei hierarchisch angeordnet sein: Auf der obersten Ebene gibt es die Gesamtaufgabe des Programms (wie das Anfertigen eines Serienbriefs); diese große Aufgabe wird in kleinere Aufgaben wie das Anfertigen der Eröffnung eines Serienbriefs zerlegt, diese können ggf. wieder in weitere kleine Aufgaben zerlegt werden.

Die Struktur der Funktionen sollte der hierarchischen Struktur der Aufgaben folgen. Zu jeder Aufgabe gibt es eine Funktion die in ihrer Implementierung die Funktionen aufruft, die zu den Unteraufgaben korrespondieren. Die Funktionen können dementsprechend als Baum (oder azyklischer Graph) angeordnet werden, an dessen Wurzel die Hauptfunktion steht und an dessen Blättern die Funktionen der untersten Ebene (die nur primitive Funktionen aufrufen) stehen.

Ein Programm, in dem jede Funktion eine klare Aufgabe hat, ist leicht zu verstehen und leichter zu warten, denn wenn es Änderungswünsche gibt, so betreffen diese typischerweise eine bestimmte Aufgabe des Systems. Ist die Implementation dieser Aufgabe in Form einer Funktionsdefinition lokalisiert, so muss nur diese eine Funktion modifiziert werden.

Wenn man eine größere Menge von Funktionen zu programmieren hat, stellt sich die Frage, in welcher Reihenfolge man diese Funktionen programmiert. Zwei wichtige Varianten sind "Top-Down" und "Bottom-Up". "Top-Down" bedeutet, dass man mit der Hauptfunktion anfängt, dann alle Funktionen programmiert die in der Hauptfunktion aufgerufen werden, dann alle Funktionen die widerum von diesen Funktionen aufgerufen werden, und so weiter. "Bottom-Up" ist genau umgekehrt: Man programmiert zunächst die einfachsten Funktionen, die wiederum nur primitive Funktionen aufrufen, dann die Funktionen, welche die gerade programmierten Funktionen aufrufen. Dies setzt man fort bis man ganz zum Schluss die Hauptfunktion programmiert. Abzuschätzen, welche Vorgehensweise in welcher Situation die richtige ist, ist ein wichtiges Thema in der Methodik der Softwarekonstruktion.

Wenn die Aufrufstruktur der Funktionen ein azyklischer Graph ist, so ergibt sich automatisch eine Schichtenstruktur, in der alle Funktionen in einer Schicht nur Funktionen aus darunter liegenden Schichten aufrufen. Eine noch stärkere Einschränkung ist die, dass Funktionen ausschliesslich Funktionen aus der Schicht direkt unter ihnen aufrufen dürfen. Diese Einschränkung ist allerdings sinnvoll, denn sie ermöglicht bei geschickter Wahl der Funktionen, eine Schicht zu verstehen ohne alle darunter liegenden Schichten verstehen zu müssen. Diese Situation nennt man *hierarchische Abstraktion*, und sie ist der Schlüssel, um mit Komplexität großer Softwaresysteme umzugehen.

Die Grundidee bei hierarchischer Abstraktion durch Funktionen ist die, dass man Funktionen so gestalten sollte, dass ein Programmierer anhand des Namens und der Dokumentation (der *Spezifikation*) einer Funktion in der Lage ist, eine Funktion effektiv in einem Programm zu verwenden — es also nicht notwendig ist, den Programmtext der Funktion und (transitiv) aller Funktionen, die diese Funktion aufruft, zu verstehen. Später werden wir sehen, dass die Spezifikation einer Funktion zumindest partiell in

Form von Testfällen formalisiert werden kann. Für informelle Spezifikationen werden meist Kommentare verwendet, die oberhalb der Funktionsdefinition stehen.

In unserem Beispiel oben ist es so, dass man nicht die Details der `opening` Funktion verstehen muss, um die `letter` Funktion zu verstehen; es reicht, zu wissen, dass diese Funktion die Eröffnung des Briefes zurückgibt — wie auch immer die Details davon aussehen werden. In diesem Fall ist der Name der Funktion ausreichend, um `opening` effektiv benutzen zu können; in anderen Fällen ist weitere Dokumentation erforderlich. In jedem Fall sollten Sie jedoch versuchen, Funktionsnamen so zu wählen, dass möglichst wenig weitere Dokumentation erforderlich ist, denn wenn sich ein Programm weiterentwickelt, passiert es häufig, dass nur der Programmtext aber nicht die Kommentare "gewartet" werden und dementsprechend Dokumentation häufig obsolet ist.

3.2 Vom Problem zum Programm

Betrachten Sie folgende Problemstellung für ein zu erstellendes Programm:

The owner of a movie theater has complete freedom in setting ticket prices. The more he charges, the fewer the people who can afford tickets. In a recent experiment the owner determined a precise relationship between the price of a ticket and average attendance. At a price of $5.00 per ticket, 120 people attend a performance. Decreasing the price by a dime ($.10) increases attendance by 15. Unfortunately, the increased attendance also comes at an increased cost. Every performance costs the owner $180. Each attendee costs another four cents ($0.04). The owner would like to know the exact relationship between profit and ticket price so that he can determine the price at which he can make the highest profit.

Die Aufgabe ist relativ klar, aber wie man daraus ein Programm macht nicht. Eine gute Art, diese Aufgabe anzugehen, ist es, die Quantitäten und ihre Abhängigkeiten voneinander zu betrachten und nacheinander in Form einer Funktion zu definieren:

Die Problemstellung sagt, wie die Anzahl der Zuschauer vom Eintrittspreis abhängt. Dies ist eine klar definierte Unteraufgabe, daher verdient sie eine eigene Funktion:

```
(define (attendees ticket-price)
  (+ 120 (* (/ 15 0.1) (- 5.0 ticket-price))))
```

Der Umsatz hängt vom Verkauf der Eintrittskarten ab: Es ist das Produkt aus Eintrittspreis und Anzahl der Zuschauer.

```
(define (revenue ticket-price)
  (*  (attendees ticket-price) ticket-price))
```

Die Kosten setzen sich aus zwei Teilen zusammen: Einem festen Anteil ($180) und einem variablen Teil, der von der Anzahl der Zuschauer abhängt. Da die Zahl der Zuschauer wiederum vom Eintrittspreis abhängt, muss diese Funktion auch den Ticketpreis als Eingabeparameter entgegennehmen und verwendet die bereits definierte `attendees` Funktion:

Dieses Programm enthält diverse *magic numbers*. Eliminieren Sie diese durch entsprechende Konstantendefinitionen!

```
(define (cost ticket-price)
   (+ 180 (* 0.04 (attendees ticket-price))))
```

Der Gewinn ist schliesslich die Differenz zwischen Umsatz und Kosten. Da wir bereits Funktionen für die Berechnung von Umsatz und Kosten haben, muss diese Funktion all die Werte als Eingabe bekommen, die diese Funktionen benötigen — in diesem Fall ist dies der Eintrittspreis:

```
(define (profit ticket-price)
  (- (revenue ticket-price)
     (cost ticket-price)))
```

Diese Funktionen können wir nun verwenden, um den Gewinn bei einem bestimmten Eintrittspreis zu berechnen. Probieren Sie aus, bei welchem Preis der Gewinn maximiert wird!

Hier ist eine alternative Version des gleichen Programms, welches nur aus einer einzigen Funktion besteht:

```
(define (profit price)
 (- (* (+ 120
          (* (/ 15 0.1)
             (- 5.0 price)))
          price)
    (+ 180
       (* 0.04
          (+ 120
             (* (/ 15 0.1)
                (- 5.0 price)))))))
```

Überprüfen Sie, dass diese Definition tatsächlich die gleichen Ergebnisse wie das Programm oben produziert. Wir benötigen also prinzipiell nur eine Funktion für dieses Programm; dennoch ist es offensichtlich, dass die erste Version oben deutlich lesbarer ist. Sie ist auch besser wartbar: Überlegen Sie sich beispielsweise, welche Änderungen im Programmtext erforderlich sind, wenn die festen Kosten entfallen und stattdessen pro Zuschauer Kosten in Höhe von $1,50 anfallen. Probieren Sie in beiden Versionen, diese Änderung zu implementieren. Vergleichen Sie.

3.3 Systematischer Entwurf mit Entwurfsrezepten

Das meiste, was sie bisher über das Programmieren gelernt haben, dreht sich darum, wie die Programmiersprache, die sie verwenden, funktioniert und was ihre Konstrukte bedeutet. Wir haben einige wichtige Sprachkonstrukte kennengelernt (Funktionsdefinitionen, Konstantendefinitionen, Ausdrücke) und etwas Erfahrung damit gesammelt, wie man diese Konstrukte einsetzen kann.

Diese Kenntnisse sind jedoch nicht ausreichend, um systematisch aus einer Problembeschreibung ein Programm zu konstruieren. Hierzu müssen wir lernen, was in einer Problembeschreibung relevant ist und was nicht. Wir müssen verstehen, welche

Daten das Programm konsumiert und welche es abhängig von den Eingabedaten produzieren muss. Wir müssen herausfinden, ob eine benötigte Funktion in einer Bibliothek vielleicht schon vorhanden ist oder ob wir selber diese Funktion programmieren müssen. Wenn wir ein Programm haben, müssen wir sicherstellen, dass es sich tatsächlich wie gewünscht verhält. Hierbei können allerlei Fehler auftreten, die wir verstehen und beheben müssen.

Gute Programme haben eine kurze Beschreibung dessen, was sie tun, welche Eingabe sie erwarten, und was für eine Ausgabe sie produzieren. Am besten wird gleichzeitig dokumentiert, dass das Programm tatsächlich funktioniert. Nebenbei sollten Programme auch noch so strukturiert sein, dass eine kleine Änderung in der Problembeschreibung auch nur eine kleine Änderung am Programm bedeutet.

Diese ganze Arbeit ist notwendig, weil Programmierer nur selten für sich selbst Programme schreiben. Programmierer schreiben Programme, die andere Programmierer verstehen und weiterentwickeln müssen. Große Programme werden über lange Zeiträume von großen Teams entwickelt. Neue Programmierer stoßen während dieses Zeitraums hinzu, andere gehen. Kunden ändern ständig ihre Anforderungen an das Programm. Große Programme enthalten fast immer Fehler, und oft sind diejenigen, die den Fehler beheben müssen nicht identisch mit denen, die die Fehler eingebaut haben.

Ein reales, im Einsatz befindliches Programm ist zudem niemals "fertig". Es muss in den meisten Fällen ständig weiterentwickelt oder zumindest gewartet oder an neue Technik angepasst werden. Es ist keine Seltenheit, dass Programme eine Lebenszeit von vielen Jahrzehnten haben und daher die ursprünglichen Programmierer eines Programms, das weiterentwickelt oder gewartet werden soll, vielleicht schon im Ruhestand oder verstorben sind.

Recherchieren Sie, was das "Jahr 2000 Problem" ist.

Die gleichen Probleme treten übrigens selbst dann auf, wenn es nur einen einzigen Programmierer gibt, denn auch dieser vergißt nach einiger Zeit, was er vor sich einmal bei einem Programm gedacht hat, und profitiert dann genau so von einem sinnvollen Entwurf als wenn ein anderer Programmierer seinen Code lesen würde.

Aus diesen Gründen werden wir Ihnen systematische Anleitungen an die Hand geben, mit denen unterschiedliche Entwurfsaufgaben Schritt für Schritt gelöst werden können. Diese Anleitungen nennen wir *Entwurfsrezepte*.

3.3.1 Testen

Ein wichtiger Bestandteil der vorgestellten Methodik wird es sein, Programme systematisch und automatisiert zu *testen*. Beim Testen führt man ein Programm oder einen Programmteil (wie eine Funktion) mit Testdaten aus und überprüft, ob das Resultat das ist, welches man erwartet. Natürlich kann man "von Hand" testen, indem man zum Beispiel im Interaktionsbereich Ausdrücke auswertet und die Ergebnisse überprüft. Allerdings wird es schnell langweilig, wenn man die gleichen Tests immer wieder aufschreibt, um zu überprüfen, ob man bei einer Programmänderung nichts "kaputt" gemacht hat.

Wann, wo, wieso und wieviel man testen sollte werden wir später diskutieren. Hier beschreiben wir nur, *wie* man in DrRacket automatisiert testet. Hierzu gibt es eine spezielle Funktion in BSL, `check-expect`. Diese erwartet zwei Parameter, von denen

der erste ein Ausdruck ist, der getestet werden soll, und der zweite ein Ausdruck, der das gewünschte Ergebnis beschreibt. Beispiel: `(check-expect (+ 2 3) 5)` überprüft, ob das Ergebnis der Auswertung von `(+ 2 3)` den Wert `5` ergibt.

Hier ist ein Beispiel wie `check-expect` verwendet werden kann, um eine Funktion zur Konvertierung zwischen Fahrenheit und Grad Celsius zu testen:

```
(check-expect (f2c -40) -40)
(check-expect (f2c 32) 0)
(check-expect (f2c 212) 100)

(define (f2c f)
  (* 5/9 (- f 32)))
```

Alle `check-expect` Tests werden jedesmal ausgeführt, wenn auf den "Start" Knopf gedrückt wird. Falls alle Tests erfolgreich waren, wird dies durch eine kurze Meldung quittiert. Falls mindestens ein Test fehlschlägt, wird dies durch eine Fehlermeldung mit genaueren Informationen dazu angezeigt.

Es gibt einige Varianten von `check-expect`, wie zum Beispiel `check-within` und `check-range`. Verschaffen Sie sich mit Hilfe der Dokumentation einen Überblick.

Um den Programmierer zu unterstützen, zeigt DrRacket durch Färbung des Codes an, welche Teile ihres Programms bei der Ausführung der Tests durchlaufen wurden. Probieren Sie dieses Verhalten selbst aus!

Selbstverständlich funktionieren Tests nicht nur mit Zahlen sondern mit allen Datentypen wie beispielsweise auch Bildern. Beispielsweise können wir diese Funktion

```
(define (ring innerradius outerradius color)
  (overlay (circle innerradius "solid" "white")
           (circle outerradius "solid" color)))
```

so testen:

```
(check-expect (ring 5 10 "red")  )
```

Da `check-expect` beliebige Ausdrücke als Parameter erwartet, können statt dem Ergebnis selber auch auch *Eigenschaften* von Ergebnissen überprüft werden, zum Beispiel so:

```
(check-expect (image-width (ring 5 10 "red")) 20)
```

Wir werden später sehen, dass dies wichtig ist, um Tests nicht zu stark an die Implementation einer Funktion zu koppeln.

3.3.2 Informationen und Daten

Ein Programm beschreibt eine Berechnung, die *Informationen* aus der Domäne des Programms verarbeitet und produziert. Eine Information ist zum Beispiel so etwas wie "das Auto ist 5m lang" oder "der Name des Angestellten ist 'Müller' ".

Ein Programm kann jedoch solche Informationen nicht direkt verarbeiten. Wir müssen stattdessen Informationen als *Daten* repräsentieren. Diese Daten können wiederum als Information *interpretiert* werden. Beispielsweise könnten wir die erste Information oben als Zahl mit dem Wert `5` und die zweite Information als String mit dem Wert `"Müller"` repräsentieren.

Eine Datum wie die Zahl `5` wiederum kann auf vielfältige Weise interpretiert werden. Beispielsweise können, wie im Beispiel oben, die Länge eines Autos in Metern gemeint sein. Genau so kann sie aber als Temperatur in Grad Celsius, als Geldbetrag in Euro, oder als Endnote Ihrer Klausur in dieser Veranstaltung interpretiert werden (hoffentlich nicht :-).

Da diese Beziehung zwischen Information und Daten so wichtig ist, werden wir Sie von nun an in Form spezieller Kommentare, die wir *Datendefinitionen* nennen, aufschreiben. Eine Datendefinition beschreibt eine Klasse von Daten durch einen sinnvollen Namen, der auf die Interpretation der Daten hinweist.

Hier sind einige Beispiele für Datendefinitionen:

```
; Distance is a Number.
; interp. the number of pixels from the top margin of a canvas

; Speed is a Number.
; interp. the number of pixels moved per clock tick

; Temperature is a Number.
; interp. degrees Celsius

; Length is a Number.
; interp. the length in centimeters

; Count is a Number.
; interp. the number of characters in a string.
...
```

Zum jetzigen Zeitpunkt kennen Sie nur einige wenige Formen von Daten (Zahlen, Strings, Bilder, Wahrheitswerte), daher müssen Sie alle Informationen mit diesen Datentypen repräsentieren. Später werden wir andere Datentypen kennenlernen, in denen es deutlich anspruchsvoller wird, eine geeignete Repräsentation für seine Informationen zu wählen.

Oft ist es hilfreich, Datenbeispiele zu einer Datendefinition anzugeben.

```
; Temperature is a Number.
; interp. degrees Celsius
; Examples:
(define sunny-weather 25)
(define bloody-cold -5)
```

Die Definition von Datenbeispielen hat zwei Vorteile: 1) Sie helfen, eine Datendefinition zu verstehen. Beispielsweise könnte es sein, dass Sie versehentlich ihre Datendefinition so gestaltet haben, dass es gar keine Datenbeispiele gibt. 2) Sie können

die Beispiele in Tests von Funktionen verwenden, die solche Daten konsumieren oder produzieren.

3.3.3 Entwurfsrezept zur Funktionsdefinition

Auf Basis der gerade besprochenen Trennung zwischen Informationen und Daten können wir nun den Entwurf einzelner Funktionen als eine Abfolge von Schritten beschreiben.

1. Definieren Sie wie Sie die für die Funktion relevanten Informationen (Eingabe und Ausgabe) als Daten repräsentieren. Formulieren Sie entsprechende Datendefinitionen (sofern nicht bereits vorhanden). Geben Sie für nicht-triviale Datendefinitionen einige interessante Beispiele für die Datendefinition an.

2. Schreiben Sie eine Signatur, eine Aufgabenbeschreibung, und einen Funktionskopf. Eine *Signatur* ist ein BSL Kommentar, der dem Leser sagt, wieviele und welche Eingaben die Funktion konsumiert und was für eine Ausgabe sie produziert. Hier sind zwei Beispiele:

   - Für eine Funktion, die einen String konsumiert und eine Zahl produziert:
     ; String -> Number
   - Für eine Funktion die eine Temperatur und einen Boolschen Wert konsumiert und einen String produziert:
     ; Temperature Boolean -> String
     Beachten Sie dass wir die vorherige Datendefinition für *Temperature* verwendet haben.

   Eine *Aufgabenbeschreibung* ist ein BSL Kommentar der den Zweck der Funktion **in einer Zeile** zusammenfasst. Die Aufgabenbeschreibung ist die kürzestmögliche Antwort auf die Frage: *Was berechnet die Funktion?* Jeder Leser ihres Programms sollte verstehen, was eine Funktion berechnet ohne die Funktionsdefinition selbst lesen zu müssen.

   Ein *Funktionskopf*, manchmal auch *Header* oder *Stub* genannt, ist eine Funktionsdefinition, die zur Signatur passt aber in der der Body der Funktion nur ein Dummy-Wert ist, zum Beispiel `0` falls eine Zahl zurückgegeben werden soll oder `(empty-scene 100 100)` falls ein Bild zurückgegeben werden soll. Beim Entwurf des Funktionskopfs müssen trotzdem wichtige Entscheidungen getroffen werden, nämlich die Namen der Funktion und der Eingabeparameter müssen bestimmt werden. Typischerweise sollten die Parameternamen einen Hinweis darauf geben, was für Informationen oder welchen Zweck die Parameter repräsentieren. Die Namen der Parameter können oft sinnvoll in der Aufgabenbeschreibung verwendet werden.

   Hier ein vollständiges Beispiel für eine Funktionsdefinition nach diesem Schritt:

```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left
```

```
(define (add-image y s img)
  (empty-scene 100 100))
```

Zu diesem Zeitpunkt können Sie bereits auf den "Start" Knopf drücken und die Funktion benutzen — allerdings wird natürlich stets nur der Dummy-Wert und nicht das gewünschte Ergebnis zurückgegeben.

3. Schreiben Sie zwischen Aufgabenbeschreibung und Funktionskopf *Tests*, die anhand von Beispielen dokumentieren, was die Funktion macht. Diese Tests sind einerseits Teil der Dokumentation der Funktion, auf der anderen Seite werden diese Tests automatisiert ausgeführt und schützen sie damit vor Fehlern im Funktionsbody. Hier ist ein Beispiel, wie eine Funktion nach diesem Schritt aussieht:

   ```
   ; Number -> Number
   ; compute the area of a square whose side is len
   (check-expect (area-of-square 2) 4)
   (check-expect (area-of-square 7) 49)

   (define (area-of-square len) 0)
   ```

   Führen Sie die Tests nun einmal aus und überprüfen, dass alle Tests (für die die Dummy-Implementierung nicht zufällig das korrekte Ergebnis liefert) fehlschlagen. Dieser Schritt ist wichtig um Fehler in der Formulierung der Tests zu finden, die bewirken könnten, dass der Test immer erfolgreich ist.

4. In diesem Schritt überlegen Sie sich, welche der Ihnen zur Verfügung stehenden Eingabedaten und ggf. Hilfsfunktionen und Variablen Sie zur Berechnung benötigen. Sie ersetzen den Dummy-Wert aus dem zweiten Schritt mit einem *Template* (Schablone), in dem die Eingabedaten/Funktionen/Variablen von oben vorkommen. Im Moment sieht dieses Template so aus, dass einfach die Eingabedaten/Funktionen/Variablen durch `...` voneinander getrennt unsortiert im Funktionsbody stehen. Später werden wir interessantere Templates kennenlernen.

   In unserem letzten Beispiel könnte die Funktion nach diesem Schritt beispielsweise so aussehen:

   ```
   ; Number -> Number
   ; compute the area of a square whose side is len
   (check-expect (area-of-square 2) 4)
   (check-expect (area-of-square 7) 49)

   (define (area-of-square len) (... len ...))
   ```

5. Jetzt ist es an der Zeit, den Funktionsbody zu implementieren, also das Template nach und nach durch einen Ausdruck zu ersetzen, der die Spezifikation (Signatur, Aufgabenbeschreibung, Tests) erfüllt. Unsere `area-of-square` Funktion könnte nun so aussehen:

```
; Number -> Number
; compute the area of a square whose side is len
(check-expect (area-of-square 2) 4)
(check-expect (area-of-square 7) 49)

(define (area-of-square len) (* len len))
```

Die `add-image` Funktion könnte nach diesem Schritt so aussehen:

```
; Number String Image -> Image
; add s to img, y pixels from top, 10 pixels to the left
(check-expect (add-image 5 "hello" (empty-scene 100 100))
              (place-image (text "hello" 10 "red") 10 5 (empty-
scene 100 100)))
(define (add-image y s img)
  (place-image (text s 10 "red") 10 y img))
```

Es ist wichtig, zu verstehen, dass der Test *nicht* aussagt, dass `add-image` mit Hilfe von `place-image` implementiert wurde. Die `check-expect` Funktion vergleicht die Bilder, die aus den Audrücken entstehen, und nicht die Ausdrücke selber. Beispielsweise könnten wir den zweiten Parameter auch durch ein Bildliteral (also ein Bild im Programmtext) ersetzen. Daher "verrät" der Test nicht die Implementation der Funktion. Dies ist wichtig, weil wir damit die Möglichkeit haben, die Implementation der Funktion zu verändern, ohne dass Klienten der Funktion (also Aufrufer der Funktion) davon betroffen werden. Dieses Konzept nennt man auch *Information Hiding*.

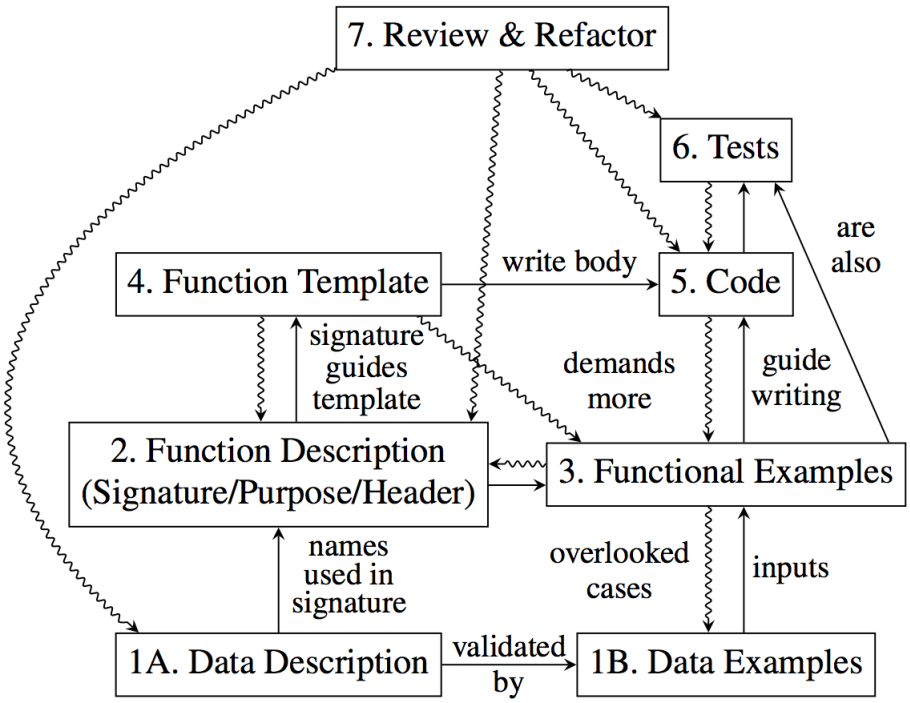
6. Nun ist es Zeit, die Funktion zu testen. Da unsere Tests automatisiert sind, genügt hierzu ein Klick auf "Start". Falls ein Test fehlschlägt, ist entweder der Test falsch, oder die Funktionsdefinition enthält einen Fehler (oder beides gleichzeitig). Als erstes sollten Sie in diesem Fall überprüfen, ob das beobachtete Verhalten wirklich fehlerhaft war oder nur ihr Test fehlerhaft ist. Reparieren Sie, je nachdem, den Test beziehungsweise die Funktionsdefinition, bis der Test fehlerfrei ausgeführt wird.

7. Der letzte Schritt besteht in der Nachbearbeitung und dem Refactoring der neu definierten Funktion und ggf. anderer Teile des Programms. Aktivitäten in diesem Schritt umfassen:

   - Überprüfung der Korrektheit der Signatur und Aufganbenbeschreibung und der Übereinstimmung mit der Implementierung der Funktion (z.B. Anzahl und Aufgabe der Parameter).
   - Überprüfung der Testabdeckung: Wird der Code vollständig durch Testcases abgedeckt? Gibt es ein Beispiel für jeden interessanten Fall der Eingabeparameter ("corner cases")? Gibt es ein Beispiel für jeden interessanten Fall der Ausgabe?

- Überprüfen Sie, ob die Funktionsdefinition dem vorgeschlagenen Template des Entwurfsrezept entspricht.
- Überprüfen Sie, ob es Funktionen oder Konstanten gibt, die nicht mehr benötigt und daher gelöscht werden können.
- Suchen Sie nach redundantem Code, also Code der identisch oder ähnlich an mehreren Stellen des Programms vorkommt. Identifizieren Sie ggf., wie Sie durch eine Konstanten- oder Funktionsdefinition die Redundanz eliminieren können.
- Überprüfen Sie, ob es Funktionen mit ähnlichen Aufgabenbeschreibungen und/oder ähnlichen Ein-/Ausgaben gibt. Identifizieren und eliminieren Sie ggf. die Redundanz.
- Vereinfachen Sie konditionale Ausdrücke in denen verschiedene Fälle zusammengefasst werden können.
- Testen Sie nach jedem Refactoring unverzüglich das Programm. Ein Refactoring sollte niemals das Verhalten des Programms ändern. Vermischen Sie auf keinen Fall Refactoring und Erweiterung/Modifikation des Programms.

Hier ist eine graphische Darstellung des Entwurfsrezepts. Die geraden Pfeile beschreiben den Informationsfluss bzw. die Reihenfolge im initialen Entwurf; die schnörkeligen Pfeile beschreiben die Rückkopplung durch die Nachbearbeitung und das Refactoring des Programms.



Diese Abbildung
ımmt aus dem
ʻtikel "On
aching How to
ɛsign Programs"
n Norman
ımsey.

3.3.4 Programme mit vielen Funktionen

Die meisten Programme bestehen nicht aus einer sondern aus vielen Funktionen. Diese Funktionen sind häufig voneinander abhängig, da eine Funktion eine andere Funktion aufrufen kann.

Sie haben oben ein Entwurfsrezept für den Entwurf einzelner Funktionen gesehen. Dieses sollten beim Entwurf jeder einzelnen Funktion verwenden. Wenn Sie viele Funktionen und globale Konstanten (Variablen) definiert haben, so sollten Sie im Funktionstemplate die Funktionen und Konstanten aufführen, von denen Sie glauben, dass sie im endgültigen Funktionsbody benötigt werden.

Da sie nicht alle Funktionen auf einmal programmieren können, stellt sich die Frage, in welcher Reihenfolge sie vorgehen. Ein häufiger Ansatz ist der *Top-Down Entwurf*, bei dem man zuerst die Hauptfunktion(en) der Anwendung programmiert. Diese Funktionen werden zweckmäßigerweise in weitere Hilfsfunktionen zerlegt, die während des Programmierens der Funktion noch gar nicht existieren. Deshalb bietet es sich an, stets eine "Wunschliste" der Funktionen, die noch programmiert werden müssen, zu führen. Ein Eintrag auf der Wunschliste besteht aus einem sinnvollen Funktionsnamen, einer Signatur und einer Aufgabenbeschreibung. Am Anfang steht auf dieser Liste nur die Hauptfunktion. Stellen Sie beim Programmieren einer Funktion fest, dass Sie eine neue Funktion benötigen, fügen Sie der Wunschliste einen entsprechenden Eintrag hinzu. Sind sie mit einer Funktion fertig, suchen Sie sich die nächste Funktion von der Liste, die sie implementieren möchten.
Ist die Liste leer, sind sie fertig.

Recherchieren Sie was die Abkürzungen FIFO und LIFO bedeuten. Diskutieren Sie, ob FIFO oder LIFO für die Wunschliste geeignet sind und was für Konsequenzen dies hat.

Ein Vorteil von Top-Down Entwurf ist, dass Sie Schritt für Schritt ihr großes Entwurfsproblem in immer kleinere Probleme zerlegen können, bis die Probleme so klein werden, dass Sie sie direkt lösen können (im Fall von Funktionsdefinitionen sind dies Funktionen, die nur noch eingebaute Funktionen oder Bibliotheksfunktionen verwenden).

Ein wichtiger Nachteil ist, dass Sie erst relativ spät die Details der Hilfsfunktionen programmieren. Falls Sie einen Denkfehler gemacht haben und die Hilfsfunktionen so gar nicht implementiert werden können, müssen Sie unter Umständen einen großen Teil ihrer Arbeit wieder in den virtuellen Papierkorb werfen. Ein anderer wichtiger Nachteil ist der, dass Sie erst sehr spät ihre Funktionen testen können, nämlich erst wenn alle Hilfsfunktionen vollständig implementiert wurden. Eine Möglichkeit, dieses Problem zu umgehen, ist, eine Hilfsfunktion erstmal durch einen *Test Stub* zu ersetzen. Ein Test Stub ist eine Dummy-Funktionsdefinition, die eine vordefinierte Antwort zurückliefert, wie sie im Kontext eines Tests erwartet wird. Nehmen Sie beispielsweise an, sie möchten eine `area-of-cube` Funktion definieren, die die eine noch nicht programmierte `area-of-square` Funktion benutzt. Um `area-of-cube` trotzdem testen zu können, können Sie `area-of-square` zunächst provisorisch durch einen Test Stub wie in diesem Beispiel zu implementieren. Wenn Sie sich später dafür entscheiden, diese Funktion zu implementieren, ersetzen Sie den Test Stub durch die richtige Funktionsdefinition.

```
; Number -> Number
; computes the area of a cube with side length len
```

```
(check-expect (area-of-cube 3) 54)
(define (area-of-cube len) (* 6 (area-of-square len)))

; Number -> Number
; computes the area of a square with side length len
(check-expect (area-of-square 3) 9)
(define (area-of-square len) (if (= len 3) 9 (error "not yet
implemented")))
```

Der Test Stub für `area-of-square` benutzt die Funktion `error`. Diese ist gut dafür geeignet, zu dokumentieren, dass eine Funktion noch nicht fertig implementiert wurde. Dies ist insbesondere besser, als stillschweigend ein falsches Ergebnis zurückzuliefern, denn dann fällt Ihnen unter Umständen erst sehr spät auf, dass Sie diesen Teil noch implementieren müssen.

3.4 Information Hiding

Der Name, die Signatur, die Aufgabenbeschreibung und die Tests bilden zusammen die *Spezifikation* einer Funktion. Die Spezifikation sollte ausreichend viele Informationen enthalten, um die Funktion benutzen zu können — ein Aufrufer sollte also nicht erst die Implementation der Funktion (und vielleicht sogar rekursiv die Implementationen aller Funktionen die darin aufgerufen werden) studieren müssen, um die Funktion nutzen zu können.

Eines der wichtigsten Prinzipien, um in der Programmierung mit der Komplexität großer Programme umzugehen, heißt *Information Hiding*, im Deutschen manchmal auch *Geheimnisprinzip* genannt. In Bezug auf Funktionen sagt dieses Prinzip aus, dass ein Programm besser lesbar, verstehbar und wartbar ist, wenn alle Aufrufer von Funktionen sich nur auf die Spezifikationen der Funktion verlassen, aber nicht von Implementierungsdetails abhängen. Ferner sagt dieses Prinzip aus, dass es einen Unterschied zwischen Spezifikation und Implementation geben sollte, und zwar dergestalt, dass es viele mögliche Implementationen der gleichen Spezifikation gibt. Wenn sich alle an diese Regel halten, so ist garantiert, dass man die Implementation jeder Funktion beliebig modifizieren kann — solange die Spezifikation weiterhin eingehalten wird, ist durch dieses Prinzip sichergestellt, dass das Programm weiterhin funktioniert.

Betrachten Sie als Beispiel die `body` Funktion aus dem Spam-Mail-Generator von oben. Hier ist die Definition mit einer möglichen Spezifikation:

```
; String String -> String

; generates the pretense for money transfer for the victim fst
last

(check-range (string-length (body "Tillman" "Rendel")) 50 300)

(define (body fst lst)
  (string-append
```

```
    "Herr Gadafi aus Libyen ist gestorben und hat Sie, "
    fst
    ", in seinem Testament als Alleinerben eingesetzt.\n"
    "Lieber " fst " " lst ", gegen eine kleine Bearbeitungsge-
bühr überweise ich das Vermögen."))
```

Ein Beispiel für einen Aufrufer, der sich nicht an die Spezifikation hält und unzulässig an die Implementation der koppelt wäre einer, der Folgetext für den Brief definiert, der sich auf Details des Textes wie Namen und Orte bezieht, die nicht in der Spezifikation gennant werden.

Halten sich jedoch alle Aufrufer an das Geheimnisprinzip, so ist sichergestellt, dass sich die Implementation von `body` weitgehend ändern läßt, solange es ein plausibler Text gemäß der Aufgabenbeschreibung ist, der zwischen 50 und 300 Zeichen lang ist.

Dieses Beispiel illustriert weiterhin, wieso es häufig sinnvoll ist, in Tests nur bestimmte Eigenschaften des Ergebnisses zu testen, aber nicht das Ergebnis exakt vorzuschreiben. Im Beispiel wird nur getestet, dass die Länge des generierten Strings zwischen 50 und 300 ist — dies wird damit zum Teil der Spezifikation, auf die sich Aufrufer verlassen können. Würde hingegen auf einen festen Ausgabestring getestet, so würde der Test zu viele Details über die Implementierung verraten und damit das Information Hiding Prinzip nutzlos gemacht.

4 Batchprogramme und interaktive Programme

Ein Programm, wie wir es bisher kennen, besteht immer aus Ausdrücken, Funktionsdefinitionen und Variablendefinitionen. Aus der Perspektive der Benutzung eines Programms können wir jedoch mehrere Unterarten dieser Programme unterscheiden. Zwei wichtige Unterarten sind *Batchprogramme* und *interaktive Programme*.

4.1 Batchprogramme

Batchprogramme bestehen aus einer Hauptfunktion. Die Hauptfunktion nutzt Hilfsfunktionen, die wiederum weitere Hilfsfunktionen benutzen können. Ein Batchprogramm aufzurufen bedeutet, dass die Hauptfunktion mit der Eingabe des Programms aufgerufen wird und nach Beendigung der Hauptfunktion mit einem Ergebnis zurückkehrt. Während die Hauptfunktion ausgewertet wird, gibt es keine Interaktion mit dem Benutzer mehr.

Häufig wird Batchprogrammen ihre Eingabe in Form von Kommandozeilenparametern oder Standardströmen übergeben. Weitere Eingaben können aus Dateien kommen, deren Name beispielsweise als Kommandozeilenparameter übergeben wurde. Die Ausgabe wird häufig über den Standardausgabestrom (stdout) ausgegeben. Ein gutes Beispiel für Batchprogramme sind die Kommandozeilen-Tools der Unix Welt, wie zum Beispiel *ls*, *grep* oder *find*. Auch DrRacket gibt es in einer Version als Batchprogramm, *raco*, siehe `http://docs.racket-lang.org/raco/index.html`.

Recherchieren Sie, was Standardströme (*standard streams*) wie *stdin* und *stdout* in Unix sind.

Batchprogramme verhalten sich also in gewissem Sinne wie unsere Funktionen, die wir definieren: Wenn man Ihnen eine Eingabe gibt, rechnen sie eine Zeit lang selbstständig und geben dann das Ergebnis zurück. Sie wissen bereits, dass Funktionen sehr flexibel miteinander verknüpft und kombiniert werden können. So ist es daher auch mit Batchprogrammen, die sehr häufig in Skripten oder auf der Kommandozeile miteinander kombiniert werden. Möchte man beispielsweise in einer Unix Shell die Liste aller Dateien des aktuellen Verzeichnisses wissen, die im März modifiziert wurden und diese dann nach ihrer Größe sortieren, so können wir das erreichen, indem wir einfache Batchprogramme wie `ls`, `grep` und `sort` miteinander verknüpfen:

```
$ ls -l | grep "Mar" | sort +4n
-rwxr-xr-x  1 klaus    Admin      193 Mar  7 19:50 sync.bat
-rw-r--r--  1 klaus    Admin      317 Mar 30 12:53 10-modules.bak
-rw-r--r--  1 klaus    Admin      348 Mar  8 12:30 arrow-big.png
-rw-r--r--  1 klaus    Admin      550 Mar 30 13:02 10-
modules.scrbl
-rw-r--r--  1 klaus    Admin     3611 Mar  8 12:35 arrow.png
-rw-r--r--  1 klaus    Admin     4083 Mar 27 15:16 marburg-
utils.rkt
-rw-r--r--  1 klaus    Admin     4267 Mar 27 15:15 marburg-
utils.bak
-rw-r--r--  1 klaus    Admin     7782 Mar 30 13:02 10-
modules.html
-rw-r--r--  1 klaus    Admin    14952 Mar  2 13:49 rocket-s.jpg
```

Ein gutes Beispiel für ein Batchprogramm in BSL ist die `letter` Funktion aus Abschnitt §3.1 “Funktionale Dekomposition”. Wir rufen es im Interaktionsbereich mit der gewünschten Eingabe auf und erhalten dann die Ausgabe. Es ist möglich, dieses Programm auch außerhalb von DrRacket als Batchprogramm zu verwenden. Allerdings muss in diesem Fall die Eingabe anders übergeben werden, nämlich in Form von Kommandozeilenparametern. Wie dies aussehen kann, zeigt das folgende Programm. Um auf die Kommandozeilenparameter zuzugreifen, sind einige Sprachkonstrukte notwendig, die Sie bisher noch nicht kennen. Versuchen Sie daher nicht, die Details des unten stehenden Programms zu verstehen; es soll lediglich illustrieren, dass man Racket-Programme ohne DrRacket als Batchprogramme ausführen kann.

```
(require racket/base)

(define (letter fst lst signature-name)
  (string-append
    (opening lst)
    "\n"
    (body fst lst)
    "\n"
    (closing signature-name)))

(define (opening lst)
  (string-append "Sehr geehrte(r) Herr/Frau " lst ","))

(define (body fst lst)
  (string-append
   "Herr Gadafi aus Libyen ist gestorben.\n"
   "Er hat Sie, "  fst  ", in seinem Testament als Alleinerben
eingesetzt.\n"
   "Lieber " fst " " lst ", gegen eine kleine Bearbeitungsge-
buehr\n ueberweise ich das Vermoegen."))

(define (closing signature-name)
  (string-append
   "Mit freundlichen Gruessen,\n"
   signature-name))

(define args (current-command-line-arguments))

(if (= (vector-length args) 3)
    (display (letter (vector-ref args 0)
                     (vector-ref args 1)
                     (vector-ref args 2)))
    (error "Bitte genau drei Parameter uebergeben"))
```

Falls dieses Programm in der Datei `letter.rkt` abgespeichert ist, so können Sie dieses zum Beispiel in einer Unix oder DOS Shell so aufrufen:

```
$ racket letter.rkt Tillmann Rendel Klaus
```
und erhalten dann die folgende Ausgabe:
```
Sehr geehrte(r) Herr/Frau Rendel,
Herr Gadafi aus Libyen ist gestorben.
Er hat Sie, Tillmann, in seinem Testament als Alleinerben
eingesetzt.
Lieber Tillmann Rendel, gegen eine kleine Bearbeitungsgebuehr
 ueberweise ich das Vermoegen.
Mit freundlichen Gruessen,
Klaus Ostermann
```

Dieses Batchprogramm können Sie nun auch mit anderen Kommandozeilenprogrammen verknüpfen; beispielsweise können Sie es mit dem *wc* Programm verknüpfen um die die Anzahl der Wörter in dem generierten Text zu bestimmen:

```
$ racket letter.rkt Tillmann Rendel Klaus | wc -w
     35
```

Das Programm `racket`, das in diesem Befehl aufgerufen wird, ist die Kommandozeilenversion von DrRacket. Sie können Racket Programme aber natürlich auch ganz ohne die Racket Umgebung ausführen. Beispielsweise können Sie mit dem Menüeintrag Racket -> Programmdatei eine ausführbare Datei erzeugen, die unabhängig von DrRacket auf jedem Rechner mit dem passenden Betriebssystem ausgeführt werden kann.

Wenn Sie in Ihren BSL Programmen Dateien lesen oder schreiben möchten, können Sie hierzu das Teachpack `2htdp/batch-io` verwenden.

Zusammenfassend läßt sich sagen, dass die Stärke von Batchprogrammen ist, dass sie sich leicht und auf vielfältige Weise mit anderen Batchprogrammen kombinieren lassen, entweder auf der Kommandozeile oder auch automatisiert innerhalb von Batch-Files oder Shell-Skripten. Batchprogramme sind definitionsgemäß nicht für Interaktionen mit dem Benutzer geeignet. Sie sind jedoch sehr häufig Bausteine für interaktive Programme.

4.2 Interaktive Programme

Interaktive Programme bestehen aus mehreren Hauptfunktionen sowie einem Ausdruck, der den Computer informiert, welche dieser Funktionen welche Art von Eingaben verarbeitet und welche der Funktionen Ausgaben produziert. Jeder dieser Hauptfunktionen kann natürlich wieder Hilfsfunktionen verwenden. Wir werden zur Konstruktion interaktiver Programme das Universe Teachpack verwenden.

4.2.1 Das Universe Teachpack

Das "universe" Teachpack unterstützt die Konstruktion interaktiver Programme: Programme, die auf Zeitsignale, Mausklicks, Tastatureingaben oder Netzwerkverkehr reagieren und grafische Ausgaben erzeugen. Wir werden die Funktionsweise dieses Teachpacks anhand des folgenden Beispiels erklären. Bitte probieren Sie vor dem weiterlesen aus, was dieses Programm macht.

```
; WorldState is a number
; interp. the countdown when the bomb explodes

; WorldState -> WorldState
; reduces countdown by one whenever a clock tick occurs
(check-expect (on-tick-event 42) 41)

(define (on-tick-event world) (- world 1))


; WorldState Number Number MouseEvent -> WorldState
; decreases countdown by 100 when mouse button is pressed
(check-expect (on-mouse-event 300 12 27 "button-down") 200)
(check-expect (on-mouse-event 300 12 27 "button-up") 300)

(define (on-mouse-event world mouse-x mouse-y mouse-event)
  (if (string=? mouse-event "button-down")
      (- world 100)
      world))


; WorldState -> Image
; renders the current countdown t as an image
(check-expect (image? (render 100)) true)
(check-expect (image? (render 0)) true)
(define (render world)
  (if (> world 0)
      (above (text (string-append "Countdown for the bomb: "
                                  (number->string world))
                   30 "red")
             (text "Click to disarm!" 30 "red"))
      (text "Boooom!!" 60 "red")))


; WorldState -> Boolean
; the program is over when the countdown is <= 0
(check-expect (end-of-the-world 42) false)
(check-expect (end-of-the-world 0) true)
(define (end-of-the-world world) (<= world 0))

; install all event handlers; initialize world state to 500
(big-bang 500
          (on-tick on-tick-event 0.1)
          (on-mouse on-mouse-event)
          (to-draw render)
          (stop-when end-of-the-world))
```

Ein zentrales Konzept bei interaktiven Programmen ist das des *WorldState*s. Eine interaktive Anwendung befindet sich typischerweise in einem bestimmten *Zustand*, den wir WorldState nennen.
Der Zustand bei einem Pacman-Spiel umfasst beispielsweise die gegenwärtige Position aller Spielfiguren und den aktuellen Punktestand. Im Programm oben besteht der aktuelle Zustand lediglich aus einer Zahl, die den aktuellen Countdown zur Bombenexplosion enthält.

Falls Sie Pacman nicht kennen, suchen Sie bitte unverzüglich eine Online-Version von Pacman im Internet und spielen eine Runde!

Der Zustand eines interaktiven Programms ändert sich, wenn bestimmte *Ereignisse* eintreten. Ereignisse können zum Beispiel das Drücken von Tasten auf der Tastatur, Aktionen mit der Maus oder der Ablauf bestimmter Zeitintervalle (*Timer*) sein. Auf Ereignisse wird in Form von *Event Handlern* reagiert. Ein Event Handler ist eine Funktion, die den gegenwärtigen WorldState sowie weitere Informationen über das eingetretene Ereignis als Eingabe erhält und einen neuen WorldState als Ausgabe produziert.

Im Beispiel oben werden zwei Event Handler definiert, `on-mouse-event` für Mausevents und `on-tick-event` für Timer-Ereignisse. Das Intervall zwischen zwei Timer-Ereignissen werden wir später auf 0.1 Sekunden festlegen. In der Anwendung oben haben wir uns dafür entschieden, jedesmal beim Eintreten eines Timer-Events den aktuellen Countdown um eins zu reduzieren.

Die `"on-mouse-event"` Funktion bekommt als Eingabe außer dem aktuellen WorldState noch die Position der Maus sowie die genaue Art des Mausereignisses als Eingabe. In unserem Beispiel wird der Countdown bei jedem Klick auf die Maus um 100 erniedrigt.

Die `render` Funktion produziert aus dem aktuellen WorldState ein Bild, welches den aktuellen Zustand des Programms grafisch darstellt. Diese Funktion wird jedesmal aufgerufen, wenn sich der aktuelle WorldState geändert hat.

Die letzte Funktion, `end-of-the-world`, dient dazu, zu überprüfen, ob die Anwendung beendet werden soll. Hierzu prüft sie den WorldState und produziert als Ergebnis einen Wahrheitswert. In unserem Beispiel möchten wir die Anwendung beenden, nachdem die Bombe explodiert ist.

Der letzte `big-bang` Ausdruck dient dazu, den WorldState zu initialisieren und alle Event Handler zu installieren. Der `big-bang` Operator ist eine Sonderform, also keine normale BSL Funktion. Das erste Argument von `big-bang` ist der initiale WorldState. Da wir uns dafür entschieden haben, dass in dieser Anwendung der WorldState eine Zahl ist, die den aktuellen Countdown repräsentiert, wählen wir einen initialen Countdown, `500`.

Die Unterausdrücke wie `(on-tick on-tick-event)` sind keine Funktionsaufrufe sondern spezielle Klauseln des `big-bang` Operators, in denen man die Event-Handler Funktionen sowie die Funktionen zum Zeichnen (`to-draw`) und Beenden (`stop-when`) des Programms angibt. Selbstverständlich hätten wir den Event-Handler Funktionen auch andere Namen geben können – der `big-bang` ist die Stelle, wo definiert wird, welcher Ereignistyp mit welcher Event-Handler Funktion verknüpft wird.

Einige dieser Klauseln erwarten außer dem Namen der Event-Handler Funktion noch weitere Argumente. So sagt beispielsweise die Zahl in der `(on-tick on-tick-event 0.1)`, dass alle 0,1 Sekunden ein Timer-Event ausgelöst werden soll. Man kann

diese Angabe auch weglassen; dann wird als Standardwert 1/28 Sekunde angenommen. Es gibt noch eine Reihe weiterer Klauseln und Varianten der oben stehenden Klauseln; für weitere Informationen dazu siehe die Dokumentation des universe Teachpacks. Bis auf die `to-draw` Klauseln sind alle diese Klauseln optional; ist für einen Eventtyp kein Event Handler installiert, werden diese Events ignoriert.

Wird ein interaktives Programm (im folgenden auch *World Programm*) beendet, so gibt der `big-bang` Ausdruck den letzten WorldState zurück.

Nehmen wir beispielsweise an, dass in folgender Reihenfolge folgende Events eintreten:

1. ein Timerevent
2. ein Timerevent
3. ein Mausevent (die Maus wurde an die Position (123,456) bewegt)
4. ein Mausevent (eine Maustaste wurde an der Stelle (123,456) gedrückt)

Nehmen wir an, der Zustand vor Eintreten dieser Events ist `500`. Nach Eintreten des jeweiligen Ereignisses ist der aktuelle Zustand:

1. `499`, also `(on-tick-event 500)`
2. `498`, also `(on-tick-event 499)`
3. `498`, also `(on-mouse-event 498 123 456 "move")`
4. `398`, also `(on-mouse-event 498 123 456 "button-down")`

Da Event Handler nur Funktionen sind, kann man das letzte Resultat auch durch die *Funktionskomposition* der Event Handler berechnen:

```
(on-mouse-event
  (on-mouse-event
    (on-tick-event
      (on-tick-event 500))
    123 456 "move")
  123 456 "button-down")
```

Die Reihenfolge der Ereignisse bestimmt also die Reihenfolge, in der die Event Handler Funktionen hintereinandergeschaltet werden. Möchte man, zum Beispiel in Tests, das Eintreten einer bestimmten Sequenz von Ereignissen simulieren, so kann man dies also einfach durch Komposition der entsprechenden Event Handler tun. Dies ist wichtig, denn viele interessante Situationen (die man testen möchte) ergeben sich erst in der Komposition verschiedener Ereignisse.

5 Datendefinition durch Alternativen: Summentypen

Die Datentypen, die wir bisher kennengelernt und genutzt haben, umfassen Zahlen, Strings, Wahrheitswerte sowie in informellen Datendefinitionen definierte Datentypen wie `Temperatur`.

Um realistische Anwendungen zu programmieren, ist es hilfreich, ein größeres und flexibel erweiterbares Repertoire an Datentypen zu haben. In diesem Kapitel schauen wir uns an, wie man mit Datentypen unterschiedliche Situationen unterscheiden kann und wie diese Datentypen den Entwurf von Programmen beeinflussen.

5.1 Aufzählungstypen

Häufig hat man den Fall, dass ein Datentyp nur eine endliche aufzählbare Menge von Werten enthält. Wir nennen solche Datentypen *Aufzählungstypen* oder *Enumerationstypen*. Hier ist ein Beispiel, wie ein Aufzählungstyp definiert wird:

```
; A TrafficLight shows one of three colors:
; - "red"
; - "green"
; - "yellow"
; interp. each element of TrafficLight represents which col-
ored
; bulb is currently turned on
```

Das Programmieren mit Aufzählungstypen ist sehr einfach. Wenn eine Funktion einen Parameter hat, der einen Aufzählungstyp hat, dann enthält das Programm typischerweise eine Fallunterscheidung, in denen alle Alternativen des Aufzählungstyps separat voneinander behandelt werden. Hier ein Beispiel:

```
; TrafficLight -> TrafficLight
; given state s, determine the next state of the traffic light

(check-expect (traffic-light-next "red") "green")

(define (traffic-light-next s)
  (cond
    [(string=? "red" s) "green"]
    [(string=? "green" s) "yellow"]
    [(string=? "yellow" s) "red"]))
```

Manchmal ist eine Funktion auch nur an einer Teilmenge der möglichen Werte eines Aufzählungstypen interessiert. In diesem Fall werden nur die interessanten Fälle abgefragt und der Rest ignoriert. Ein Beispiel dafür ist die `on-mouse-event` Funktion aus §4.2.1 “Das Universe Teachpack”, die eine Eingabe vom Aufzählungstypen `MouseEvent` erhält. `MouseEvent` ist folgendermaßen definiert:

```
; A MouseEvt is one of these strings:
```

```
; - "button-down"
; - "button-up"
; - "drag"
; - "move"
; - "enter"
; - "leave"
```

In der Funktion `on-mouse-event` interessieren wir uns nur für die Alternative `"button-down"`. Für alle anderen Alternativen von `MouseEvent`, lassen wir den Zustand unverändert.

Es kann auch sein, dass ein Funktionsparameter einen Aufzählungstyp hat, aber die Funktion trotzdem keine Fallunterscheidung vornimmt, weil hierzu eine Hilfsfunktion aufgerufen wird. Dennoch ist es eine gute Heuristik, im Schritt 3 des Entwurfsrezepts aus Abschnitt §3.3.3 “Entwurfsrezept zur Funktionsdefinition” mit einem Funktionstemplate zu starten, welches alle Fälle des Parameters unterscheidet. Beispielsweise würde das Template für die `traffic-light-next` Funktion von oben so aussehen:

```
(define (traffic-light-next s)
  (cond
    [(string=? "red" s) ...]
    [(string=? "green" s) ...]
    [(string=? "yellow" s) ...]))
```

Wie sie sehen, können sie einen großen Teil einer Funktionsdefinition quasi mechanisch generieren, indem Sie systematisch den Schritten aus dem Entwurfsrezept folgen.

5.2 Intervalltypen

Betrachten Sie ein Programm, welches ein Ufo beim landen zeigen soll. Ein Programm hierzu könnte wie folgt aussehen:

```
; constants:
(define WIDTH 300)
(define HEIGHT 100)

; visual constants:
(define MT (empty-scene WIDTH HEIGHT))
(define UFO
  (overlay (circle 10 "solid" "green")
           (rectangle 40 2 "solid" "green")))

(define BOTTOM (- HEIGHT (/ (image-height UFO) 2)))

; A WorldState is a number.
; interp. height of UFO (from top)

; WorldState -> WorldState
```

```
; compute next location of UFO
(define (nxt y)
  (+ y 1))

; WorldState -> Image
; place UFO at given height into the center of MT
(define (render y)
  (place-image UFO (/ WIDTH 2) y MT))

; WorldState -> Boolean
; returns #true when UFO has reached the bottom, i.e. y >=
BOTTOM
(define (end-of-the-world y) (>= y BOTTOM))

; wire everything together; start descend at position 0
(big-bang 0 (on-tick nxt) (to-draw render) (stop-when end-of-
the-world))
```

Betrachten Sie nun folgende Erweiterung der Problemstellung:

The status line should say "descending" when the UFO's height is above one third of the height of the canvas. It should switch to "closing in" below that. And finally, when the UFO has reached the bottom of the canvas, the status should notify the player that the UFO has "landed."

Der Datentyp, der sich in dieser Problemstellung versteckt, ist kein Enumerationstyp, denn es gibt eine unendliche Zahl unterschiedlicher Höhen (oder, sofern wir nur die ganzen Zahlen zählen, so viele unterschiedliche Höhen, dass es unpraktisch wäre, hierfür einen Aufzählungstypen zu definieren).

Daher verwenden wir *Intervalle*, um zusätzliche Struktur auf geordneten Datentypen wie Zahlen oder Strings zu definieren. Im folgenden konzentrieren wir uns auf (ganzzahlige oder nicht-ganzzahlige) Zahlen. Ein Intervall wird durch seine *Grenzen* definiert. Ein Intervall hat entweder eine unter und obere Grenze, oder es hat nur eine dieser Grenzen und ist zur anderen Seite offen.

In unserem Beispiel können wir die Datendefinition für den WorldState als Intervall ausdrücken, um eine Datendefinition zu haben, die die Intention der Problemstellung besser erfasst.

```
; constants:
(define CLOSE (* 2 (/ HEIGHT 3)))

; A WorldState is a number. It falls into one of three
intervals:
; - between 0 and CLOSE
; - between CLOSE and BOTTOM
; - at BOTTOM
; interp. height of UFO (from top)
```

Nicht alle Funktionen, die einen Intervalltypen als Argument bekommen, nutzen

diese zusätzliche Struktur. So kann zum Beispiel die `render` Funktion von oben so bleiben wie sie ist, weil das Intervall nur für die Statusanzeige aber nicht für das Ufo relevant ist.

Funktionen, die jedoch die zusätzliche Struktur des Intervalls benötigen, enthalten typischerweise einen `cond` Ausdruck, der die unterschiedlichen Intervalle unterscheidet.

```
; WorldState -> Image
; add a status line to the scene create by render
(define (render/status y)
  (cond
    [(<= 0 y CLOSE) (above (text "descending" 12 "black") (render y))]
    [(< CLOSE y BOTTOM) (above (text "closing
in" 12 "black") (render y))]
    [(= y BOTTOM) (above (text "landed" 12 "black") (render y))]))
```

Es empfiehlt sich, diesen `cond` Ausdruck im Rahmen der Templatekonstruktion aus dem Entwurfsrezept direkt in das Template mit aufzunehmen. Allerdings findet die Fallunterscheidung nicht notwendigerweise als erstes statt sondern kann auch tiefer im Funktionsbody stattfinden. Dies bietet sich in unserem Beispiel an, denn wir haben gegen das DRY Prinzip verstossen (Wenn wir die Farbe des Textes beispielsweise auf "red" ändern möchten, müssten wir drei Zeilen ändern). Deshalb ist es vorteilhaft, den konditionalen Ausdruck nach innen zu ziehen:

```
; WorldState -> Image
; add a status line to the scene create by render
(define (render/status y)
  (above
   (text
     (cond
      [(<= 0 y CLOSE) "descending"]
      [(< CLOSE y BOTTOM) "closing in"]
      [(= y BOTTOM) "landed"])
     12 "black")
   (render y)))
```

Nun muss nur noch der `big-bang` Ausdruck angepasst werden, so dass `render/status` und nicht mehr `render` zum Zeichnen verwendet wird.

```
; wire everything together; start descend at position 0
(big-bang 0 (on-tick nxt) (to-draw render/status) (stop-
when end-of-the-world))
```

Intervalle liefern uns außer neuen Funktionstemplates auch Anhalte zum Testen: Typischerweise möchte man einen Testcase für jedes Intervall und insbesondere für die Intervallgrenzen haben.

5.3 Summentypen

Intervalltypen unterscheiden unterschiedliche Teilmengen der Zahlen (oder anderer geordneter Werte). Aufzählungstypen zählen die unterschiedlichen Elemente des Typs Wert für Wert auf.

Summentypen verallgemeinern Intervalltypen und Aufzählungstypen. Mit Summentypen können existierende Datentypen und individuelle Werte beliebig miteinander kombiniert werden. Ein Summentyp gibt verschiedene Alternativen an, von denen jeder Wert dieses Typs genau eine Alternative erfüllt.

Betrachten wir als Beispiel die `string->number` Funktion, die einen String in eine Zahl konvertiert, falls dies möglich ist, und andernfalls `#false` zurückgibt.

Hier ist die Definition eines Summentyps, der das Verhalten dieser Funktion beschreibt:

```
; A MaybeNumber is one of:
; - #false
; - a Number
; interp. a number if successful, else false.
```

Damit können wir für `string->number` folgende Signatur definieren:

```
; String -> MaybeNumber
; converts the given string into a number;
; produces #false if impossible
(define (string->number str) ...)
```

Summentypen werden manchmal auch Vereinigungstypen (union types) genannt. In HTDP/2e werden sie *Itemizations* genannt.

In der Dokumentation der HTDP-Sprachen ist die Signatur von `string->number` so angegeben:

```
String -> (union Number #false)
```

Der Operator `union` steht für die "on-the-fly" Konstruktion eines anonymen Summentyps mit der gleichen Bedeutung wie unser `MaybeNumber` oben.

Was macht man mit einem Wert, der einen Summentyp hat? Wie bei Aufzählungs- und Intervalltypen auch ist typischerweise die einzige sinnvolle Operation die, welche die unterschiedlichen Alternativen voneinander trennt, beispielsweise im Rahmen eines `cond` Ausdrucks. Hier ein Beispiel:

```
; MaybeNumber -> MaybeNumber
; adds 3 to a if it is a number; returns #false otherwise
(check-expect (add3 5) 8)
(check-expect (add3 #false) #false)
(define (add3 a)
  (cond [(number? a) (+ a 3)]
        [else #false]))
```

Funktion mit Summentypen entscheiden sich in ihrer Entwurfsmethodik etwas von den Funktionen, die wir bisher kennengelernt haben. Deswegen gehen wir nochmal durch unser Entwurfsrezept (§3.3.3 “Entwurfsrezept zur Funktionsdefinition”) und beschreiben, wie sich der Entwurf von Funktionen mit Summentypen vom allgemeinen Entwurfsrezept unterscheidet.

The tax on an item is either an absolute tax of 5 or 10 currency units, or a linear tax. Design a function that computes the price of an item after applying its tax.

5.3.1 Entwurf mit Summentypen

Das Entwurfsrezept für den Entwurf von Funktionen ergänzen wir wie folgt:

1. Falls die Problemstellung Werte in unterschiedliche Klassen unterteilt, so sollten diese Klassen durch eine Datendefinition explizit definiert werden.

   Beispiel: Die Problemstellung oben unterscheidet drei verschiedene Arten der Steuer. Dies motiviert folgende Datendefinition:

   ```
   ; A Tax is one of
   ; - "absolute5"
   ; - "absolute10"
   ; - a Number representing a linear tax rate in percent
   ```

2. Im zweiten Schritt ändert sich nichts, außer dass typischerweise der definierte Datentyp in der Signatur verwendet wird.

   Beispiel:

   ```
   ; Number Tax -> Number
   ; computes price of an item after applying tax
   (define (total-price itemprice tax) 0)
   ```

3. Bzgl. der Tests ist es ratsam, pro Alternative des Datentypen mindestens ein Beispiel zu haben. Bei Intervallen sollten die Grenzen der Intervalle getestet werden. Gibt es mehrere Parameter mit Summentypen, so sollten alle Kombinationen der Alternativen getestet werden.

   Beispiel:

   ```
   (check-expect (total-price 10 "absolute5") 15)
   (check-expect (total-price 10 "absolute10") 20)
   (check-expect (total-price 10 25) 12.5)
   ```

4. Die größte Neuerung ist die des Templates für die Funktion. Im Allgemeinen *folgt die Struktur der Funktionen aus der Struktur der Daten*. Dies bedeutet, dass in den meisten Fällen der Funktionskörper mit einem `cond` Ausdruck startet, der die unterschiedlichen Fälle des Summentypen unterscheidet.

Wieso ist es in diesem Beispiel die Reihenfolge der Zweige des `cond` Ausdrucks wichtig?

```
(define (total-price itemprice tax)
  (cond [(number? tax) ...]
        [(string=? tax "absolute5") ...]
        [(string=? tax "absolute10") ...]))
```

5. Im fünften Schritt werden die `...` Platzhalter durch den korrekten Code ersetzt. Hier ist es sinnvoll, jeden Zweig des `cond` Ausdrucks einzeln und nacheinander durchzugehen und zu implementieren. Der Vorteil ist, dass Sie sich bei dieser Art der Implementierung immer nur um einen der Fälle kümmern müssen und alle anderen Fälle ignorieren können.

   Möglicherweise stellen Sie während dieses Schritts fest, dass es sinnvoll ist, den `cond` Ausdruck nach innen zu ziehen (ähnlich wie in `create-rocket-scene-v6` in Abschnitt §2.6.2 "DRY Redux"), oder vielleicht müssen Sie gar nicht alle Fälle unterscheiden, oder vielleicht möchten Sie die Unterscheidung auch in eine Hilfsfunktion auslagern – dennoch ist es in der Regel sinnvoll, mit diesem Template zu starten, selbst wenn am Ende ihre Funktion anders aussieht.

   Diese Unterscheidung der Fälle ist ein Beispiel für ein allgemeineres Entwurfskonzept für komplexe Systeme, welches sich *separation of concerns* (Trennung der Belange) nennt.

   Beispiel:

   ```
   (define (total-price itemprice tax)
     (cond [(number? tax) (* itemprice (+ 1 (/ tax 100)))]
           [(string=? tax "absolute5") (+ itemprice 5)]
           [(string=? tax "absolute10") (+ itemprice 10)]))
   ```

6. Im letzten Schritt ändert sich nichts, aber überprüfen Sie, dass Sie Testfälle für alle Alternativen definiert haben.

5.4 Unterscheidbarkeit der Alternativen

Was wäre, wenn wir in unserem letzten Beispiel statt zwei absoluten Steuersätzen ein kontinuierliches Spektrum hätten? Wir könnten unsere Datendefinition wie folgt abändern:

```
; A Tax is one of
; - a Number representing an absolute tax in currency units
; - a Number representing a linear tax rate in percent
```

So weit so gut – aber wie können wir in einem `cond` Ausdruck diese Fälle unterscheiden? Wir haben zwar das Prädikat `number?`, aber damit können wir nicht zwischen diesen beiden Fällen unterscheiden.

In unseren Summentypen ist es wichtig, dass man eindeutig unterscheiden kann, welche Alternative in einem Wert, der einen Summentyp hat, gewählt wurde. Daher müssen die Mengen der möglichen Werte für jede Alternative disjunkt sein.
Falls sie nicht disjunkt sind, muss man zu jedem Wert eine zusätzliche Information abspeichern, die aussagt, zu welcher Alternative dieser Wert gehört: ein sogenanntes *tag* ("Etikett", "Anhänger"). Mit den Mitteln, die wir bisher kennengelernt haben, können wir den oben gewünschten Datentyp nicht sinnvoll ausdrücken. Im nächsten Kapitel

Die Variante der Summentypen, die wir verwenden, bezeichnet man deshalb auch als *untagged unions*.

werden wir jedoch ein Sprachkonstrukt kennenlernen, mit dem man solche *tags* und damit auch solche Datentypen strukturiert definieren kann.

6 Datendefinition durch Zerlegung: Produkttypen

Dieser Teil des Skripts basiert auf [HTDP/2e] Kapitel 5

Nehmen Sie an, Sie möchten mit dem "universe" Teachpack ein Programm schreiben, welches einen Ball simuliert, der zwischen vier Wänden hin und her prallt. Nehmen wir der Einfachheit halber an, der Ball bewegt sich konstant mit zwei Pixeln pro Zeiteinheit.

Wenn Sie sich an das Entwurfsrezept halten, ist ihre erste Aufgabe, eine Datenrepräsentation für all die Dinge, die sich ändern, zu definieren. Für unseren Ball mit konstanter Geschwindigkeit sind dies zwei Eigenschaften: Die aktuelle Position sowie die Richtung, in die sich der Ball bewegt.

Wäre es sinnvoll, Varianten des "universe" Teachpacks zu haben, in denen der WorldState zum Beispiel durch zwei Werte repräsentiert wird?

Der WorldState im "universe" Teachpack ist allerdings nur ein einzelner Wert. Die Werte, die wir bisher kennen, sind allerdings stets einzelne Zahlen, Bilder, Strings oder Wahrheitswerte — daran ändern auch Summentypen nichts. Wir müssen also irgendwie mehrere Werte so zusammenstellen können, dass sie wie ein einzelner Wert behandelt werden können.

Es wäre zwar denkbar, zum Beispiel zwei Zahlen in einen String reinzukodieren und bei Bedarf wieder zu dekodieren (in der theoretischen Informatik sind solche Techniken als *Gödelisierung* bekannt), aber für die praktische Programmierung sind diese Techniken ungeeignet.

Jede höhere Programmiersprache hat Mechanismen, um mehrere Daten zu einem Datum zusammenzupacken und bei Bedarf wieder in seine Bestandteile zu zerlegen. In BSL gibt es zu diesem Zweck *Strukturen* (*structs*). Man nennt Strukturdefinitionen auch oft *Produkte*, weil Sie dem Kreuzprodukt von Mengen in der Mathematik entsprechen. Aus dieser Analogie ergibt sich übrigens auch der Name 'Summentyp' aus dem letzten Kapitel, denn die Vereinigung von Mengen wird auch als die Summe der Mengen bezeichnet. In manchen Sprachen werden Strukturen auch *Records* genannt.

6.1 Die `posn` Struktur

Eine Position in einem Bild wird durch zwei Zahlen eindeutig identifiziert: Die Distanz vom linken Rand und die Distanz vom oberen Rand. Die erse Zahl nennt man die x-Koordinae und die zweite die y-Koordinae.

In BSL werden solche Positionen mit Hilfe der `posn` Struktur repräseniert. Eine `posn` (Aussprache: Position) ist also *ein* Wert, der *zwei* Werte enthält.

Im Deutschen wäre es eigentlich korrekt, von einem *Exemplar* einer Struktur zu sprechen. Es hat sich jedoch eingebürgert, analog zum Wort *instance* im Englischen von einer Instanz zu reden, deshalb werden auch wir diese grammatikalisch fragwürdige Terminologie verwenden.

Wir können eine *Instanz* der `posn` Struktur mit der Funktion `make-posn` erzeugen. Die Signatur dieser Funktion ist *Number Number -> Posn*.

Beispiel: Diese drei Ausdrücke erzeugen jeweils eine Instanz der `posn` Struktur.

```
(make-posn 3 4)
(make-posn 8 6)
(make-posn 5 12)
```

Eine `posn` hat den gleichen Status wie Zahlen oder Strings in dem Sinne, dass Funktionen Instanzen von Strukturen konsumieren oder produzieren können.

Betrachten Sie nun eine Funktion, die die Distanz einer Position von der oberen linken Bildecke berechnet. Hier ist die Signatur, Aufgabenbeschreibung und der Header einer solchen Funktion:

```
; Posn -> Number
; to compute the distance of a-posn to the origin
(define (distance-to-0 a-posn) 0)
```

Was neu ist an dieser Funktion ist, dass sie nur einen Parameter, `a-posn`, hat, in dem aber beide Koordinaten übergeben werden. Hier sind einige Tests die verdeutlichen, was `distance-to-0` berechnen soll. Die Distanz kann natürlich über die bekannte Pythagaros Formel berechnet werden.

```
(check-expect (distance-to-0 (make-posn 0 5)) 5)
(check-expect (distance-to-0 (make-posn 7 0)) 7)
(check-expect (distance-to-0 (make-posn 3 4)) 5)
(check-expect (distance-to-0 (make-posn 8 6)) 10)
```

Wie sieht nun der Funktionsbody von `distance-to-0` aus? Offensichtlich müssen wir zu diesem Zweck die x- und y-Koordinate aus dem Parameter `a-posn` extrahieren. Hierzu gibt es zwei Funktionen `posn-x` und `posn-y`. Die erste Funktion extrahiert die x-Koordinate, die zweite die y-Koordinate. Hier zwei Beispiele die diese Funktionen illustrieren:

```
> (posn-x (make-posn 3 4))
3
> (posn-y (make-posn 3 4))
4
```

Damit wissen wir nun genug, um `distance-to-0` zu implementieren. Als Zwischenschritt definieren wir ein Template für `distance-to-0`, in welchem die Ausdrücke zur Extraktion der x- und y-Koordinate vorgezeichnet sind.

```
(define (distance-to-0 a-posn)
  (... (posn-x a-posn) ...
   ... (posn-y a-posn) ...))
```

Auf Basis dieses Templates ist es nun leicht, die Funktionsdefinition zu vervollständigen:

```
(define (distance-to-0 a-posn)
  (sqrt
    (+ (sqr (posn-x a-posn))
       (sqr (posn-y a-posn)))))
```

6.2 Strukturdefinitionen

Strukturen wie `posn` werden normalerweise nicht fest in eine Programmiersprache eingebaut. Stattdessen wird von der Sprache nur ein Mechanismus zur Verfügung

gestellt, um eigene Strukturen zu definieren. Die Menge der verfügbaren Strukturen kann also von jedem Programmierer beliebig erweitert werden.

Eine Struktur wird durch eine spezielle Strukturdefinition erzeugt. Hier ist die Definition der `posn` Struktur in BSL:

```
(define-struct posn (x y))
```

Im Allgemeinen hat eine Strukturdefinition diese Form:

```
(define-struct StructureName (FieldName ... FieldName))
```

Das Schlüsselwort `define-struct` bedeutet, dass hier eine Struktur definiert wird. Danach kommt der Name der Struktur. Danach folgen, eingeschlossen in Klammern, die Namen der *Felder* der Struktur.

Im Gegensatz zu einer normale Funktionsdefinition definiert man durch eine Strukturdefinition gleich einen ganzen Satz an Funktionen, und zwar wie folgt:

- Einen *Konstruktor* — eine Funktion, die soviele Parameter hat wie die Struktur Felder hat und eine Instanz der Struktur zurückliefert. Im Falle von `posn` heißt diese Funktion `make-posn`; im allgemeinen Fall heißt sie `make-StructureName`, wobei `StructureName` der Name der Struktur ist.
- Pro Feld der Struktur einen *Selektor* — eine Funktion, die den Wert eines Feldes aus einer Instanz der Struktur ausliest. Im Falle von `posn` heißen diese Funktionen `posn-x` und `posn-y`; im Allgemeinen heißen Sie `StructureName-FieldName`, wobei `StructureName` der Name der Struktur und `FielName` der Name des Feldes ist.
- Ein *Strukturprädikat* — eine boolsche Funktion, die berechnet, ob ein Wert eine Instanz dieser Struktur ist. Im Falle von `posn` heißt dieses Prädikat `posn?`; im Allgemeinen heißt es `StructureName?`, wobei `StructureName` der Name der Struktur ist.

Der Rest des Programms kann diese Funktionen benutzen als wären sie primitive Funktionen.

6.3 Verschachtelte Strukturen

Ein sich mit konstanter Geschwindigkeit bewegender Ball im zweidimensionalen Raum kann durch zwei Eigenschaften beschrieben werden: Seinen Ort und die Geschwindigkeit und Richtung in die er sich bewegt. Wir wissen bereits wie man den Ort eines Objekts im zweidimensionalen Raum beschreibt: Mit Hilfe der `posn` Struktur. Es gibt verschiedene Möglichkeiten, die Geschwindigkeit und Richtung eines Objekts zu repräsentieren. Eine dieser Möglichkeiten ist die, einen Bewegungsvektor anzugeben, zum Beispiel in Form einer Strukturdefinition wie dieser:

```
(define-struct vel (deltax deltay))
```

Der Name `vel` steht für *velocity*; `deltax` und `deltay` beschreiben, wieviele Punkte auf der x- und y-Achse sich das Objekt pro Zeiteinheit bewegt.

Der Grad der Veränderung von Einheiten wird in der Informatik (und anderswo) häufig als *Delta* bezeichnet, daher der Name der Felder.

Auf Basis dieser Definitionen können wir beispielsweise berechnen, wie sich der Ort eines Objekts ändert:

```
; posn vel -> posn
; computes position of loc after applying v
(check-expect (move (make-posn 5 6) (make-vel 1 2)) (make-
posn 6 8))
(define (move loc v)
  (make-posn
   (+ (posn-x loc) (vel-deltax v))
   (+ (posn-y loc) (vel-deltay v))))
```

Wie können wir nun ein sich bewegendes Balls selber repräsentieren? Wir haben gesehen, dass dieser durch seinen Ort und seinen Bewegungsvektor beschrieben wird. Eine Möglichkeit der Repräsentation wäre diese:

```
(define-struct ball (x y deltax deltay))
```

Hier ist ein Beispiel für einen Ball in dieser Repräsentation:

```
(define some-ball (make-ball 5 6 1 2))
```

Allerdings geht in dieser Repräsentation die Zusammengehörigkeit der Felder verloren: Die ersten zwei Felder repräsentieren den Ort, die anderen zwei Felder die Bewegung. Eine praktische Konsequenz ist, dass es auch umständlich ist, Funktionen aufzurufen, die etwas mit Geschwindigkeiten und/oder Bewegungen machen aber nichts über Bälle wissen, wie zum Beispiel `move` oben, denn wir müssen die Daten erst immer manuell in die richtigen Strukturen verpacken. Um `some-ball` mit Hilfe von `move` zu bewegen, müssten wir Ausdrucke wie diesen schreiben:

```
(move (make-posn (ball-x some-ball) (ball-y some-ball))
      (make-vel (ball-deltax some-ball) (ball-deltay some-
ball)))
```

Eine bessere Repräsentation *verschachtelt* Strukturen ineinander:

```
(define-struct ball (loc vel))
```

Diese Definition ist so noch nicht verschachtelt — dies werden wir in Kürze durch Datendefinitionen für Strukturen deutlich machen. Die Schachtelung können wir sehen, wenn wir Instanzen dieser Struktur erzeugen. Der Beispielball `some-ball` wird konstruiert, indem die Konstruktoren ineinander verschachtelt werden:

```
(define some-ball (make-ball (make-posn 5 6) (make-vel 1 2)))
```

In dieser Repräsentation bleibt die logische Gruppierung der Daten intakt. Auch der Aufruf von `move` gestaltet sich nun einfacher:

```
(move (ball-loc some-ball) (ball-vel some-ball))
```

Im Allgemeinen kann man durch Verschachtelung von Strukturen also Daten hierarchisch in Form eines Baums repräsentieren.

6.4 Datendefinitionen für Strukturen

Der Zweck einer Datendefinition für Strukturen ist, zu beschreiben, welche Art von Daten jedes Feld enthalten darf. Für einige Strukturen ist die dazugehörige Datendefinition recht offensichtlich:

```
(define-struct posn (x y))
; A Posn is a structure: (make-posn Number Number)
; interp. the number of pixels from left and from top
```

Hier sind zwei plausible Datendefinitionen für `vel` und `ball`:

```
(define-struct vel (deltax deltay))
; a Vel is a structure: (make-vel Number Number)
; interp. the velocity vector of a moving object

(define-struct ball (loc vel))
; a Ball is a structure: (make-ball Posn Vel)
; interp. the position and velocity of a ball
```

Eine Struktur hat jedoch nicht notwendigerweise genau eine zugehörige Datendefinition. Beispielsweise können wir Bälle nicht nur im zweidimensionalen, sondern auch im ein- oder drei-dimensionalen Raum betrachten. Im eindimensionalen Raum können Position und Velocity jeweils durch eine Zahl repräsentiert werden. Daher können wir definieren:

```
; a Ball1d is a structure: (make-ball Number Number)
; interp. the position and velocity of a 1D ball
```

Strukturen können also "wiederverwendet" werden. Sollte man dies stets tun wenn möglich?

Prinzipiell bräuchten wir nur eine einzige Struktur mit zwei Feldern und könnten damit alle Produkttypen mit zwei Komponenten kodieren. Beispielsweise könnten wir uns auch `vel` und `ball` sparen und stattdessen nur `posn` verwenden:

```
; a Vel is a structure: (make-posn Number Number)
; interp. the velocity vector of a moving object

(define-struct ball (loc vel))
; a Ball is a structure: (make-posn Posn Vel)
; interp. the position and velocity of a ball
```

Wir können sogar prinzipiell *alle* Produkttypen, auch solche mit mehr als zwei Feldern, mit Hilfe von `posn` ausdrücken, indem wir `posn` verschachteln.

Beipiel:

```
; a 3DPosn is a structure: (make-posn Number (make-posn Number
Number))
; interp. the x/y/z coordinates of a point in 3D space
```

Die Sprache LISP basierte auf diesem Prinzip: Es gab in ihr nur eine universelle Datenstruktur, die sogenannte *cons-Zelle*, die verwendet wurde, um alle Arten von Produkten zu repräsentieren. Die cons-Zelle entspricht folgender Strukturdefinition in BSL:

```
(define-struct cons-cell (car cdr))
```

Die Namen `cons`, `car` und `cdr` haben eine Historie, die für uns aber nicht relevant ist. `car` ist einfach der Name für die erste Komponente und `cdr` der für die zweite Komponente des Paars.

Ist diese Wiederverwendung von Strukturdefinitionen eine gute Idee? Der Preis, den man dafür bezahlt, ist, dass man die unterschiedlichen Daten nicht mehr unterscheiden kann, weil es pro Strukturdefinition nur ein Prädikat gibt. Unsere Empfehlung ist daher, Strukturdefinitionen nur dann in mehreren Datendefinitionen zu verwenden, wenn die unterschiedlichen Daten ein gemeinsames semantisches Konzept haben. Im Beispiel oben gibt es für `Ball` und `Ball1d` das gemeinsame Konzept des Balls im n-dimensionalen Raum. Es gibt jedoch kein sinnvolles gemeinsames Konzept für `Ball` und `Posn`; daher ist es nicht sinnvoll, eine gemeinsame Strukturdefinition zu verwenden.

Ein anderes sinnvolles Kriterium, um über Wiederverwendung zu entscheiden, ist die Frage, ob es wichtig ist, dass man mit einem Prädikat die unterschiedlichen Daten unterscheiden kann — falls ja, so sollte man jeweils eigene Strukturen verwenden.

6.5 Fallstudie: Ein Ball in Bewegung

Probieren Sie aus, was das folgende Programm macht. Verstehen Sie, wie Strukturen und Datendefinitionen verwendet wurden, um das Programm zu strukturieren!

```
(define WIDTH 200)
(define HEIGHT 200)
(define BALL-IMG (circle 10 "solid" "red"))
(define BALL-RADIUS (/ (image-width BALL-IMG) 2))

(define-struct vel (delta-x delta-y))
; a Vel is a structure: (make-vel Number Number)
; interp. the velocity vector of a moving object

(define-struct ball (loc velocity))
; a Ball is a structure: (make-ball Posn Vel)
; interp. the position and velocity of a object

; Posn Vel -> Posn
```

```
; applies q to p and simulates the movement in one clock tick
(check-expect (posn+vel (make-posn 5 6) (make-vel 1 2))
              (make-posn 6 8))
(define (posn+vel p q)
  (make-posn (+ (posn-x p) (vel-delta-x q))
             (+ (posn-y p) (vel-delta-y q))))


; Ball -> Ball
; computes movement of ball in one clock tick
(check-expect (move-ball (make-ball (make-posn 20 30)
                                    (make-vel 5 10)))
              (make-ball (make-posn 25 40)
                         (make-vel 5 10)))
(define (move-ball ball)
  (make-ball (posn+vel (ball-loc ball)
                       (ball-velocity ball))
             (ball-velocity ball)))

; A Collision is either
; - "top"
; - "down"
; - "left"
; - "right"
; - "none"
; interp. the location where a ball collides with a wall

; Posn -> Collision
; detects with which of the walls (if any) the ball collides
(check-expect (collision (make-posn 0 12))  "left")
(check-expect (collision (make-posn 15 HEIGHT)) "down")
(check-expect (collision (make-posn WIDTH 12))  "right")
(check-expect (collision (make-posn 15 0)) "top")
(check-expect (collision (make-posn 55 55)) "none")
(define (collision posn)
  (cond
    [(<= (posn-x posn) BALL-RADIUS) "left"]
    [(<= (posn-y posn) BALL-RADIUS)  "top"]
    [(>= (posn-x posn) (- WIDTH BALL-RADIUS)) "right"]
    [(>= (posn-y posn) (- HEIGHT BALL-RADIUS)) "down"]
    [else "none"]))

; Vel Collision -> Vel
; computes the velocity of an object after a collision
(check-expect (bounce (make-vel 3 4) "left")
              (make-vel -3 4))
```

```
(check-expect (bounce (make-vel 3 4) "top")
              (make-vel 3 -4))
(check-expect (bounce (make-vel 3 4) "none")
              (make-vel 3 4))
(define (bounce vel collision)
  (cond [(or (string=? collision "left")
             (string=? collision "right"))
         (make-vel (- (vel-delta-x vel))
                   (vel-delta-y vel))]
        [(or (string=? collision "down")
             (string=? collision "top"))
         (make-vel (vel-delta-x vel)
                   (- (vel-delta-y vel)))]
        [else vel]))

; WorldState is a Ball

; WorldState -> Image
; renders ball at its position
(check-expect (image? (render INITIAL-BALL)) #true)
(define (render ball)
  (place-image BALL-IMG
               (posn-x (ball-loc ball))
               (posn-y (ball-loc ball))
               (empty-scene WIDTH HEIGHT)))


; WorldState -> WorldState
; moves ball to its next location
(check-expect (tick (make-ball (make-posn 20 12) (make-
vel 1 2)))
              (make-ball (make-posn 21 14) (make-vel 1 2)))
(define (tick ball)
  (move-ball (make-ball (ball-loc ball)
                        (bounce (ball-velocity ball)
                                (collision (ball-
loc ball))))))

(define INITIAL-BALL (make-ball (make-posn 20 12)
                                (make-vel 1 2)))

(define (main ws)
  (big-bang ws (on-tick tick 0.01) (to-draw render)))

; start with: (main INITIAL-BALL)
```

Probieren Sie aus, was passiert, wenn der Ball genau in eine Ecke des Spielfeldes fliegt. Wie kann man dieses Problem lösen? Reflektieren Sie in Anbetracht dieses Problems darüber, wieso es wichtig ist, immer die Extremfälle (im Englischen: ***Corner Cases***) zu testen ;-)

6.6 Tagged Unions und Maybe

Wie bereits im Abschnitt §5.4 "Unterscheidbarkeit der Alternativen" bemerkt wurde ist es für Summentypen wichtig, dass die einzelnen Alternativen unterscheidbar sind. Nehmen wir zum Beispiel den Datentyp `MaybeNumber` aus Abschnitt §5.3 "Summentypen":

```
; A MaybeNumber is one of:
; - a Number
; - #false
; interp. a number if successful, else false.
```

Das Konzept, dass eine Berechnung fehlschlägt, bzw. dass eine Eingabe optional ist, ist sehr häufig anzutreffen. Hier stellt der obige Summentyp eine gute Lösung dar.

Nun würden wir gerne die gleiche Idee auch für optionale Wahrheitswerte verwenden und definieren daher:

```
; A BadMaybeBoolean is one of:
; - a Boolean
; - #false
; interp. a boolean if successful, else false.
```

Zur Erinnerung: Wir können den Summentyp aber nur so definieren, da der Wert `#false` nicht in `Number` enthalten ist und keine Zahl aus `Number` in der ein-elementigen Menge { `#false` } enthalten ist. Die beiden Mengen sind disjunkt.

Die beiden Varianten sind offensichtlich nicht disjunkt: Die Vereinigung von { `#true`, `#false` } und { `#false` } hätte wieder nur zwei Elemente.

Um das zu verhindern, können wir mit einem *tag* markieren, zu welcher Alternative ein Wert gehört. Hierzu definieren wir zunächst die Strukturen

```
(define-struct some (value))
(define-struct none ())
```

Die erste Struktur nutzen wir, um zu markieren, dass es sich bei der Variante um einen vorhandenen Wert handelt – die zweite Struktur nutzen wir, um zu markieren, dass kein Wert vorhanden ist:

```
; A MaybeBoolean is one of:
; - a structure: (make-some Boolean)
; - (make-none)
; interp. a boolean if successful, else none.
```

Der Summentyp `MaybeBoolean` hat nun drei Elemente { `(make-none)`, `(make-some #true)`, `(make-some #false)` }. Die beiden Alternativen sind nun eindeutig zu unterscheiden. Insbesondere können wir jetzt eine Funktion, die ein `MaybeBoolean` erwartet, einfach nach dem Entwurfsrezept (in der Version für Summentypen) implementieren.

Würden wir `MaybeNumber` nun auch auf diese Weise definieren und danach die Summe aus `MaybeBoolean` und `MaybeNumber` bilden, hätten wir wieder das ursprüngliche Problem: Es ist nicht

6.7 Erweiterung des Entwurfsrezepts

Die vorhergehenden Beispiele haben gezeigt, dass viele Probleme es erfordern, parallel mit Funktionen passende Datenstrukturen zu entwickeln. Dies bedeutet, dass sich die Schritte des Entwurfsrezepts aus Abschnitt §3.3.3 “Entwurfsrezept zur Funktionsdefinition” wie folgt ändern:

1. Wenn in einer Problembeschreibung Information auftauchen, die zusammengehören oder ein Ganzes beschreiben, benötigt man Strukturen. Die Struktur korrespondiert zu dem "Ganzen" und hat für jede "relevante" Eigenschaft ein Feld.

   Eine Datendefinition für ein Feld muss einen Namen für die Menge der Instanzen der Struktur angeben, die durch diese Datendefinition beschrieben werden. Sie muss beschreiben, welche Daten für welches Feld erlaubt sind. Hierzu sollten nur Namen von eingebauten Datentypen oder von Ihnen bereits definierten Daten verwendet werden.

   Geben Sie in der Datendefinition Beispiele für Instanzen der Struktur, die der Datendefinition entsprechen, an.

2. Nichts ändert sich im zweiten Schritt.

3. Verwenden Sie im dritten Schritt die Beispiele aus dem ersten Schritt, um Tests zu entwerfen. Wenn eines der Felder einer Struktur, die Eingabeparameter ist, einen Summentypen hat, so sollten Testfälle für alle Alternativen vorliegen. Bei Intervallen sollten die Endpunkte der Intervalle getestet werden.

4. Eine Funktion die Instanzen von Strukturen als Eingabe erhält wird in vielen Fällen die Felder der Strukturinstanz lesen. Um Sie an diese Möglichkeit zu erinnern, sollte das Template für solche Funktionen die Selektorausdrücke (zum Beispiel `(posn-x param)` falls `param` ein Parameter vom Typ `Posn` ist) zum Auslesen der Felder enthalten.

   Falls der Wert eines Feldes selber Instanz einer Struktur ist, sollten Sie jedoch *nicht* Selektorausdrücke für die Felder dieser verschachtelten Strukturinstanz ins Template aufnehmen. Meistens ist es besser, die Funktionalität, die diese Unterstruktur betrifft, in eine neue Hilfsfunktion auszulagern.

5. Benutzen Sie die Selektorausdrücke aus dem Template um die Funktion zu implementieren. Beachten Sie, dass Sie möglicherweise nicht die Werte aller Felder benötigen.

6. Testen Sie, sobald Sie den Funktionsheader geschrieben haben. Überprüfen Sie, dass zu diesem Zeitpunkt alle Tests fehlschlagen (bis auf die bei denen zufällig der eingesetzte Dummy-Wert richtig ist). Dieser Schritt ist wichtig, denn er bewahrt Sie vor Fehlern in den Tests und stellt sicher, dass ihre Tests auch wirklich eine nicht-triviale Eigenschaft testen.

   Testen Sie so lange, bis alle Ausdrücke im Programm während des Testens mindestens einmal ausgeführt wurden. Die Codefärbung in DrRacket nach dem Testen unterstützt Sie dabei.

7 Datendefinition durch Alternativen und Zerlegung: Algebraische Datentypen

In den letzten beiden Kapiteln haben wir zwei neue Arten von Datendefinitionen kennengelernt: Summentypen (in Form von Aufzählungen und Intervallen), mit denen man zwischen verschiedenen Alternativen auswählen kann. Produkttypen (in Form von Datendefinitionen für Strukturen), mit denen man Daten zerlegen kann.

In vielen Fällen ist es sinnvoll, Summen und Produkte miteinander zu kombinieren, also zum Beispiel Summentypen zu definieren, bei denen einige Alternativen einen Produkttyp haben; oder Produkttypen bei denen einige Felder Summentypen haben.

Kombinationen von Summen- und Produkttypen nennen wir *algebraische Datentypen*, kurz *ADT*.

7.1 Beispiel: Kollisionen zwischen Shapes

Nehmen Sie an, sie möchten ein Computerspiel programmieren, in dem Spielfiguren in unterschiedlichen geometrischen Formen vorkommen, zum Beispiel Kreise und Rechtecke. Diese können wir beispielsweise so modellieren:

Wir wählen den Namen `gcircle` (für *geometric* circle) um keinen Namenskonflikt mit der `circle` Funktion aus dem image.ss Teachpack zu haben.

```
(define-struct gcircle (center radius))
; A GCircle is (make-gcircle Posn Number)
; interp. the geometrical representation of a circle

(define-struct grectangle (corner-ul corner-dr))
; A GRrectangle is (make-grectangle Posn Posn)
; interp. the geometrical representation of a rectangle
; where corner-ul is the upper left corner
; and corner-dr the down right corner
```

Das interessante an diesen beiden Definitionen ist, dass Kreise und Rechtecke viele Gemeinsamkeiten haben. Beispielsweise können beide verschoben, vergrößert oder gezeichnet werden. Viele Algorithmen sind für beliebige geometrische Formen sinnvoll und unterscheiden sich nur in den Details. Nennen wir die Abstraktion, die beliebige geometrische Figuren bezeichnet, beispielsweise `Shape`. Nehmen wir an, wir haben bereits eine Funktion, die für beliebige geometrische Formen bestimmen kann, ob sie überlappen:

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2) ...)
```

Auf Basis dieser und ähnlicher Funktionen können wir weitere Funktionen programmieren, die völlig unabhängig vom konkreten Shape (ob Kreis oder Rechteck) sind, beispielsweise eine Funktion, die für drei Shapes testet ob sie paarweise überlappen:

```
; Shape Shape Shape -> Boolean
; determines whether the shapes overlap pairwise
(define (overlaps/3 shape1 shape2 shape3)
  (and
   (overlaps shape1 shape2)
   (overlaps shape1 shape3)
   (overlaps shape2 shape3)))
```

Die Funktion `overlaps` muss zwar unterscheiden, was für eine genaue Form ihre Parameter haben, aber die Funktion `overlaps/3` nicht mehr. Dies ist eine sehr mächtige Form der Abstraktion und Code-Wiederverwendung: Wir formulieren abstrakte Algorithmen auf Basis einer Menge einfacher Funktionen wie `overlaps` und können dann diese Algorithmen auf *alle* Arten von Shapes anwenden.

Die Funktion `overlaps/3` steht also wirklich für eine ganze Familie von Algorithmen: Je nachdem, was für Shapes ich gerade als Argument verwende, machen die Hilfsfunktionen wie `overlaps` etwas anderes, aber der abstrakte Algorithmus in `overlaps/3` ist immer gleich.

Schauen wir uns nun an, wie wir Datentypen wie `Shape` und Funktionen wie `overlaps` definieren können. Um die Menge der möglichen geometrischen Figuren zu beschreiben, verwenden wir einen Summentyp, dessen Alternativen die Produkttypen von oben sind:

```
; A Shape is either:
; - a GCircle
; - a GRectangle
; interp. a geometrical shape representing a circle or a rect-
angle
```

Nicht alle Algorithmen sind wie `overlaps/3` — sie sind unterschiedlich je nachdem welches Shape gerade vorliegt. Wie bereits aus dem Entwurfsrezept für Summentypen bekannt, strukturieren wir solche Funktionen, indem wir eine Fallunterscheidung machen.

Beispiel:

```
; Shape Posn -> Boolean
; Determines whether a point is inside a shape
(define (point-inside shape point)
  (cond [(gcircle? shape) (point-inside-circle shape point)]
        [(grectangle? shape) (point-inside-
rectangle shape point)]))
```

In dieser Funktion fällt auf, dass in den Zweigen des `cond` Ausdrucks Hilfsfunktionen aufgerufen werden, die wir noch implementieren müssen. Man könnte stattdessen die Implementation dieser Funktionen auch direkt in den `cond` Ausdruck schreiben, aber wenn die Alternativen eines Summentyps Produkte sind, so werden diese Ausdrücke häufig so komplex, dass man sie besser in eigene Funktionen auslagern sollte. Außerdem ergeben sich durch die Auslagerung in separate Funktionen häufig Gelegenheiten zur Wiederverwendung dieser Funktionenen.

In unserem Beispiel sehen die Implementationen dieser Hilfsfunktionen wie folgt aus:

Ergänzen Sie die Definitionen für `vector-length` und `posn-`!

```
; GCircle Posn -> Boolean
; Determines whether a point is inside a circle
(define (point-inside-circle circle point)
  (<= (vector-length (posn- (gcircle-center circle) point))
      (gcircle-radius circle)))

; GRectangle Posn -> Boolean
; Determines whether a point is inside a rectangle
(define (point-inside-rectangle rectangle point)
  (and
   (<= (posn-x (grectangle-corner-ul rectangle))
       (posn-x point)
       (posn-x (grectangle-corner-dr rectangle)))
   (<= (posn-y (grectangle-corner-ul rectangle))
       (posn-y point)
       (posn-y (grectangle-corner-dr rectangle)))))
```

Bei manchen Funktionen haben gleich mehrere Parameter einen Summentypen. In diesem Fall kann man erst für den ersten Parameter die Fallunterscheidung machen, dann geschachtelt für den zweiten Parameter usw. Häufig kann man jedoch auch die Fallunterscheidungen zusammenfassen. Dies ist beispielsweise bei dem oben angesprochenen `overlaps` der Fall:

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2)
  (cond [(and (gcircle? shape1) (gcircle? shape2))
         (overlaps-circle-circle shape1 shape2)]
        [(and (grectangle? shape1) (grectangle? shape2))
         (overlaps-rectangle-rectangle shape1 shape2)]
        [(and (grectangle? shape1) (gcircle? shape2))
         (overlaps-rectangle-circle shape1 shape2)]
        [(and (gcircle? shape1) (grectangle? shape2))
         (overlaps-rectangle-circle shape2 shape1)]))
```

Auch hier haben wir wieder die Implementierung der einzelnen Fälle in Funktionen ausgelagert. Die Tatsache, dass wir in den beiden letzten Fällen die gleiche Funktion aufrufen, illustriert, dass diese Hilfsfunktionen die Wiederverwendung von Code fördern. Hier die Implementation der Hilfsfunktionen.

Wenn Sie Spass an Geometrie haben, ergänzen Sie die Implementation von `overlaps-rectangle-rectangle` und `overlaps-rectangle-circle`.

```
; GCircle GCircle -> Boolean
; determines whether c1 overlaps with c2
(define (overlaps-circle-circle c1 c2)
  ; Two circles overlap if and only if the distance of their
```

```
  ;  centers is smaller than the sum of their radii
  (<= (vector-length (posn- (gcircle-center c1) (gcircle-
center c2)))
      (+ (gcircle-radius c1) (gcircle-radius c2))))

; GRectangle GRectangle -> Boolean
; determines whether r1 overlaps with r2
(define (overlaps-rectangle-rectangle r1 r2) ...)

; GRectangle GCircle -> Boolean
; determines whether r overlaps with c
(define (overlaps-rectangle-circle r c) ...)
```

7.2 Programmentwurf mit ADTs

Das Entwurfsrezept aus Abschnitt §3.3.3 “Entwurfsrezept zur Funktionsdefinition” ergänzen wir wie folgt:

1. Wenn Sie eine Funktion programmieren möchten, die Informationen als Eingabe erhält oder als Ausgabe produziert, die man am besten durch ADTs repräsentiert, so sollten Sie vor dem Programmieren dieser Funktion diesen ADT definieren (falls sie ihn nicht schon im Kontext einer anderen Funktion definiert haben).

   Ein ADT ist dann sinnvoll, wenn in ihrer Problemstellung unterschiedliche Arten von Informationen unterschieden werden, die jeweils als Produkttyp formuliert werden können, die aber ein gemeinsames Konzept repräsentieren.

   Ein wichtiger Punkt ist, dass man Daten mit ADTs hierarchisch organisieren kann. Die Feldtypen der Produkte, die man definiert, können also selber wieder einen ADT haben. Wenn also ein Produkttyp sehr viele Felder hat, oder ein Summentyp sehr viele Alternativen, so deutet dies darauf hin, dass Sie eine tiefere Hierarchie (durch Verschachtelung von ADTs) verwendensollten.

2. Im zweiten Schritt, der Definition der Funktionssignatur und der Aufgabenbeschreibung, ändert sich nichts — allerdings können und sollten Sie nun natürlich die Namen der definierten ADTs verwenden.

3. Bei der Definition der Testcases sollten Sie bei Summentypen mindestens einen Test pro Alternative definieren. Beachten Sie, dass es durch die Schachtelung von Summen- und Produkttypen jetzt möglicherweise viel mehr Alternativen gibt. Bei sehr großen Datentypen ist es unter Umständen nicht mehr realistisch, jede Alternative zu testen, weil im schlechtesten Fall die Menge der Alternativen exponentiell mit der Tiefe des Datentypen wächst.

4. Bei der Definition des Funktionstemplates gibt es nun zwei Dimensionen: Den Summentyp und die Produkttypen in (einigen seiner) Alternativen.

   Falls wir als äußerstes einen Summentyp haben, so sollten wir zunächst in einem `cond` Ausdruck alle Fälle unterscheiden.

Für alle Alternativen, die Produkttypen sind, sollte in der Regel eine Hilfsfunktion definiert werden, die diesen Fall abdeckt. Diese Hilfsfunktion sollten sie in diesem Schritt auch nur (durch erneute Anwendung dieses Entwurfsrezepts) bis zum Template-Schritt implementieren.

Nur wenn die Implementation dieses Falls wahrscheinlich sehr einfach ist, sollten die Selektoren für die Felder des Produkts in das Template mit aufgenommen werden.

Allerdings gibt es einen wichtigen Fall, in dem sie *keinen* `cond` Ausdruck zur Unterscheidung der Alternativen ins Template aufnehme sollte, nämlich dann, wenn es möglich ist, die Funktion abstrakt zu formulieren — sie also die Fälle nicht unterscheiden, sondern lediglich bereits existierende Funktionen aufrufen, die auch auf Basis des ADTs implementiert wurden. Als Beispiel haben wir die `overlap/3` Funktion gesehen.

5. In diesem Schritt sollten sie aus dem Template ein lauffähiges Programm machen. Da sie im vorherigen Schritt eventuell Templates für Hilfsfunktionen definiert haben, müssen sie auch diese Hilfsfunktionen nun implementieren.

6. Testen sie. Falls Tests fehlschlagen, gehen sie zurück zum vorherigen Schritt.

7. Überprüfen Sie beim Refactoring zusätzlich die neu definierten ADTs. Auch hier kann es Verstöße gegen das DRY-Prinzip geben, z.B. wenn es große Gemeinsamkeiten zwischen ADTs gibt. Gibt es beispielsweise in mehreren Datentypen zwei Felder zur Repräsentation von Koordinaten, so bietet es sich an, stattdessen die `posn` Struktur zu verwenden. Vermeiden Sie sehr breite aber flache ADTs; die logische Gruppierung der Daten durch eine Hierarchie von ADTs fördert die Wiederverwendbarkeit und Lesbarkeit des Codes.

7.3 Refactoring von algebraischen Datentypen

Algebraische Datentypen sind Kombinationen von Produkttypen und Summentypen. Wie bereits in Schritt 7 des Entwurfsrezepts oben angedeutet, kann es mehrere Arten geben Daten zu modellieren. Manche Lösungen für eine Datenmodellierung zeigen dabei mehr Wiederholungen, als andere und verstoßen damit gegen das DRY-Prinzip.

Wenn wir ein Programm umschreiben können, dass es statt eines Datentyps *A* nun einen Datentyp *B* verwendet, und es eine bijektive Abbildung zwischen den Daten, die von *A* und *B* beschrieben werden gibt, sagen wir auch *A* und *B* sind *isomorph*.

Beispiel: Betrachten Sie die folgenden drei Definitionen für einen Datentyp *Student*:

```
(define-struct student1 (lastname firstname matnr))
; a Student1 is: (make-student1 String String Number)
; interp. lastname, firstname, and matrikel number of a stu-
dent

(define-struct student2 (matnr lastname firstname))
```

```
; a Student2 is: (make-student2 Number String String)
; interp. matrikel number, lastname, and firstname of a stu-
dent

(define-struct fullname (firstname lastname))
; a FullName is: (make-fullname String String)
; interp. first name and last name of a person

(define-struct student3 (fullname matnr))
; a Student3 is: (make-student3 FullName Number)
; interp. full name and matrikel number of a stu-
dent
```

Jede der drei Repräsentationen kann die gleichen Informationen darstellen. Programme, die einen dieser Typen verwenden können so refactored werden, dass sie einen der beiden anderen verwenden. Was wir an *Student1* und *Student2* sehen, ist, dass Produkttypen, die sich nur in der Reihenfolge der Komponenten unterscheiden, isomorph sind: Wir könnten zwischen den Datentypen hin- und herkonvertieren,

```
; Student1 -> Student2
(define (student1->student2 s)
  (make-student2 (student1-matnr s) (student1-
lastname s) (student1-firstname s)))

; Student2 -> Student1
(define (student2->student1 s)
  (make-student1 (student1-lastname s) (student1-
firstname s) (student1-matnr s)))
```

und wir können Programme, die *Student1* verwenden, so refactoren, dass sie stattdessen *Student2* verwenden, nämlich indem die entsprechenden Konstruktoren und Selektoren angepasst werden.

Das dritte Beispiel, *Student3*, zeigt, dass wir Daten gruppieren und in separate Datentypen auslagern können. Auch hier gibt es eine Bijektion und ein offensichtliches Refactoring des Programms.

Wenn wir im Allgemeinen Fall solche Typisomorphien betrachten, spielen offensichtlich die verwendeten Namen für die Struktur und die Komponenten keine Rolle. Wir könnten für diesen Zweck Produkttypen so schreiben:

```
(* String String Number)
```

statt

```
; A Student1 is: (make-student1 String String Number)
; interp. lastname, firstname, and matrikel number of a stu-
dent
```

Analog dazu können wir Summentypen mit der + Notation beschreiben. Statt

```
; A UniversityPerson is either:
; - a Student
; - a Professor
; - a ResearchAssociate
; interp. the kind of person you can meet at university.
```

schreiben wir `(+ Student Professor ResearchAssociate)`.

Achtung: Wir verwenden diese Schreibweise lediglich, um über Isomorphien und Refactorings nachzudenken. In Ihrem Programmcode sollten Sie weiterhin bei der üblichen Schreibweise bleiben!

In dieser Notation können wir die Typisomorphien auf algebraischen Datentypen wie folgt ausdrücken. Wenn `X`, `Y` und `Z` beliebige Typen sind, dann gelten folgende Isomorphien:

```
; Assoziativität von *
(* X (* Y Z)) = (* (* X Y) Z) = (* X Y Z)

; Kommutativität von *
(* X Y) = (* Y X)

; Assoziativität von +
(+ X (+ Y Z)) = (+ (+ X Y) Z) = (+ X Y Z)

; Kommutativität von +
(+ X Y) = (+ Y X)

; Distributivität von * und +
(* X (+ Y Z)) = (+ (* X Y) (* X Z))
```

Es ist kein Zufall, dass wir mit Typen rechnen als wären es Ausdrücke in der Algebra (daher kommt übrigens der Name "algebraische Datentypen"). Wenn man Summentypen als sogenannte "tagged unions" definiert (also die Alternativen immer durch Tags unterschieden werden können), so geht diese Analogie sogar noch viel weiter. Wenn man beispielsweise die Datentypen, die nur ein einiges Element haben, als `1` bezeichnet und Datentypen mit zwei Elementen (wie Boolean) als `2`, gelten auch Isomorphien wie

```
(+ 1 1) = 2
```

oder

```
(+ X X) = (* 2 X)
```

Die oben stehenden Isomorphien rechtfertigen eine große Klasse von Refactorings von algebraischen Datentypen: Vertauschung von Reihenfolgen, "Inlining" bzw. "Outsourcing" von Datentypen, "Ausmultiplikation" von Produkten.

8 Bedeutung von BSL

In diesem Kapitel werden wir die Bedeutung (fast) aller Sprachkonstrukte von BSL zusammenfassen und formal definieren.

Dies geschieht in zwei Schritten: Wir werden zunächst die *Syntax* der Sprache definieren. Die Syntax definiert, welche Texte BSL-Programme sind. Die Syntax wird in Form einer *Grammatik* definiert. Die Grammatik sagt nicht nur, welche Texte BSL-Programme sind, sondern zerlegt ein BSL-Programm in seine Teile, genau so wie eine Grammatik für natürliche Sprache einen Satz in Teile wie Subjekt, Prädikat und Objekt zerlegt.

Im zweiten Schritt definieren wir für die grammatikalisch korrekten BSL-Programme, was diese bedeuten. Die Bedeutung legen wir durch die Definition von Reduktionsschritten fest, mit denen BSL Programme zu Werten ausgewertet werden können (sofern kein Fehler auftritt und sie terminieren).

Wir haben bereits in den Abschnitten §1.5 “Bedeutung von BSL Ausdrücken”, §2.3 “Bedeutung von Funktionsdefinitionen”, §2.4.2 “Bedeutung konditionaler Ausdrücke” und §2.7 “Bedeutung von Funktions- und Konstantendefinitionen” diese Reduktionsschritte für die meisten Sprachkonstrukte definiert. Wir werden hier diese Reduktionsschritte anhand der formalen Syntaxdefinition nochmal präzisieren. Außerdem werden wir nun auch definieren, welche Bedeutung Strukturen haben.

Es gibt verschiedene Möglichkeiten, die Bedeutung eine Programmiersprache zu definieren; die, die wir benutzen, nennt man *Reduktionssemantik* oder *strukturelle operationelle Semantik* oder *Plotkin Semantik* (nach Gordon Plotkin). Für die Formalisierung der Auswertungspositionen, von denen wir in den vorherigen Kapiteln gesprochen haben, verwenden wir sogenannte *Auswertungskontexte*, die 1989 von Matthias Felleisen und Robert Hieb vorgeschlagen wurde. Das alles hört sich für einen Programmieranfänger vielleicht angsteinflößend an, aber Sie werden sehen, dass es nicht so kompliziert ist wie es sich anhört :-)

8.1 Wieso?

Die meisten Programmierer dieser Welt programmieren, ohne dass sie jeweils eine formale Definition der Bedeutung ihrer Programmiersprache gesehen haben. Insofern ist die Frage berechtigt, wieso wir uns dies "antun".

Hierzu ist zu sagen, dass viele Programmierer die Programmiersprache, die sie verwenden, nicht wirklich verstehen. Dies führt zu einer Methodik, in der statt systematischem Programmentwurf einfach so lange am Programm "herumgedoktort" wird, bis es "läuft". Ein Programm durch Ausführen und Tests zu validieren ist zwar sinnvoll, aber dennoch kann dies nicht den gedanklichen Prozess ersetzen, wie ein Programm ablaufen muss, damit zu jeder Eingabe die korrekte Ausgabe produziert wird. Dazu ist es unumgänglich, dass Sie genau verstehen, was der Code bedeutet, den Sie gerade programmiert haben.

Wir möchten, dass Sie prinzipiell in der Lage sind, ihre Programme auf einem Blatt Papier auszuführen und exakt vorherzusagen, was ihr Code bewirkt. Auch wenn Ihnen der Umgang mit den eher theoretischen Konzepten dieses Kapitels vielleicht am

Anfang schwerstellt, glauben wir, dass Ihnen dieses Kapitel helfen kann, ein besserer und effektiverer Programmierer zu werden.

Davon abgesehen werden Sie sehen, dass die theoretischen Konzepte, die sie in diesem Kapitel kennenlernen, eine Eleganz haben, die es allein aus diesem Grund wert macht, sie zu studieren.

8.2 Kontextfreie Grammatiken

Bisher haben wir nur informell beschrieben, wie BSL Programme aussehen. Mit Hilfe einer *Grammatik* kann man diese informelle Beschreibung präzise und prägnant darstellen. Es gibt viele unterschiedliche Arten von Grammatiken. Im Bereich der Programmiersprachen verwendet man meistens sogenannte *kontextfreie* Grammatiken. Diese und andere Grammatikformalismen werden in der Vorlesung "Theoretische Informatik" im Detail behandelt; wir werden Grammatiken hier nur soweit besprechen, wie es zum Verständnis der Definitionen erforderlich ist.

Es gibt unterschiedliche Notationen für kontextfreie Grammatiken. Wir verwenden die sogenannte EBNF — die Erweiterte Backus Naur Form.

Hier direkt ein Beispiel einer Grammatik für Zahlen:

⟨*Zahl*⟩ ::= ⟨*PositiveZahl*⟩
  | `-`⟨*PositiveZahl*⟩
⟨*PositiveZahl*⟩ ::= ⟨*GanzeZahl*⟩
  | ⟨*KommaZahl*⟩
⟨*GanzeZahl*⟩ ::= ⟨*ZifferNichtNull*⟩ ⟨*Ziffer*⟩*
  | `0`
⟨*Kommazahl*⟩ ::= ⟨*GanzeZahl*⟩ `.` ⟨*Ziffer*⟩+
⟨*ZifferNichtNull*⟩ ::= `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
⟨*Ziffer*⟩ ::= `0` | ⟨*ZifferNichtNull*⟩

Beispiele für Texte, die der ⟨*Zahl*⟩ Definition dieser Grammatik entsprechen, sind: `0`, `420`, `-87`, `3.1416`, `-2.09900`.

Beispiele für Texte, die nicht der ⟨*Zahl*⟩ Definition dieser Grammatik entsprechen, sind: `007`, `-.65`, `13.`, `zwölf`, `111Nonsense222`.

Die mit spitzen Klammern markierten Bezeichner wie ⟨*Zahl*⟩ heißen *Nichtterminale*; die farblich markierten Symbole wie `3` oder `.` heißen *Terminalsymbole*. Eine Klausel wie die ersten beiden Zeilen der obigen Grammatik heißt *Produktion*. Eine Produktion besteht aus einem Nichtterminal auf der linken Seite der Definition und auf der rechten Seite aus einer Menge von Alternativen, die durch das Symbol | voneinander getrennt werden. Zu jedem Nichtterminal gibt es genau eine Produktion.

Zu jedem Nichtterminal kann man eine Menge von *Ableitungsbäumen* bilden. Ein Ableitungsbaum entsteht durch das Ersetzen der Nichtterminale in einer der Alternativen der dazugehörigen Produktion durch Ableitungsbäume für diese Nichtterminale. Die Konstruktion der Ableitungsbäume ist also ein rekursiver Prozess. Der Prozess stoppt dort, wo man eine Alternative wählt, die nur aus Terminalsymbolen bestehen. Falls ein Nichtterminal durch ein Sternchen oder ein Pluszeichen markiert wird, so wie ⟨*Ziffer*⟩* oder ⟨*Ziffer*⟩+ oben, so bedeutet dies 0 oder mehr Wiederholungen (für *) bzw. 1 oder mehr Wiederholungen (für +) des Nichtterminals.

Jeder Ableitungsbaum steht für einen Text (häufig *Wort* oder *Satz* genannt), nämlich die Sequenz der Terminalsymbole, die in dem Baum vorkommen, von links nach rechts im Baum abgelesen. Die durch eine Grammatik definierte Sprache ist die Menge aller Worte, für die man Ableitungsbäume bilden kann.

Hier einige Beispiele für Ableitungsbäume[2] des Nichtterminals ⟨*Zahl*⟩ und die Worte, die sie repräsentieren. Wir stellen die Bäume der Einfachheit halber durch Einrückung des Textes dar. Da damit die Bäume um 90 Grad gegenüber der Standarddarstellung gedreht sind, müssen die Terminalsymbole von oben nach unten (statt von links nach rechts) abgelesen werden.

Der Ableitungsbaum für `0` ist:

⟨*Zahl*⟩
⟨*PositiveZahl*⟩
⟨*GanzeZahl*⟩
`0`

Der Ableitungsbaum für `420` ist:

⟨*Zahl*⟩
⟨*PositiveZahl*⟩
⟨*GanzeZahl*⟩
⟨*ZifferNichtNull*⟩
`4`
⟨*Ziffer*⟩
⟨*ZifferNichtNull*⟩
`2`
⟨*Ziffer*⟩
`0`

8.3 Syntax von BSL

Nach diesen Vorarbeiten können wir nun präzise die Syntax von BSL durch eine kontextfreie Grammatik definieren. Diese Syntax ist vollständig bis auf folgende Sprachfeatures, die wir noch nicht behandelt haben: Definition von Funktionen durch lambda-Ausdrücke, Quoting, Zeichendaten, Bibliotheksimporte. Außerdem werden in der Grammatik Aspekte, die für die Bedeutung der Sprache irrelevant sind, außer Acht gelassen, zum Beispiel Kommentare, Zeilenumbrüche und Leerzeichen. Aus diesem Grund bezeichnet man Grammatiken wie die folgende für BSL häufig als die *abstrakte Syntax* einer Programmiersprache, im Unterschied zur *konkreten Syntax*, die auch Aspekte wie Kommentare und Zeilenumbrüche umfasst. Analog dazu werden Ableitungsbäume, wie sie oben beschrieben wurden, im Kontext abstrakter Syntax

[2]Wenn Sie mit Grammatiken und Ableitungsbäumen experimentieren möchten, schauen Sie sich mal das Grammatik-Werkzeug unter `http://www.cburch.com/proj/grammar/index.html` an. Mit diesem Werkzeug können Sie automatisch Ableitungsbäume für Wörter kontextfreier Grammatiken eingeben. Die Notation für kontextfreie Grammatiken in dem Tool ist allerdings etwas anders und sie u nterstützt nicht die + und * Operatoren sowie nur alphanumerische Nichtterminale. Eine Variante der Grammatik oben, die dieses Tool versteht, finden Sie unter der URL `https://github.com/ps-mr/KdP2014/blob/master/materials/grammar`.

häufig als *abstrakte Syntaxbäume* (*abstract syntax trees (AST)* bezeichnet.

⟨*program*⟩ ::= ⟨*def-or-expr*⟩*
⟨*def-or-expr*⟩ ::= ⟨*definition*⟩ | ⟨*e*⟩
⟨*definition*⟩ ::= `(` `define` `(` ⟨*name*⟩ ⟨*name*⟩+ `)` ⟨*e*⟩ `)`
| `(` `define` ⟨*name*⟩ ⟨*e*⟩ `)`
| `(` `define-struct` ⟨*name*⟩ `(` ⟨*name*⟩* `)` `)`
⟨*e*⟩ ::= `(` ⟨*name*⟩ ⟨*e*⟩* `)`
| `(` `cond` {`[` ⟨*e*⟩ ⟨*e*⟩ `]`}+ `)`
| `(` `cond` {`[` ⟨*e*⟩ ⟨*e*⟩ `]`}* `[` `else` ⟨*e*⟩ `]` `)`
| `(` `if` ⟨*e*⟩ ⟨*e*⟩ ⟨*e*⟩ `)`
| `(` `and` ⟨*e*⟩ ⟨*e*⟩+ `)`
| `(` `or` ⟨*e*⟩ ⟨*e*⟩+ `)`
| ⟨*name*⟩
| ⟨*v*⟩
⟨*v*⟩ ::= `<` `make-`⟨*name*⟩ ⟨*v*⟩* `>`
| ⟨*number*⟩
| ⟨*boolean*⟩
| ⟨*string*⟩
| ⟨*image*⟩

Das Nichtterminal ⟨*program*⟩ steht für die Syntax ganzer Programme; ⟨*def-or-expr*⟩ für Definitionen oder Ausdrücke, ⟨*definition*⟩ für Funktions-/Konstanten-/Strukturdefinitionen, ⟨*e*⟩ für Ausdrücke und ⟨*v*⟩ für Werte.

Die geschweiften Klammern um Teilsequenzen wie in {`[` ⟨*e*⟩ ⟨*e*⟩ `]`}+ dienen dazu, um den * oder + Operator auf eine ganze Sequenz von Terminalsymbolen und Nichtterminalen anzuwenden und nicht nur auf ein einzelens Nichtterminal. In diesem Beispiel bedeutet es, dass 1 oder mehr Vorkommen von `[` ⟨*e*⟩ ⟨*e*⟩ `]` erwartet werden.

Die Produktionen für einige Nichtterminale, deren genaue Form nicht interessant ist, wurden in der Grammatik ausgelassen: ⟨*name*⟩ steht für die zugelassenen Bezeichner für Funktionen, Strukturen und Konstanten. ⟨*number*⟩ steht für die zugelassenen Zahlen. ⟨*boolean*⟩ steht für `#true` oder `#false`. ⟨*string*⟩ steht für alle Strings wie `"asdf"`. Das Nichtterminal ⟨*image*⟩ steht für Bilder im Programmtext (Bildliterale) wie  .

Die Werte der Form `<` `make-`⟨*name*⟩ ⟨*v*⟩* `>` dienen dazu, Instanzen von Strukturen zu repräsentieren. Sie dürfen in BSL nicht direkt im Original-Programmtext vorkommen, aber sie werden während der Reduktion erzeugt und in das Programm eingefügt.

8.4 Die BSL Kernsprache

Wenn man die Bedeutung einer Sprache definiert, möchte man normalerweise, dass diese Definition so kurz wie möglich ist, denn nur dann kann ein Benutzer sie leicht verstehen und Schlussfolgerungen ziehen.

Aus diesem Grund identifizieren wir eine Untersprache der BSL, die bereits ausreichend ist, um alle Programme zu schreiben, die man auch in BSL schreiben kann. Der

einzige Unterschied ist, dass man an einigen Stellen vielleicht etwas umständlicheres schreiben muss.

Wir haben bereits ein intellektuelles Werkzeug kennengelernt, um Kernsprachenelemente von eher unwichtigem Beiwerk zu unterscheiden, nämlich den syntaktischen Zucker. Im Abschnitt §2.4.2 “Bedeutung konditionaler Ausdrücke” haben wir gesehen, dass es nicht notwendig ist, `if` Ausdrücke und innerhalb von `cond` Ausdrücken den `else` Operator zu unterstützen, weil man diese Sprachfeatures leicht mit dem einfachen `cond` Ausdruck simulieren kann. Die in §2.4.2 “Bedeutung konditionaler Ausdrücke” angegebenen Transformationen betrachten wir daher als die *Definition* dieser Sprachfeatures und betrachten daher im folgenden nur noch die Kernsprache, in die diese Transformationen abbilden.

Recherchieren Sie, was die *de Morgan'schen Regeln* sind, falls ihnen die Transformation nicht klar ist.

Die Syntax oben enthält auch spezielle Syntax für die logischen Funktionen `and` und `or`, weil deren Argumente anders ausgewertet werden als die Argumente normaler Funktionen. Allerdings ist es in unserer Kernsprache nicht nötig, beide Funktionen zu betrachten, da `or` mit Hilfe von `and` und `not` ausgedrückt werden kann. Wir "entzuckern" `(or e-1 ... e-n)` also zu `(not (and (not e-1) ... (not e-n)))`.

Man könnte versuchen, auch `and` noch wegzutransformieren und durch einen `cond` Ausdruck zu ersetzen: `(and e-1 e-2)` wird transformiert zu `(cond [e-1 e-2] [else #false])`. Zwar simuliert dies korrekt die Auswertungsreihenfolge, aber diese Transformation ist nicht adäquat für das in DrRacket implementierte Verhalten, wie folgendes Beispiel illustriert:

```
> (and #true 42)
and: question result is not true or false: 42

> (cond [#true 42] [else #false])
42
```

Damit sieht die Grammatik unserer Kernsprache wie folgt aus. Die Grammatik für Werte $\langle v \rangle$ bleibt unverändert.

$\langle program \rangle$ ::= $\langle def\text{-}or\text{-}expr \rangle^*$

$\langle def\text{-}or\text{-}expr \rangle$ ::= $\langle definition \rangle$ | $\langle e \rangle$

$\langle definition \rangle$ ::= `(` `define` `(` $\langle name \rangle$ $\langle name \rangle^+$ `)` $\langle e \rangle$ `)`
| `(` `define` $\langle name \rangle$ $\langle e \rangle$ `)`
| `(` `define-struct` $\langle name \rangle$ `(` $\langle name \rangle^*$ `)` `)`

$\langle e \rangle$ ::= `(` $\langle name \rangle$ $\langle e \rangle^*$ `)`
| `(` `cond` {`[` $\langle e \rangle$ $\langle e \rangle$ `]`}$^+$ `)`
| `(` `and` $\langle e \rangle$ $\langle e \rangle^+$ `)`
| $\langle name \rangle$
| $\langle v \rangle$

8.5 Werte und Umgebungen

Was bedeuten nun Programme in der Sprache, deren Syntax oben definiert wurde? Die Bedeutung eines Ausdrucks wollen wir modellieren als Sequenz von Reduktionsschritten, die am Ende zu einem Wert führt (oder mit einem Fehler abbricht oder nicht terminiert).

Werte haben wir bereits oben durch die Grammatik definiert. Alle Konstanten wie `#true`, `42` oder `"xyz"` sind also Werte. Außerdem sind Instanzen von Strukturen Werte; die Werte aller Felder der Struktur müssen ebenfalls Werte sein. Also ist beispielsweise `<make-posn 3 4>` ein Wert. Wir modellieren Strukturen so, dass Ausdrücke wie `(make-posn 3 (+ 2 2))` zu diesem Wert ausgewertet werden — hier ist also der Ausdruck der `make-posn` aufruft (mit runden Klammern) von dem Wert `<make-posn 3 4>` (mit spitzen Klammern) zu unterscheiden.

Sofern in Ausdrücken Funktionen, Konstanten, oder Strukturen benutzt werden, kann die Auswertung eines Ausdrucks nicht im "luftleeren" Raum stattfinden, sondern man muss die *Umgebung* (*Environment*, im folgenden als *env* abgekürzt) kennen, in dem der Ausdruck ausgewertet wird, um Zugriff auf die dazugehörigen Definitionen zu haben. Vom Prinzip her ist die Umgebung einfach der Teil des Programms bis zu dem Ausdruck, der gerade ausgewertet wird. Allerdings werden Konstantenfinitionen ja auch ausgewertet (siehe §2.7 “Bedeutung von Funktions- und Konstantendefinitionen”). Dies bringen wir durch folgende Definition zum Ausdruck. Beachten Sie, dass im Unterschied zur Grammatik von BSL Konstantendefinitionen die Form `(` `define` ⟨*name*⟩ ⟨*v*⟩ `)` und nicht `(` `define` ⟨*name*⟩ ⟨*e*⟩ `)` haben.

⟨*env*⟩ ::= ⟨*env-element*⟩*
⟨*env-element*⟩ ::= `(` `define` `(` ⟨*name*⟩ ⟨*name*⟩+ `)` ⟨*e*⟩ `)`
| `(` `define` ⟨*name*⟩ ⟨*v*⟩ `)`
| `(` `define-struct` ⟨*name*⟩ `(` ⟨*name*⟩* `)` `)`

Ein Umgebung besteht also aus einer Sequenz von Funktions-, Konstanten- oder Strukturdefinitionen, wobei der Ausdruck in Konstantendefinitionen bereits zu einem Wert ausgewertet wurde.

8.6 Auswertungspositionen und die Kongruenzregel

Im Abschnitt §1.5 “Bedeutung von BSL Ausdrücken” haben wir das erste Mal über Auswertungspositionen und die Kongruenzregel gesprochen. Die Auswertungspositionen markieren, welcher Teil eines Programms als nächstes ausgewertet werden soll. Die Kongruenzregel sagt, dass man Unterausdrücke in Auswertungspositionen auswerten und das Ergebnis der Auswertung wieder in den ganzen Ausdruck einbauen darf.

Betrachten wir als Beispiel den Ausdruck `(* (+ 1 2) (+ 3 4))`. Der Unterausdruck `(+ 1 2)` befindet sich in Auswertungsposition und kann zu `3` ausgewertet werden. Gemäß der Kongruenzregel kann ich den Gesamtausdruck also zu `(* 3 (+ 3 4))` reduzieren.

Wir werden Auswertungspositionen und die Kongruenzregel durch einen *Auswertungskontext* formalisieren. Ein Auswertungskontext ist eine Grammatik für Programme, die ein "Loch", `[]`, enthalten. In Bezug auf den DrRacket Stepper kann man den Auswertungskontext als den während einer Reduktion nicht farblich markierten Teil des Ausdrucks verstehen. Die Grammatik ist so strukturiert, dass jedes Element der definierten Sprache genau ein "Loch" enthält.

⟨*E*⟩ ::= `[]`
| `(` ⟨*name*⟩ ⟨*v*⟩* ⟨*E*⟩ ⟨*e*⟩* `)`
| `(` `cond` `[` ⟨*E*⟩ ⟨*e*⟩ `]` {`[` ⟨*e*⟩ ⟨*e*⟩ `]`}* `)`

| `(` `and` ⟨*E*⟩ ⟨*e*⟩$^+$ `)`
| `(` `and` `#true` ⟨*E*⟩ `)`

Hier einige Beispiele für Auswertungskontexte:

```
(* [] (+ 3 4))
```

```
(posn-x (make-posn 14 []))
```

```
(and #true (and [] x))
```

Dies sind alles *keine* Auswertungskontexte:

```
(* (+ 3 4) [])
```

```
(posn-x (make-posn 14 17))
```

```
(and #true (and x []))
```

Das "Loch" in diesen Ausdrücken steht genau für die Unterausdrücke in einem Programm, die in Auswertungsposition sind. Wir verwenden die Definition für Werte, ⟨*v*⟩, von oben, um zu steuern, dass die Auswertung der Argumente in Funktionsaufrufen von links nach rechts erfolgt.

Bisher (in Abschnitt §1.5 “Bedeutung von BSL Ausdrücken” und §2.3 “Bedeutung von Funktionsdefinitionen”) hatten wir die Auswertungspositionen so definiert, dass man bei Funktionsaufrufen die Argumente in beliebiger Reihenfolge auswerten kann. Die Auswertungskontexte wie oben definiert legen diese Reihenfolge fest, nämlich von links nach rechts. Wir werden später mehr zu diesem Unterschied sagen.

Sei *E* ein Auswertungskontext. Wir verwenden die Schreibweise *E*[*e*], um das "Loch" in dem Auswertungskontext durch einen Ausdruck *e* zu ersetzen.

Beispiel: Wenn *E* = `(* [] (+ 3 4))`, dann ist *E*[`(+ 1 2)`] = `(* (+ 1 2) (+ 3 4))`.

Mit Hilfe dieser Schreibweise können wir nun die Kongruenzregel so definieren:

(KONG): Falls *e-1* → *e-2*, dann *E[e-1]* → *E[e-2]*.

Wir schreiben die Auswertungsregeln generell so auf, dass wir jeder Regel einen Namen geben. Diese Regel heißt *(KONG)*.

Beispiel: Betrachten wir den Ausdruck *e* = `(* (+ 1 2) (+ 3 4))`. Diesen können wir zerlegen in einen Auswertungskontext *E* = `(* [] (+ 3 4))` und einen Ausdruck *e-1* = `(+ 1 2)`, so dass *e* = *E*[*e-1*]. Da wir *e-1* reduzieren können, `(+ 1 2)` → `3`, können wir auch dank *(KONG)* *e* reduzieren zu *E*[`3`] = `(* 3 (+ 3 4))`.

8.7 Nichtterminale und Metavariablen - Keine Panik!

In der Kongruenzregel von oben stehen Namen wie *e-1* und *e-2* für beliebige Ausdrücke und *E* für einen beliebigen Auswertungskontext.

Im Allgemeinen verwenden wir die Konvention, dass der Name x und Varianten wie *x-1* und *x-2* für beliebige Worte des Nichtterminals $\langle x \rangle$ steht (zum Beispiel für $\langle x \rangle = \langle e \rangle$ oder $\langle x \rangle = \langle v \rangle$). Derart verwendete Bezeichner wie *v-1* oder *e-2* nennt man auch *Metavariablen*, weil sie nicht Variablen von BSL sind, sondern Variablen sind, die für Teile von BSL Programmen stehen.

Wenn wir Nonterminale als Mengen interpretieren (nämlich die Menge der Worte für die es Ableitungsbäume gibt), so könnten wir die Regel von oben auch so aufschreiben:

Für alle *e-1*$\in\langle e \rangle$ und alle *e-2*$\in\langle e \rangle$ und alle $E \in \langle E \rangle$: Falls *e-1* $\rightarrow$ *e-2*, dann *E[e-1]* $\rightarrow$ *E[e-2]*.

Da dies die Regeln aber viel länger macht, verwenden wir die hier beschriebene Konvention.

8.8 Bedeutung von Programmen

Gemäß unserer Grammatik besteht ein Programm aus einer Sequenz von Definitionen und Ausdrücken. Die Auswertungsregel für Programme nennen wir *(PROG)*:

(PROG): Ein Programm wird von links nach rechts ausgeführt und startet mit der leeren Umgebung. Ist das nächste Programmelement eine Funktions- oder Strukturdefinition, so wird diese Definition in die Umgebung aufgenommen und die Ausführung mit dem nächsten Programmelement in der erweiterten Umgebung fortgesetzt. Ist das nächste Programmelement ein Ausdruck, so wird dieser gemäß der unten stehenden Regeln in der aktuellen Umgebung zu einem Wert ausgewert. Ist das nächste Programmelement eine Konstantendefinition `(define x e)`, so wird in der aktuellen Umgebung zunächst `e` zu einem Wert `v` ausgewertet und dann `(define x v)` zur aktuellen Umgebung hinzugefügt.

Beispiel: Das Programm ist:

```
(define (f x) (+ x 1))
(define c (f 5))
(+ c 3)
```

Im ersten Schritt wird `(define (f x) (+ x 1))` zur (leeren) Umgebung hinzugefügt. Die Konstantendefinition wird in der Umgebung `(define (f x) (+ x 1))` ausgewertet zu `(define c 6)` und dann dem Kontext hinzugefügt. Der Ausdruck `(+ c 3)` wird schliesslich in der Umgebung

```
(define (f x) (+ x 1))
(define c 6)
```

ausgewertet.

8.9 Bedeutung von Ausdrücken

Jeder Ausdruck wird in einer Umgebung *env* ausgewertet, wie sie im vorherigen Abschnitt definiert wurde. Um die Notation nicht zu überladen, werden wir *env* nicht explizit zu jeder Reduktionsregel dazuschreiben sondern als implizit gegeben annehmen.

Die Auswertung wird, wie aus Abschnitt §1.5 “Bedeutung von BSL Ausdrücken” bekannt, in Form von Reduktionsregeln der Form *e-1* → *e-2* definiert. Ein Ausdruck *e* wird ausgewertet, indem er solange reduziert wird, bis ein Wert herauskommt: *e* → *e-1* → ... → *v*.

Ein Fehler während der Auswertung äußert sich darin, dass die Reduktion "steckenbleibt", also wir bei einem Ausdruck ankommen, der kein Wert ist und der nicht weiter reduziert werden kann.

8.9.1 Bedeutung von Funktionsaufrufen

Funktionen werden unterschiedlich ausgeführt je nachdem ob der Funktionsname eine primitive Funktion oder eine in der Umgebung definierte Funktion ist. Im ersten Fall wird die primitive Funktion auf den Argumenten ausgewertet. Ist dies erfolgreich, so kann auf das Result reduziert werden. Ist dies nicht erfolgreich, so kann der Ausdruck nicht reduziert werden.

Ist die Funktion hingegen in der Umgebung definiert, so wird der Aufruf zum Body der Funktionsdefintion reduziert, wobei vorher alle Parameternamen durch die aktuellen Parameterwerte ersetzt werden. Dies ist die Bedeutung der Notation *e*[*name-1* := *v-1* ... *name-n* := *v-n*].

Die Reduktionsregeln sind also:

(FUN): Falls `(` `define` `(` *name name-1 ... name-n* `)` *e* `)` in der Umgebung, dann `(` *name v-1 ... v-n* `)` → *e*[*name-1* := *v-1* ... *name-n* := *v-n*]

(PRIM): Falls *name* eine primitive Funktion *f* ist und *f(v-1,...,v-n)=v*, dann `(` *name v-1 ... v-n* `)` → *v*.

8.9.2 Bedeutung von Konstanten

Konstanten werden ausgewertet, indem sie in der Umgebung nachgeschlagen werden:

(CONST): Falls `(` `define` *name v* `)` in der Umgebung, dann *name* → *v*.

8.9.3 Bedeutung konditionaler Ausdrücke

Konditionale Ausdrücke werden ausgewertet, wie schon in §2.4.2 “Bedeutung konditionaler Ausdrücke” beschrieben. Gemäß der Definition des Auswertungskontextes wird stets nur der erste Bedingungsausdruck ausgewertet. Je nachdem ob dieser `#true` oder `#false` ergibt, wird auf den Ergebnisausdruck oder den um die fehlgeschlagene Bedingung gekürzten `cond` Ausdruck reduziert:

(COND-True): `(` `cond` `[` `#true` *e* `]` ... `)` → *e*

(COND-False): `(` `cond` `[` `#false` *e-1* `]` `[` *e-2 e-3* `]` ... `)` → `(` `cond` `[` *e-2 e-3* `]` ... `)`

8.9.4 Bedeutung boolscher Ausdrücke

Die Definition der Auswertung boolscher Ausdrücke wertet die Bedingungen nur soweit wie nötig aus. Insbesondere wird die Auswertung abgebrochen, sobald einer der Ausdrücke `#false` ergibt.

Die ersten beiden Reduktionsregeln sind erforderlich, um zu überprüfen, dass alle Argumente boolsche Werte sind; andernfalls hätten die beiden Regeln zu `( and #true` *v* `)` → *v* zusammengefasst werden können. Insgesamt benötigen wir für boolsche Ausdrücke die folgenden vier Regeln:

(AND-1): `( and #true #true )` → `#true`
(AND-2): `( and #true #false )` → `#false`
(AND-3): `( and #false` ... `)` → `#false`
(AND-4): `( and #true` *e-1 e-2* ... `)` → `( and` *e-1 e-2* ... `)`

8.9.5 Bedeutung von Strukturkonstruktoren und Selektoren

Strukturdefinitionen definieren drei Arten von Funktionen: Konstruktoren wie `make-posn`, Selektoren wie `posn-x` und Prädikate wie `posn?`. Zu jeder dieser drei Arten benötigen wir eine Reduktionsregel.

Konstruktoren erzeugen Instanzen einer Struktur. Dies gelingt, wenn eine Struktur des gleichen Namens in der Umgebung zu finden ist, und diese so viele Felder wie der Konstruktor Parameter hat. Dies bringt uns zu folgender Regel:

(STRUCT-make): Falls `( define-struct` *name* `(` *name-1* ... *name-n* `) )` in der Umgebung, dann `(` `make-`*name* *v-1* ... *v-n* `)` → `<` `make-`*name* *v-1* ... *v-n* `>`.

Selektoraufrufe werden reduziert, indem in der Umgebung die Strukturdefinition nachgeschlagen wird, um den Namen des Feldes auf die Argumentposition des Konstruktoraufrufs abzubilden. Dann wird der entsprechende Wert des Feldes zurückgegeben:

(STRUCT-select): Falls `( define-struct` *name* `(` *name-1* ... *name-n* `) )` in der Umgebung, dann `(` *name-name-i* `<` `make-`*name* *v-1* ... *v-n* `>` `)` → *v-i*

Bei Prädikaten wird geschaut, ob es sich beim Argument des Prädikats um eine Strukturinstanz der in Frage stehenden Struktur handelt oder nicht, und je nachdem `#true` bzw. `#false` zurückgegeben:

(STRUCT-predtrue): `(` *name*`?` `<` `make-`*name* ... `>` `)` → `true`
(STRUCT-predfalse): Falls *v* nicht `<` `make-`*name* ... `>`, dann `(` *name*`?` *v* `)` → `#false`

8.10 Reduktion am Beispiel

Betrachten Sie folgendes Programm, dessen Bedeutung wir Schritt für Schritt mit Hilfe der Auswertungsregeln ermitteln werden:

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [#true (+ x 1)]
                    [#true x]))
(define c (make-s 5 (+ (* 2 3) 4)))
(f (s-x c))
```

- Gemäß *(PROG)* starten wir mit der leeren Umgebung *env* = leer. Das erste Programmelement ist eine Strukturdefinition, daher ist gemäß *(PROG)* die Umgebung im nächsten Schritt *env* = `(define-struct s (x y))`.

- Das nächste Programmelement ist eine Funktionsdefinition, daher ist gemäß *(PROG)* die Umgebung im nächsten Schritt *env* =

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [#true (+ x 1)]
                    [#true x]))
```

- Das nächste Programmelement ist eine Konstantendefinition. Gemäß *(PROG)* müssen wir also zunächst `(make-s 5 (+ (* 2 3) 4))` auswerten:
  - *e* = `(make-s 5 (+ (* 2 3) 4))` zerfällt in *E* = `(make-s 5 (+ [] 4))` und *e-1* = `(* 2 3)`. Gemäß *(PRIM)* gilt *e-1* → `6`; gemäß *(KONG)* gilt daher *e* → `(make-s 5 (+ 6 4))`.
  - *e* = `(make-s 5 (+ 6 4))` zerfällt in *E* = `(make-s 5 [])` und *e-1* = `(+ 6 4)`. Gemäß *(PRIM)* gilt *e-1* → `10`; gemäß *(KONG)* gilt daher *e* → `(make-s 5 10)`.
  - `(make-s 5 10)` → `<make-s 5 10>` gemäß *(STRUCT-make)*.

  Gemäß *(PROG)* ist unsere neue Umgebung daher nun *env* =

```
(define-struct s (x y))
(define (f x) (cond [(< x 1) (/ x 0)]
                    [#true (+ x 1)]
                    [#true x]))
(define c <make-s 5 10>)
```

- Das letzte Programmelement ist ein Ausdruck, den wir gemäß *(PROG)* in der aktuellen Umgebung auswerten:
  - *e* = `(f (s-x c))` zerfällt in *E* = `(f (s-x []))` und *e-1* = `c`. Gemäß *(CONST)* gilt `c` → `<make-s 5 10>`; gemäß *(KONG)* gilt daher *e* → `(f (s-x <make-s 5 10>))`.
  - *e* = `(f (s-x <make-s 5 10>))` zerfällt in *E* = `(f [])` und *e-1* = `(s-x <make-s 5 10>)`. Gemäß *(STRUCT-select)* gilt *e-1* → `5`; gemäß *(KONG)* gilt daher *e* → `(f 5)`.
  - `(f 5)` → `(cond [(< 5 1) (/ 5 0)] [#true (+ 5 1)] [#true 5])` gemäß *(FUN)*.
  - *e* = `(cond [(< 5 1) (/ 5 0)] [#true (+ 5 1)] [#true 5])` zerfällt in *E* = `(cond [[] (/ 5 0)] [#true (+ 5 1)] [#true 5])` und *e-1* = `(< 5 1)`. Gemäß *(PRIM)* gilt *e-1* → `#false`; gemäß *(KONG)* gilt daher *e* → `(cond [#false (/ 5 0)] [#true (+ 5 1)] [#true 5])`.
  - `(cond [#false (/ 5 0)] [#true (+ 5 1)] [#true 5])` → `(cond [#true (+ 5 1)] [#true 5])` gemäß *(COND-False)*.
  - `(cond [#true (+ 5 1)] [#true 5])` → `(+ 5 1)` gemäß *(COND-True)*.
  - `(+ 5 1)` → `6` gemäß *(PRIM)*.

8.11 Bedeutung von Daten und Datendefinitionen

Datendefinitionen haben auf das Programmverhalten keinen Einfluss, da sie in Form eines Kommentars definiert werden. Dennoch können wir ihnen eine präzise Bedeutung geben, die hilft, ihre Rolle zu verstehen.

Hierzu ist es wichtig, das *Datenuniversum* eines Programms zu verstehen. Das Datenuniversum umfasst alle Daten, die in einem gegebenen Programm potentiell vorkommen können. Welche Werte das sind, wird durch unsere Grammatik für Werte, $\langle v \rangle$, oben beschrieben. Allerdings können nicht alle Werte, die durch $\langle v \rangle$ beschrieben werden, in einem Programm vorkommen, sondern nur diese, für die die benutzen Strukturen auch wirklich im Programm definiert sind.

Beispiel: Ein Programm enthält die Strukturdefinitionen

```
(define-struct circle (center radius))
(define-struct rectangle (corner-ul corner-dr))
```

Das Datenuniversum für dieses Programm umfasst alle Werte der Basistypen, aber auch alle Strukturinstanzen, die sich auf Basis dieser Strukturdefinitionen bilden lassen, also zum Beispiel `<make-circle 5 6>` aber auch:

```
<make-circle <make-circle <make-rectangle 5 <make-
rectangle #true "asdf">> 77> 88>
```

Das Datenuniversum sind also alle Werte, die sich durch die Grammatik von $\langle v \rangle$ bilden lassen, eingeschränkt auf die Strukturen, die in dem Programm definiert sind.

Eine Strukturdefinition erweitert also das Datenuniversum um neue Werte, nämlich alle Werte, in denen mindestens einmal diese Struktur verwendet wird.

Eine Datendefinition, auf der anderen Seite, erweitert nicht das Datenuniversum. Eine Datendefinition definiert eine *Teilmenge* des Datenuniversums.

Beispiel:

```
; a Posn is a structure: (make-posn Number Number)
```

`<make-posn 3 4 >` ist ein Element der definierten Teilmenge, aber `<make-posn #true "x" >` oder `<make-posn <make-posn 3 4> 5>` sind es nicht.

Eine Datendefinition beschreibt im Allgemeinen eine kohärente Teilmenge des Datenuniversums. Funktionen können durch ihre Signatur deutlich machen, welche Werte des Datenuniversums sie als Argumente akzeptieren und welche Ergebnisse sie produzieren.

8.12 Refactoring von Ausdrücken und Schliessen durch Gleichungen

Wir hatten in Abschnitt §1.5 “Bedeutung von BSL Ausdrücken” vorgestellt, wie man auf Basis der Reduktionsregeln eine Äquivalenzrelation auf Ausdrücken definieren

kann. Diese Äquivalenzen können zum Refactoring von Programmen verwendet werden - also Programmänderungen, die nicht die Bedeutung verändern aber die Struktur des Programms verbessern. Außerdem können sie verwendet werden, um Eigenschaften seines Programmes herzuleiten, zum Beispiel dass die Funktion `overlaps-circle` aus dem vorherigen Kapitel kommutativ ist, also `(overlaps-circle c1 c2)` ≡ `(overlaps-circle c2 c1)`.

Die Äquivalenzrelation aus Abschnitt §1.5 “Bedeutung von BSL Ausdrücken” war allerdings zu klein für viele praktische Zwecke, denn sie erfordert beispielsweise, dass wir Funktionsaufrufe nur dann auflösen können, wenn alle Argumente Werte sind.

BSL hat jedoch eine bemerkenswerte Eigenschaft, die es uns erlaubt, eine viel mächtigere Äquivalenzrelation zu definieren: Es ist für das Ergebnis eines Programms nicht von Bedeutung, in welcher Reihenfolge Ausdrücke ausgewertet werden. Insbesondere ist es nicht notwendig, vor einem Funktionsaufruf die Argumente auszuwerten; man kann auch einfach die Argumentausdrücke verwenden.

Die Idee wird durch folgenden, allgemeineren Auswertungskontext ausgedrückt:

⟨*E*⟩ ::= `[]`
| `(` ⟨*name*⟩ ⟨*e*⟩* ⟨*E*⟩ ⟨*e*⟩* `)`
| `(` `cond` {`[` ⟨*e*⟩ ⟨*e*⟩ `]`}* `[` ⟨*E*⟩ ⟨*e*⟩ `]` {`[` ⟨*e*⟩ ⟨*e*⟩ `]`}* `)`
| `(` `cond` {`[` ⟨*e*⟩ ⟨*e*⟩ `]`}* `[` ⟨*e*⟩ ⟨*E*⟩ `]` {`[` ⟨*e*⟩ ⟨*e*⟩ `]`}* `)`
| `(` `and` ⟨*e*⟩* ⟨*E*⟩ ⟨*e*⟩*

Zusammen mit der folgenden Kongruenzregel für unsere Äquivalenzrelation, drückt dieser Auswertungskontext aus, dass überall "gleiches mit gleichem" ersetzt werden darf:

(EKONG): Falls *e-1* ≡ *e-2*, dann *E[e-1]* ≡ *E[e-2]*.

Eine Äquivalenzsrelation sollte möglichst groß sein, damit wir so viele Äquivalenzen wie möglich zeigen können. Gleichzeitig sollte sie korrekt sein. Dies bedeutet, dass äquivalente Programme das gleiche Verhalten haben, also insbesondere – sofern sie terminieren – bei Auswertung den gleichen Wert ergeben.

Wir definieren nun nach und nach die Regeln, die für die Äquivalenzrelation gelten sollen. Zunächst einmal sollte es tatsächlich eine Äquivalenzrelation — also reflexiv, kommutativ und transitiv — sein:

(EREFL): *e* ≡ *e*.

(EKOMM): Falls *e1* ≡ *e2*, dann *e2* ≡ *e1*.

(ETRANS): Falls *e-1* ≡ *e-2* und *e-2* ≡ *e-3*, dann *e-1* ≡ *e-3*.

Die Verknüpfung zur Auswertungsrelation wird durch diese Regel geschaffen: Reduktion erhält Äquivalenz.

(ERED): Falls *e-1* → *e-2* dann *e-1* ≡ *e-2*.

Damit wir auch "symbolisch" Funktionen auswerten können, erweitern wir die Regel für Funktionsaufrufe, so dass es für die Bestimmung von Äquivalenzen nicht notwendig ist, die Argumente auszuwerten.

(EFUN): Falls `(` `define` `(` *name name-1 ... name-n* `)` *e* `)` in der Umgebung, dann `(` *name e-1 ... e-n* `)` ≡ *e*[*name-1* := *e-1* ... *name-n* := *e-n*]

Bei der Konjunktion wissen wir, dass der Gesamtausdruck zu `#false` auswertet (oder nicht terminiert), wenn mindestens eines der Argumente äquivalent zu `#false` ist.

(EAND): `(` `and` ... `#false` ... `)` ≡ `#false`

Außerdem können wir Wissen, das wir über die eingebauten Funktionen haben, beim Schliessen mit Äquivalenzen nutzen. Beispielsweise wissen wir, dass `(+ a b)` $\equiv$ `(+ b a)`. Wir fassen die Menge der Äquivalenzen, die für die eingebauten Funktionen gelten unter dem Namen *(EPRIM)* zusammen.

Einen kleinen Hakenfuss gibt es allerdings doch noch. Man würde sich von einer Äquivalenzrelation für Programme wünschen, dass folgende Eigenschaft gilt: Falls *e-1* $\equiv$ *e-2* und *e-1* $\rightarrow^*$ *v*, dann auch *e-2* $\rightarrow^*$ *v*. Diese Eigenschaft gilt jedoch nicht, weil es sein kann, dass *e-1* terminiert aber *e-2* nicht.

Beispiel: Betrachten Sie folgendes Programm:

```
(define (f x) (f x))
(define (g x) 42)
(g (f 1))
```

Da `(f 1)` $\rightarrow$ `(f 1)`, terminiert die Berechnung des Arguments für `g` nicht, und gemäß der Kongruenzregel gilt damit `(g (f 1))` $\rightarrow$ `(g (f 1))`, daher terminiert die Berechnung des Ausdrucks `(g (f 1))` $\rightarrow$ `(g (f 1))` nicht. Auf der anderen Seite gilt jedoch gemäß *(EFUN)* `(g (f 1))` $\equiv$ 42. Man muss daher bei der Verwendung der Äquivalenzregeln berücksichtigen, dass die Äquivalenz nur unter der Voraussetzung gilt, dass die Terme auf beiden Seiten terminieren..

Es gilt jedoch folgende etwas schwächere Eigenschaft, die wir ohne Beweis aufführen:

Falls *e-1* $\equiv$ *e-2* und *e-1* $\rightarrow^*$ *v-1* und *e-2* $\rightarrow^*$ *v-2*, dann *v1* = *v2*.

Wenn also *e-1* und *e-2* gleich sind und beide terminieren, dann ist der Wert, der herauskommt, gleich.

9 Daten beliebiger Größe

Die Datentypen, die wir bisher gesehen haben, repräsentieren grundsätzlich Daten, die aus einer festen Anzahl von atomaren Daten bestehen. Dies liegt daran, dass jede Datendefinition nur andere Datendefinitionen verwenden darf, die vorher bereits definiert wurden. Betrachten wir zum Beispiel

```
(define-struct gcircle (center radius))
; A GCircle is (make-gcircle Posn Number)
; interp. the geometrical representation of a circle
```

so wissen wir, dass eine `Posn` aus zwei und eine `Number` aus einem atomaren Datum (jeweils Zahlen) besteht und damit ein `GCircle` aus genau drei atomaren Daten.

In vielen Situationen wissen wir jedoch nicht zum Zeitpunkt des Programmierens, aus wievielen atomaren Daten ein zusammengesetztes Datum besteht. In diesem Kapitel befassen wir uns damit, wie wir Daten beliebiger (und zum Zeitpunkt des Programmierens unbekannter) Größe repräsentieren und Funktionen, die solche Daten verarbeiten, programmieren können.

9.1 Rekursive Datentypen

Betrachten wir als Beispiel ein Programm, mit dem Stammbäume von Personen verwaltet werden können. Jede Person im Stammbaum hat einen Vater und eine Mutter; manchmal sind Vater oder Mutter einer Person auch unbekannt.

Mit den bisherigen Mitteln könnten wir zum Beispiel so vorgehen, um die Vorfahren einer Person zu repräsentieren:

```
(define-struct parent (name grandfather grandmother))
; A Parent is: (make-parent String String String)
; interp. the name of a person with the names of his/her
grandparents

(define-struct person (name father mother))
; A Person is: (make-person String Parent Parent)
; interp. the name of a person with his/her parents
```

Allerdings können wir so nur die Vorfahren bis zu genau den Großeltern repräsentieren. Natürlich könnten wir noch weitere Definitionen für die Urgroßeltern usw. hinzufügen, aber stets hätten wir eine beim Programmieren festgelegte Größe. Außerdem verstoßen wir gegen das DRY Prinzip, denn die Datendefinitionen sehen für jede Vorgängergeneration sehr ähnlich aus.

Was können wir machen um mit diesem Problem umzugehen? Betrachten wir etwas genauer, was für Daten wir hier eigentlich modellieren wollen. Die Erkenntnis, die wir hier benötigen, ist, dass ein Stammbaum eine rekursive Struktur hat: Der Stammbaum einer Person besteht aus dem Namen der Person und dem Stammbaum seiner Mutter und dem Stammbaum seiner Eltern. Ein Stammbaum hat also eine Eigenschaft,

die man auch *Selbstähnlichkeit* nennt: Ein Teil eines Ganzen hat die gleiche Struktur wie das Ganze.

Dies bedeutet, dass wir die Restriktion fallen lassen müssen, dass in einer Datendefinition nur die Namen bereits vorher definierter Datentypen auftauchen dürfen. In unserem Beispiel schreiben wir:

```
(define-struct person (name father mother))

; A FamilyTree is: (make-person String FamilyTree FamilyTree)
; interp. the name of a person and the tree of his/her
parents.
```

Die Definition von `FamilyTree` benutzt also selber wieder `FamilyTree`. Erstmal ist nicht ganz klar, was das bedeuten soll. Vor allen Dingen ist aber auch nicht klar, wie wir einen Stammbaum erzeugen sollen. Wenn wir versuchen, einen zu erzeugen, bekommen wir ein Problem:

```
  (make-person "Heinz" (make-person "Horst" (make-person "Joe"
...)))
```

Wir können überhaupt gar keinen Stammbaum erzeugen, weil wir zur Erzeugung bereits einen Stammbaum haben müssen. Ein Ausdruck, der einen Stammbaum erzeugt, wäre also unendlich groß.

Bei genauer Überlegung sind Stammbäume aber niemals unendlich groß, sondern sie enden bei irgendeiner Generation - zum Beispiel weil die Vorfahren unbekannt oder nicht von Interesse sind.

Diesem Tatbestand können wir dadurch Rechnung tragen, dass wir aus `FamilyTree` einen Summentyp machen, und zwar wie folgt:

```
(define-struct person (name father mother))

; A FamilyTree is either:
; - (make-person String FamilyTree FamilyTree)
; - #false
; interp. either the name of a person and the tree of its par-
ents,
; or #false if the person is not known/relevant.
```

Dieser neue, nicht-rekursive Basisfall erlaubt es uns nun, Werte dieses Typen zu erzeugen. Hier ist ein Beispiel:

```
(define HEINZ
  (make-person "Heinz"
               (make-person "Elke" #false #false)
               (make-person "Horst"
                            (make-person "Joe"
                                         #false
                                         (make-person "Rita"
                                                      #false
```

```
                                                    #false))
                                  #false)))
```

Was bedeutet also so eine rekursive Datendefinition? Wir haben bisher Datentypen immer als Mengen von Werten aus dem Datenuniversum interpretiert, und dies ist auch weiterhin möglich, nur die Konstruktion der Menge, die der Typ repräsentiert, ist nun etwas aufwendiger:

Sei ft_0 die leere Menge, ft_1 die Menge $\{\#\texttt{false}\}$, ft_2 die Vereinigung aus $\{\#\texttt{false}\}$ und der Menge der `(make-person name false false)` für alle Strings `name`. Im Allgemeinen sei ft_{i+1} die Vereinigung aus $\{\#\texttt{false}\}$ und der Menge der `(make-person name p1 p2)` für alle Strings `name` sowie für alle `p1` und alle `p2` aus ft_i. Beispielsweise ist `HEINZ` ein Element von ft_5 (und damit auch ft_6, ft_7 usw.) aber nicht von ft_4.

Dann ist die Bedeutung von `FamilyTree`, ft, die Vereinigung aller dieser Mengen, also ft_0 vereinigt mit ft_1 vereinigt mit ft_2 vereinigt mit ft_3 vereinigt mit ... .

In mathematischer Schreibweise können wir die Konstruktion so zusammenfassen:

$$
\begin{aligned}
ft_0 &= \varnothing \\
ft_{i+1} &= \{\#\texttt{false}\} \cup \{(\texttt{make-person}\ n\ p_1\ p_2) \mid n \in \mathit{String}, p_1 \in ft_i, p_2 \in ft_i\} \\
ft &= \bigcup_{i \in \mathbb{N}} ft_i
\end{aligned}
$$

Es ist nicht schwer zu sehen, dass stets $ft_i \subseteq ft_{i+1}$; die nächste Menge umfasst also stets die vorherige.

Aus dieser Mengenkonstruktion wird klar, wieso rekursive Datentypen es ermöglichen, Daten beliebiger Größe zu repräsentieren: Jede Menge ft_i enthält die Werte, deren maximale Tiefe in Baumdarstellung i ist. Da wir alle ft_i miteinander vereinigen, ist die Tiefe (und damit auch die Größe) unbegrenzt.

Diese Mengenkonstruktion kann für jeden rekursiven Datentyp definiert werden. Falls es mehrere rekursive Alternativen gibt, so ist die i-te Menge die Vereinigung der i-ten Mengen für jede Alternative. Gäbe es also beispielsweise noch eine zusätzliche `FamilyTree` Alternative `(make-celebrity String Number FamilyTree FamilyTree)` bei der neben dem Namen auch noch das Vermögen angegeben wird, so wäre

$$
\begin{aligned}
ft_{i+1} = {} & \{\#\texttt{false}\} \cup \{(\texttt{make-person}\ n\ p_1\ p_2) \mid n \in \mathit{String}, p_1 \in ft_i, p_2 \in ft_i\} \\
& \cup \{(\texttt{make-celebrity}\ n\ w\ p_1\ p_2) \mid n \in \mathit{String}, w \in \mathit{Number}, p_1 \in ft_i, p_2 \in ft_i\}
\end{aligned}
$$

9.2 Programmieren mit rekursiven Datentypen

Im letzten Abschnitt haben wir gesehen, wie man rekursive Datentypen definiert, was sie bedeuten, und wie man Instanzen dieser Datentypen erzeugt. Nun wollen wir überlegen, wie man Funktionen programmieren kann, die Instanzen solcher Typen als Argumente haben oder als Resultat produzieren.

Betrachten Sie dazu folgende Aufgabe: Programmieren sie eine Funktion, die herausfindet, ob es im Stammbaum einer Person einen Vorfahren mit einem bestimmten Namen gibt.

Eine Signatur, Aufgabenbeschreibung und Tests sind dazu schnell definiert:

```
; FamilyTree String -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor HEINZ "Joe") #true)
(check-expect (person-has-ancestor HEINZ "Emil") #false)
(define (person-has-ancestor p a) ...)
```

Da `FamilyTree` ein Summentyp ist, sagt unser Entwurfsrezept, dass wir eine Fallunterscheidung machen. In jedem Zweig der Fallunterscheidung können wir die Selektoren für die Felder eintragen. In unserem Beispiel ergibt sich:

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         ...(person-name p) ...
         ...(person-father p)...
         ...(person-mother p) ...]
        [else ...]))
```

Die Ausdrücke `(person-father a)` und `(person-mother a)` stehen für Werte, die einen komplexen Summentypen, nämlich wieder `FamilyTree` haben. Offensichtlich hat eine Person einen Vorfahren `a`, wenn die Person entweder selber `a` ist oder die Mutter oder der Vater einen Vorfahr mit dem Namen `a` hat.

Wir interpretieren das Wort "Vorfahr" so, dass eine Person ihr eigener Vorfahr ist. Wie müsste das Programm aussehen, um die nicht-reflexive Interpretation des Worts "Vorfahr" zu realisieren?

Unser Entwurfsrezept schlägt für Fälle, in denen Felder einer Strukur selber einen komplexen Summen-/Produkttyp haben, vor, eigene Hilfsfunktionen zu definieren. Dies können wir wie folgt andeuten:

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         ... (person-name p) ...
         ... (father-has-ancestor (person-father p) ...)...
         ... (mother-has-ancestor (person-mother p) ...)...]
        [else ...]))
```

Wenn wir uns jedoch etwas genauer überlegen, was `father-has-ancestor` und `mother-has-ancestor` tun sollen, stellen wir fest, dass sie die gleiche Signatur und Aufgabenbeschreibung haben wie `person-has-ancestor`! Das bedeutet, dass die Templates für diese Funktionen wiederum jeweils zwei neue Hilfsfunktionen erfordern. Da wir nicht wissen, wie tief der Stammbaum ist, können wir auf diese Weise das Programm also gar nicht realisieren.

Zum Glück haben wir aber bereits eine Funktion, deren Aufgabe identisch mit der von `father-has-ancestor` und `mother-has-ancestor` ist, nämlich `person-has-ancestor` selbst. Dies motiviert das folgende Template:

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         ... (person-name p) ...
         ... (person-has-ancestor (person-father p) ...)...
         ... (person-has-ancestor (person-mother p) ...)...]
        [else ...]))
```

Die Struktur der Daten diktiert also die Struktur der Funktionen. Dort wo die Daten rekursiv sind, sind auch die Funktionen, die solche Daten verarbeiten, rekursiv.

Die Vervollständigung des Templates ist nun eifach:

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         (or
          (string=? (person-name p) a)
          (person-has-ancestor (person-father p) a)
          (person-has-ancestor (person-mother p) a))]
        [else #false]))
```

Die erfolgreich ablaufenden Tests illustrieren, dass diese Funktion anscheinend das tut, was sie soll, aber wieso?

Vergleichen wir die Funktion mit einem anderen Ansatz, der nicht so erfolgreich ist. Betrachten wir nochmal den Anfang der Programmierung der Funktion, diesmal mit anderem Namen:

```
; FamilyTree -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor-stupid HEINZ "Joe") #true)
(check-expect (person-has-ancestor-stupid HEINZ "Emil") #false)
(define (person-has-ancestor-stupid p a) ...)
```

Wenn wir schon eine Funktion hätten, die bestimmen kann, ob eine Person einen bestimmten Vorfahren hat, könnten wir einfach diese Funktion aufrufen. Aber zum Glück sind wir ja schon gerade dabei, diese Funktion zu programmieren. Also rufen wir doch einfach diese Funktion auf:

```
; FamilyTree -> Boolean
; determines whether person p has an ancestor a
(check-expect (person-has-ancestor-stupid HEINZ "Joe") #true)
(check-expect (person-has-ancestor-stupid HEINZ "Emil") #false)
(define (person-has-ancestor-stupid p a)
  (person-has-ancestor-stupid p a))
```

Irgendwie scheint diese Lösung zu einfach zu sein. Ein Ausführen der Tests bestätigt den Verdacht. Allerdings schlagen die Tests nicht fehl, sondern die Ausführung der Tests terminiert nicht und wir müssen sie durch Drücken auf die "Stop" Taste abbrechen.

Wieso funktioniert `person-has-ancestor` aber nicht `person-has-ancestor-stupid`?

Zunächst einmal können wir die Programme operationell miteinander vergleichen. Wenn wir den Ausdruck `(person-has-ancestor-stupid HEINZ "Joe")` betrachten, so sagt unsere Reduktionssemantik:

`(person-has-ancestor-stupid HEINZ "Joe")` → `(person-has-ancestor-stupid HEINZ "Joe")`

Es gibt also keinerlei Fortschritt bei der Auswertung. Dies erklärt, wieso die Ausführung nicht stoppt.

Dies ist anders bei `(person-has-ancestor HEINZ "Joe")`. Die rekursiven Aufrufe rufen `person-has-ancestor` stets auf Vorfahren der Person auf. Da der Stammbaum nur eine endliche Tiefe hat, muss irgendwann `(person-has-ancestor #false "Joe")` aufgerufen werden, und wir landen im zweiten Fall des konditionalen Ausdrucks, der keine rekursiven Ausdrücke mehr enthält.

Wir können das Programm auch aus Sicht der Bedeutung der rekursiven Datentypdefinition betrachten. Für jede Eingabe `p` gibt es ein minimales i so dass `p` in ft_i ist. Für `HEINZ` ist dieses i=5. Da die Werte der `mother` und `father` Felder von `p` damit in ft_{i-1} sind, ist klar, dass der `p` Parameter bei allen rekursiven Aufrufe in ft_{i-1} ist. Da das Programm offensichtlich für Werte aus ft_0 (im Beispiel die Menge {#false}) terminiert, ist klar, dass dann auch alle Aufrufe mit Werten aus ft_1 terminieren müssen, damit dann aber auch die Aufrufe mit Werten aus ft_2 und so weiter. Damit haben wir gezeigt, dass die Funktion für alle Werte aus ft_i für ein beliebiges i wohldefiniert ist — mit anderen Worten: für alle Werte aus `FamilyTree`.

Dieses informell vorgetragene Argument aus dem vorherigen Absatz ist mathematisch gesehen ein Induktionsbeweis.

Bei `person-has-ancestor-stupid` ist die Lage anders, da im rekursiven Aufruf das Argument eben nicht aus ft_{i-1} ist.

Die Schlussfolgerung aus diesen Überlegungen ist, dass Rekursion in Funktionen unproblematisch und wohldefiniert ist, solange sie der Struktur der Daten folgt. Da Werte rekursiver Datentypen eine unbestimmte aber endliche Größe haben, ist diese sogenannte *strukturelle Rekursion* stets wohldefiniert. Unser Entwurfsrezept schlägt vor, dass immer dort wo Datentypen rekursiv sind, auch die Funktionen, die darauf operieren, (strukturell) rekursiv sein sollen.

Bevor wir uns das angepasste Entwurfsrezept im Detail anschauen, wollen wir erst noch überlegen, wie Funktionen aussehen, die Exemplare rekursiver Datentypen *produzieren*.

Als Beispiel betrachten wir eine Funktion, die sehr nützlich ist, um den eigenen Stammbaum etwas eindrucksvoller aussehen zu lassen, nämlich eine, mit der der Name aller Vorfahren um einen Titel ergänzt werden kann.

Hier ist die Signatur, Aufgabenbeschreibung und ein Test für diese Funktion:

```
; FamilyTree -> FamilyTree
; prefixes all members of a family tree p with title t
(check-expect
 (promote HEINZ "Dr. ")
 (make-person
```

```
   "Dr. Heinz"
   (make-person "Dr. Elke" #false #false)
   (make-person
    "Dr. Horst"
    (make-person "Dr. Joe" #false
                 (make-person "Dr. Rita" #false #false)) #false)))

(define (promote p t) ...)
```

Die Funktion konsumiert und produziert also gleichzeitig einen Wert vom Typ `FamilyTree`. Das Template für solche Funktionen unterscheidet sich nicht von dem für `person-has-ancestor`. Auch hier wenden wir wieder dort Rekursion an, wo auch der Datentyp rekursiv ist:

```
(define (promote p t)
  (cond [(person? p)
           ...(person-name p)...
           ...(promote (person-father p) ...)...
           ...(promote (person-mother p) ...)...]
        [else ...]))
```

Nun ist es recht einfach, die Funktion zu vervollständigen. Um Werte vom Typ `FamilyTree` zu produzieren, bauen wir die Resultate der rekursiven Aufrufe in einen Aufrufe des `make-person` Konstruktors ein:

```
(define (promote p t)
  (cond [(person? p)
         (make-person
           (string-append t (person-name p))
           (promote (person-father p) t)
           (promote (person-mother p) t))]
        [else p]))
```

9.3 Listen

Die `FamilyTree` Datendefinition von oben steht für eine Menge von Bäumen, in denen jeder Knoten genau zwei ausgehende Kanten hat. Selbstverständlich können wir auch auf die gleiche Weise Bäume repräsentieren, die drei oder fünf ausgehende Kanten haben — indem wir eine Alternative des Summentyps haben, in der der Datentyp drei bzw. fünfmal vorkommt.

Ein besonders wichtiger Spezialfall ist der, wo jeder Knoten genau eine ausgehende Kante hat. Diese degenerierten Bäume nennt man auch *Listen*.

9.3.1 Listen, hausgemacht

Hier ist eine mögliche Definition für Listen von Zahlen:

```
(define-struct lst (first rest))

; A List-of-Numbers is either:
; - (make-lst Number List-Of-Numbers)
; - #false
; interp. the head and rest of a list, or the empty list
```

Hier ist ein Beispiel, wie wir eine Liste mit den Zahlen 1 bis 3 erzeugen können:

```
(make-lst 1 (make-lst 2 (make-lst 3 #false)))
```

Der Entwurf von Funktionen auf Listen funktioniert genau wie der Entwurf von Funktionen auf allen anderen Bäumen. Betrachten wir als Beispiel eine Funktion, die alle Zahlen in einer Liste aufaddiert.

Hier ist die Spezifikation dieser Funktion:

```
; List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (sum (make-lst 1 (make-lst 2 (make-
lst 3 #false)))) 6)
(define (sum l) ...)
```

Für das Template schlägt das Entwurfsrezept vor, die verschiedenen Alternativen zu unterscheiden und in den rekursiven Alternativen rekursive Funktionsaufrufe einzubauen:

```
(define (sum l)
  (cond [(lst? l) ... (lst-first l) ... (sum (lst-
rest l)) ...]
        [else ...]))
```

Auf Basis dieses Templates ist die Vervollständigung nun einfach:

```
(define (sum l)
  (cond [(lst? l) (+ (lst-first l) (sum (lst-rest l)))]
        [else 0]))
```

9.3.2 Listen aus der Konserve

Weil Listen so ein häufig vorkommender Datentyp sind, gibt es in BSL vordefinierte Funktionen für Listen. Wie `List-of-Numbers` zeigt, benötigen wir diese vordefinierten Funktionen eigentlich nicht, weil wir Listen einfach als degenerierte Bäume repräsentieren können. Dennoch machen die eingebauten Listenfunktionen das Programmieren mit Listen etwas komfortabler und sicherer.

Die eingebaute Konstruktorfunktion für Listen in BSL heißt `cons`; die leere Liste wird nicht durch `#false` sondern durch die spezielle Konstante `empty` repräsentiert.

Es ist sinnvoll, die leere Liste durch einen neuen Wert zu repräsentieren, der für nichts anderes steht, denn dann kann es niemals zu Verwechslungen kommen. Wenn

wir etwas analoges in unserer selbstgebauten Datenstruktur für Listen machen wollten, könnten wir dies so erreichen, indem wir eine neue Struktur ohne Felder definieren und ihren einzigen Wert als Variable definieren:

```
(define-struct empty-lst ())
(define EMPTYLIST (make-empty-lst))
```

Dementsprechend würde die Beispielliste von oben nun so konstruiert:

```
(make-lst 1 (make-lst 2 (make-lst 3 EMPTYLIST)))
```

Die `cons` Operation entspricht unserem `make-lst` von oben, allerdings mit einem wichtigen Unterschied: Sie überprüft zusätzlich, dass das zweite Argument auch eine Liste ist:

```
> (cons 1 2)
cons: second argument must be a list, but received 1 and 2
```

Man kann sich `cons` also so vorstellen wie diese selbstgebaute Variante von `cons` auf Basis von `List-Of-Numbers`:

```
(define (our-cons x l)
  (if (or (empty-lst? l) (lst? l))
      (make-lst x l)
      (error "second argument of our-cons must be a list")))
```

Die wichtigsten eingebauten Listenfunktionen sind:
`empty`, `empty?`, `cons`, `first`, `rest` und `cons?`.
Sie entsprechen in unserem selbstgebauten Listentyp:
`EMPTYLIST`, `empty-lst?`, `our-cons`, `lst-first`, `lst-rest` und `lst?`.
Die Funktion, die `make-lst` entspricht, wird von BSL versteckt, um sicherzustellen, dass alle Listen stets mit `cons` konstruiert werden und dementsprechend die Invariante forciert wird, dass das zweite Feld der Struktur auch wirklich eine Liste ist. Mit unseren bisherigen Mitteln können wir dieses "Verstecken" von Funktionen nicht nachbauen; dazu kommen wir später, wenn wir über Module reden.

Unser Beispiel von oben sieht bei Nutzung der eingebauten Listenfunktionen also so aus:

```
; A List-of-Numbers is one of:
; - (cons Number List-Of-Numbers)
; - empty

; List-Of-Numbers -> Number
; adds up all numbers in a list
(check-expect (sum (cons 1 (cons 2 (cons 3 empty)))) 6)
(define (sum l)
  (cond [(cons? l) (+ (first l) (sum (rest l)))]
        [else 0]))
```

9.3.3 Die `list` Funktion

Es stellt sich schnell als etwas mühselig heraus, Listen mit Hilfe von `cons` und `empty` zu konstruieren. Aus diesem Grund gibt es etwas syntaktischen Zucker, um Listen komfortabler zu erzeugen: Die `list` Funktion.

Mit der `list` Funktion können wir die Liste `(cons 1 (cons 2 (cons 3 empty)))` so erzeugen:

```
(list 1 2 3)
```

Die `list` Funktion ist jedoch nur etwas syntaktischer Zucker, der wie folgt definiert ist:

```
(list exp-1 ... exp-n)
```

steht für n verschachtelte `cons` Ausdrücke:

```
(cons exp-1 (cons ... (cons exp-n empty)))
```

Es ist wichtig, zu verstehen, dass dies wirklich nur eine abkürzende Schreibweise ist. Auch wenn Sie mittels `list` leichter Listen definieren können, ist es wichtig, stets im Kopf zu behalten, dass dies nur eine Kurzschreibweise für die Benutzung von `cons` und `empty` ist, denn nur so ist das Entwurfsrezept für rekursive Datentypen auch auf Listen anwendbar.

9.3.4 Datentypdefinitionen für Listen

Wir haben oben eine Datendefinition `List-of-Numbers` gesehen. Sollen wir auch für `List-of-Strings` oder `List-of-Booleans` eigene Datendefinitionen schreiben? Sollen diese die gleiche Struktur benutzen, oder sollen wir separate Strukturen für jede dieser Datentypen haben?

Es ist sinnvoll, dass alle diese Datentypen die gleiche Struktur benutzen, nämlich in Form der eingebauten Listenfunktionen. Hierfür gibt es zwei Gründe: Erstens wären all diese Strukturen sehr ähnlich und wir würden damit gegen das DRY Prinzip verstoßen. Zweitens gibt es eine ganze Reihe von Funktionen, die auf *beliebigen* Listen arbeiten, zum Beispiel eine Funktion `second`, die das zweite Listenelement einer Liste zurückgibt:

```
(define (second l)
  (if (or (empty? l) (empty? (rest l)))
      (error "need at least two list elements")
      (first (rest l))))
```

Diese Funktion (die übrigens schon vordefiniert ist, genau wie `third`, `fourth` und so weiter) funktioniert für beliebige Listen, unabhängig von der Art der gespeicherten Daten. Solche Funktionen könnten wir nicht schreiben, wenn wir für je nach Typ der Listenelemente andere Strukturen verwenden würden.

Der häufigste Anwendungsfall von Listen ist der, dass die Listen *homogen* sind. Das bedeutet, dass alle Listenelemente einen gemeinsamen Typ haben. Dies ist keine

sehr große Einschränkung, denn dieser gemeinsame Typ kann beispielsweise auch ein Summentyp mit vielen Alternativen sein. Für diesen Fall verwenden wir Datendefinitionen mit *Typparametern*, und zwar so:

```
; A (List-of X) is one of:
; - (cons X (List-of X)
; - empty
```

Diese Datendefinitionen benutzen wir, indem wir einen Typ für den Typparameter angeben. Hierzu verwenden wir die Syntax für Funktionsanwendung; wir schreiben also `(List-of String)`, `(List-of Boolean)`, `(List-of (List-of String))` oder `(List-of FamilyTree)` verwenden und meinen damit implizit die oben angeführte Datendefinition.

Was aber ist ein geeigneter Datentyp für die Signatur von `second` oben, also im Allgemeinen für Funktionen, die auf beliebigen Listen funktionieren?

Um deutlich zu machen, dass die Funktionen für Listen mit beliebigem Elementtyp funktionieren, sagen wir dies in der Signatur explizit. Im Beispiel der Funktion `second` sieht das so aus:

```
; [X] (List-of X) -> X
(define (second l) ...)
```

Das [X] am Anfang der Signatur sagt, dass diese Funktion die nachfolgende Signatur für jeden Typ X hat, also (List-of X) -> X für jede mögliche Ersetzung von X durch einen Typen. Man nennt Variablen wie `X` *Typvariablen*. Also hat `second` zum Beispiel den Typ (List-of Number) -> Number oder(List-of (List-of String) -> (List-of String).

Allerdings werden wir im Moment nur im Ausnahmefall Funktionen wie `second` selber programmieren. Die meisten Funktionen, die wir im Moment programmieren wollen, verarbeiten Listen mit einem konkreten Elementtyp. Auf sogenannte *polymorphe* Funktionen wie `second` werden wir später noch zurückkommen.

9.3.5 Aber sind Listen wirklich rekursive Datenstrukturen?

Wenn man sich Listen aus der realen Welt anschaut (auf einem Blatt Papier, in einer Tabellenkalkulation etc.), so suggerieren diese häufig keine rekursive, verschachtelte Struktur sondern sehen "flach" aus. Ist es also "natürlich", Listen so wie wir es getan haben, über einen rekursiven Datentyp - als degenerierten Baum - zu definieren?

In vielen (vor allem älteren) Programmiersprachen gibt es eine direktere Unterstützung für Listen. Listen sind in diesen Programmiersprachen fest eingebaut. Listen sind in diesen Sprachen nicht rekursiv; stattdessen wird häufig über einen Index auf die Elemente der Liste zugegriffen. Um die Programmierung mit Listen zu unterstützen, gibt es einen ganzen Satz von Programmiersprachenkonstrukten (wie z.B. verschiedenen Schleifen- und Iterationskonstrukten), die der Unterstützung dieser fest eingebauten Listen dienen.

Aus Hardware-Sicht sind solche Listen, über die man mit einem Index zugreift, sehr natürlich, denn sie entsprechen gut dem von der Hardware unterstützen Zugriff auf den Hauptspeicher. Insofern sind diese speziellen Listenkonstrukte zum Teil durch ihre

Es gibt allerdings auch durchaus Situationen, in denen rekursiv aufgebaute Listen effizienter sind. Mehr dazu später.

Effizienz begründet. Es gibt auch in Racket (allerdings nicht in BSL) solche Listen; dort heißen sie *Vektoren*.

Der Reiz der rekursiven Formulierung liegt darin, dass man keinerlei zusätzliche Unterstützung für Listen in der Programmiersprache benötigt. Wir haben ja oben gesehen, dass wir uns die Art von Listen, die von BSL direkt unterstützt werden, auch selber programmieren können. Es ist einfach "yet another recursive datatype", und alle Konstrukte, Entwurfsrezepte und so weiter funktionieren nahtlos auch für Listen. Wir brauchen keine speziellen Schleifen oder andere Spezialkonstrukte um Listen zu verarbeiten, sondern machen einfach das, was wir auch bei jedem anderen rekursiven Datentyp machen. Das ist bei fest eingebauten Index-basierten Listen anders; es gibt viele Beispiele für Sprachkonstrukte, die für "normale Werte" funktionieren, aber nicht für Listen, und umgekehrt.

BSL und Racket stehen in der Tradition der Programmiersprache *Scheme*. Die Philosophie, die im ersten Satz der Sprachspezifikation von Scheme (`http://www.r6rs.org`) zu finden ist, ist damit auch für BSL und Racket gültig:

Programming languages should be designed not by piling feature on top of feature, but by removing the weaknesses and restrictions that make additional features appear necessary. Scheme demonstrates that a very small number of rules for forming expressions, with no restrictions on how they are composed, suffice to form a practical and efficient programming language that is flexible enough to support most of the major programming paradigms in use today.

Die Behandlung von Listen als rekursive Datentypen ist ein Beispiel für diese Philosophie.

Manche Programmieranfänger finden es nicht intuitiv, Listen und Funktionen darauf rekursiv zu formulieren. Aber ist es nicht besser, ein Universalwerkzeug zu verwenden, welches in sehr vielen Situationen verwendbar ist, als ein Spezialwerkzeug, das in nichts besser ist als das Universalwerkzeug und nur in einer Situation anwendbar ist?

9.3.6 Natürliche Zahlen als rekursive Datenstruktur

Es gibt in BSL nicht nur Funktionen, die Listen konsumieren, sondern auch solche, die Listen produzieren. Eine davon ist `make-list`. Hier ein Beispiel:

```
> (make-list 4 "abc")
'("abc" "abc" "abc" "abc")
```

Die `make-list` Funktion konsumiert also eine natürliche Zahl n und einen Wert und produziert eine Liste mit n Wiederholungen des Werts. Obwohl diese Funktion also nur atomare Daten konsumiert, produziert sie ein beliebig großes Resultat. Wie ist das möglich?

Eine erleuchtende Antwort darauf ist, dass man auch natürliche Zahlen als Instanzen eines rekursiven Datentyps ansehen kann. Hier ist eine mögliche Definition:

Auch in der Mathematik werden natürliche Zahlen ähnlich rekursiv definiert. Recherchieren sie, was die *Peano-Axiome* sind.

```
; A Nat (Natural Number) is one of:
; - 0
; - (add1 Nat)
```

Beispielsweise können wir die Zahl 3 repräsentieren als (add1 (add1 (add1 0))). Die add1 Funktion hat also eine Rolle ähnlich zu den Konstruktorfunktionen bei Strukturen. Die Rolle der Selektorfunktion wird durch die Funktion sub1 übernommen. Das Prädikat für die erste Alternative von Nat ist zero?; das Prädikat für die zweite Alternative heißt positive?.

Mit dieser Sichtweise sind wir nun in der Lage, mit unserem Standard-Entwurfsrezept Funktionen wie make-list selber zu definieren. Nennen wir unsere Variante von make-list iterate-value. Hier ist die Spezifikation:

```
; [X] Nat X -> (List-of X)
; creates a list with n occurences of x
(check-expect (iterate-value 3 "abc") (list "abc" "abc" "abc"))
(define (iterate-value n x) ...)
```

Gemäß unseres Entwurfsrezepts für rekursive Datentypen erhalten wir folgendes Template:

```
(define (iterate-value n x)
  (cond [(zero? n) ...]
        [(positive? n) ... (iterate-value (sub1 n) ...)...]))
```

Dieses Template zu vervollständigen ist nun nur noch ein kleiner Schritt:

```
(define (iterate-value n x)
  (cond [(zero? n) empty]
        [(positive? n) (cons x (iterate-value (sub1 n) x))]))
```

9.4 Mehrere rekursive Datentypen gleichzeitig

Ein schwierigerer Fall ist es, wenn mehrere Parameter einer Funktion einen rekursiven Datentyp haben. In diesem Fall ist es meist sinnvoll, einen dieser Parameter zu bevorzugen und zu ignorieren, dass andere Parameter ebenfalls rekursiv sind. Welcher Parameter bevorzugt werden sollte, ergibt sich aus der Fragestellung, wie sie die Eingabe der Funktion am sinnvollsten zerlegen können, so dass Sie aus dem Ergebnis des rekursiven Aufrufs und den anderen Parametern am einfachsten das Gesamtergebnis berechnen können.

Betrachten wir als Beispiel eine Funktion zur Konkatenation von zwei Listen. Diese Funktion ist unter dem Namen append bereits eingebaut, aber wir bauen sie mal nach. Wir haben zwei Parameter, die beide rekursive Datentypen haben. Eine Möglichkeit wäre, den ersten Parameter zu bevorzugen. Damit erhalten wir folgendes Template:

```
; [X] (list-of X) (list-of X) -> (list-of X)
; concatenates l1 and l2
(check-expect (lst-append (list 1 2) (list 3 4)) (list 1 2 3 4))
(define (lst-append l1 l2)
  (cond [(empty? l1) ...l2...]
        [(cons? l1) ... (first l1) ... (lst-
append (rest l1) ...)]))
```

Tatsächlich ist es in diesem Fall leicht, das Template zu vervollständigen. Wenn wir das Ergebnis von `(lst-append (rest l1) l2)` haben, so müssen wir nur noch `(first l1)` vorne anhängen:

```
(define (lst-append l1 l2)
  (cond [(empty? l1) l2]
        [(cons? l1) (cons (first l1) (lst-
append (rest l1) l2))]))
```

Spielen wir jetzt einmal die zweite Möglichkeit durch: Wir bevorzugen den zweiten Parameter. Damit ergibt sich folgendes Template:

```
(define (lst-append l1 l2)
  (cond [(empty? l2) ...l1...]
        [(cons? l2) ... (first l2) ... (lst-
append .. (rest l2))]))
```

Der Basisfall ist zwar trivial, aber im rekursiven Fall ist es nicht offensichtlich, wie wir das Template vervollständigen können. Betrachten wir beispielsweise den rekursiven Aufruf `(lst-append l1 (rest l2))`, so können wir überlegen, dass uns dieses Ergebnis überhaupt nicht weiterhilft, denn wir müssten ja irgendwo in der Mitte (aber wo genau ist nicht klar) des Ergebnisses noch `(first l2)` einfügen.

9.5 Entwurfsrezept für Funktionen mit rekursiven Datentypen

Wir haben gesehen, wie wir mit Hilfe eines Entwurfsrezepts einfache Funktionen (§3.3.3 “Entwurfsrezept zur Funktionsdefinition”), Funktionen mit Summentypen (§5.3.1 “Entwurf mit Summentypen”), Funktionen mit Produkttypen (§6.7 “Erweiterung des Entwurfsrezepts”) und Funktionen mit algebraischen Datentypen (§7.2 “Programmentwurf mit ADTs”) entwerfen.

An dieser Stelle fassen wir zusammen, wie wir das Entwurfsrezept erweitern, um mit Daten beliebiger Größe umzugehen.

1. Wenn es in der Problemdomäne Informationen unbegrenzter Größe gibt, benötigen Sie eine selbstreferenzierende Datendefinition. Damit eine selbstreferenzierende Datendefinition gültig ist, muss sie drei Bedingungen erfüllen: 1) Der Datentyp ist ein Summentyp. 2) Es muss mindestens zwei Alternativen geben. 3) Mindestens eine der Alternativen referenziert nicht den gerade definierten Datentyp, ist also nicht rekursiv.

   Sie sollten für rekursive Datentypen Datenbeispiele angeben, um zu validieren, dass Ihre Definition sinnvoll ist. Wenn es nicht offensichtlich ist, wie Sie ihre Datenbeispiele beliebig groß machen, stimmt vermutlich etwas nicht.

2. Beim zweiten Schritt ändert sich nichts: Sie benötigen wie immer eine Signatur, eine Aufgabenbeschreibung und eine Dummy-Implementierung.

3. Bei selbstreferenzierenden Datentypen ist es nicht mehr möglich, für jede Alternative einen Testfall anzugeben (weil es, wenn man in die Tiefe geht, unendlich viele Alternativen gibt). Die Testfälle sollten auf jeden Fall alle Teile der Funktion abdecken. Versuchen Sie weiterhin, kritische Grenzfälle zu identifizieren und zu testen.

4. Rekursive Datentypen sind algebraische Datentypen, daher kann für den Entwurf des Templates die Methodik aus §7.2 “Programmentwurf mit ADTs” angewendet werden. Die wichtigste Ergänzung zu dem Entwurfsrezept betrifft den Fall, dass eine Alternative implementiert werden muss, die selbstreferenzierend ist. Statt wie sonst eine neue Hilfsfunktion im Template zu verwenden, wird in diesem Fall ein rekursiver Aufruf der Funktion, die Sie gerade implementieren, ins Template aufgenommen. Als Argument des rekursiven Aufrufs wird der Aufruf des Selektors, der den zur Datenrekursion gehörigen Wert extrahiert, ins Template mit aufgenommen. Wenn Sie beispielsweise eine Funktion `(define (f a-list-of-strings ...) ...)` auf Listen definieren, so sollte im `cons?` Fall der Funktion der Aufruf `(f (rest a-list-of-strings) ...)` stehen.

5. Beim Entwurf des Funktionsbodies starten wir mit den Fällen der Funktion, die nicht rekursiv sind. Diese Fälle nennt man auch die *Basisfälle*, in Analogie zu Basisfällen bei Beweisen per Induktion. Die nicht-rekursiven Fälle sind typischerweise einfach und sollten sich direkt aus den Testfällen ergeben.

   In den rekursiven Fällen müssen wir uns überlegen, was der rekursive Aufruf bedeutet. Hierzu nehmen wir an, dass der rekursive Aufruf die Funktion korrekt berechnet, so wie wir es in der Aufgabenbeschreibung in Schritt 2 festgelegt haben. Mit diesem Wissen müssen wir nun nur noch die zur Verfügung stehenden Werte zur Lösung zusammenbauen.

6. Testen Sie wie üblich, ob ihre Funktion wie gewünscht funktioniert und prüfen Sie, ob die Tests alle interessanten Fälle abdecken.

7. Für Programme mit rekursiven Datentypen gibt es einige neue Refactorings, die möglicherweise sinnvoll sind. Überprüfen Sie, ob ihr Programm Datentypen enthält, die nicht rekursiv sind, aber die durch einen rekursiven Datentyp vereinfacht werden könnten. Enthält ihr Programm beispielsweise separate Datentypen für Angestellter, Gruppenleiter, Abteilungsleiter, Bereichsleiter und so weiter, so könnte dies durch einen rekursiven Datentyp, mit dem beliebig tiefe Managementhierarchien modelliert werden können, vereinfacht werden.

9.6 Refactoring von rekursiven Datentypen

Bezüglich der Datentyp-Refactorings aus §7.3 “Refactoring von algebraischen Datentypen” ergeben sich neue Typisomorphien durch "inlining" bzw. Expansion von rekursiven Datendefinitionen. Beispielsweise ist in folgendem Beispiel `list-of-number` isomorph zu `list-of-number2`, denn letztere Definition ergibt sich aus der ersten indem man die Rekursion einmal expandiert.

```
; A list-of-number is either:
; - empty
; - (cons Number list-of-number)

(define-struct Number-and-Number-List (num numlist))
; A list-of-number2 is either:
; - empty
; - (make-Number-and-Number-List Number list-of-number)
```

Versuchen Sie, Definitionen wie `list-of-number2` zu vermeiden, denn Funktionen, die darauf definiert sind, sind komplexer als solche, die `list-of-number` verwenden.

Wenn wir die Notation aus §7.3 "Refactoring von algebraischen Datentypen" verwenden, so können wir rekursive Datentypen als Funktionen modellieren, wobei der Funktionsparameter das rekursive Vorkommen des Datentyps modelliert. Beispielsweise kann der Datentyp der Listen von Zahlen durch die Funktion `F (X) = (+ Empty (* Number X))` modelliert werden[3]. Das schöne an dieser Notation ist, dass man sehr leicht definieren kann, wann rekursive Datentypen isormoph sind. Die Expansion eines rekursiven Datentyps ergibt sich daraus, den Funktionsparameter `X` durch die rechte Seite der Definition zu ersetzen, also in dem Beispiel `F (X) = (+ Empty (* Number (+ Empty (* Number X))))`. Inlining ist die umgekehrte Operation. Die Rechtfertigung für diese Operationen ergibt sich daraus, dass man rekursive Datentypen als kleinsten Fixpunkt solcher Funktoren verstehen kann. Beispielsweise ist der kleinste Fixpunkt von `F (X) = (+ Empty (* Number X))` die unendliche Summe `(+ Empty (* Number Empty) (* Number Number Empty) (* Number Number Number Empty) ...)`. Der kleinste Fixpunkt ändert sich durch Expansion oder Inlining nicht, daher sind solche Datentypen isomorph.

9.7 Programmäquivalenz und Induktionsbeweise

Betrachten Sie die folgenden beiden Funktionen:

```
; FamilyTree -> Number
; computes the number of known ancestors of p
(check-expect (numKnownAncestors HEINZ) 5)
(define (numKnownAncestors p)
  (cond [(person? p) (+ 1
                        (numKnownAncestors (person-father p))
                        (numKnownAncestors (person-
mother p)))]
        [else 0]))

; FamilyTree -> Number
; computes the number of unknown ancestors of p
(check-expect (numUnknownAncestors HEINZ) 6)
```

[3] Solche Funktionen nennt man auch *Funktoren* und sie sind in der universellen Algebra als *F-Algebras* von wichtiger Bedeutung.

```
(define (numUnknownAncestors p)
  (cond [(person? p) (+ (numUnknownAncestors (person-
father p))
                        (numUnknownAncestors (person-
mother p)))]
        [else 1]))
```

Die Tests suggerieren, dass für alle Personen `p` folgende Äquivalenz gilt: `(+ (numKnownAncestors p) 1)` ≡ `(numUnknownAncestors p)`. Aber wie können wir zeigen, dass diese Eigenschaft tatsächlich stimmt?

Das Schliessen durch Gleichungen, wie wir es in §8.12 "Refactoring von Ausdrücken und Schliessen durch Gleichungen" kennengelernt haben, reicht hierzu alleine nicht aus. Dadurch, dass die Funktionen rekursiv sind, können wir durch *EFUN* immer größere Terme erzeugen, aber wir kommen niemals von `numKnownAncestors` zu `numUnknownAncestors`.

Bei strukturell rekursiven Funktionen auf rekursiven Datentypen können wir jedoch ein weiteres, sehr mächtiges Beweisprinzip verwenden, nämlich das Prinzip der *Induktion*. Betrachten Sie nochmal die Mengenkonstruktion der ft_i aus §9.1 "Rekursive Datentypen". Wir wissen, dass der Typ FamilyTree die Vereinigung aller ft_i ist. Desweiteren wissen wir, dass, wenn `p` in ft_{i+1} ist, dann ist `(person-father p)` und `(person-mother p)` in ft_i. Dies rechtfertigt die Verwendung des Beweisprinzips der Induktion: Wir zeigen die gewünschte Äquivalenz für den Basisfall i = 1, also `p` = `#false`. Dann zeigen wir die Äquivalenz für den Fall i = n+1, unter der Annahme, dass die Äquivalenz bereits für i = n gilt. Anders ausgedrückt zeigen wir für die Äquivalenz für den Fall `p` = `(make-person n p1 p2)` unter der Annahme, dass die Äquivalenz für `p1` und `p2` gilt.

Typischerweise läßt man bei dieser Art von Induktionsbeweisen die Mengenkonstruktion mit ihren Indizes weg und "übersetzt" die Indizes direkt in die Datentyp-Notation. Dies bedeutet, dass man die gewünschte Aussage zunächst für die nichtrekursiven Fälle des Datentyps zeigt, und dann im Induktionsschritt die Aussage für die rekursiven Fälle zeigt unter der Annahme, dass die Aussage für die Unterkomponenten des Datentyps bereits gilt. Diese Art des Induktionsbeweises nennt man auch *strukturelle Induktion*.

Wir wollen beweisen: `(+ (numKnownAncestors p) 1)` ≡ `(numUnknownAncestors p)` für alle Personen `p`. Betrachten wir zunächst den Basisfall `p` = `#false`.

Dann können wir schliessen:

`(+ (numKnownAncestors #false) 1)`
≡ (gemäß *EFUN* und *EKONG*)
`(+ (cond [(person? #false) ...] [else 0]) 1)`
≡ (gemäß *STRUCT-predfalse* und *EKONG*)
`(+ (cond [#false ...] [else 0]) 1)`
≡ (gemäß *COND-False* und *EKONG*)
`(+ (cond [else 0]) 1)`
≡ (gemäß *COND-True* und *EKONG*)
`(+ 0 1)`

≡ (gemäß *PRIM*)
`1`

Unter Nutzung von *ETRANS* können wir damit `(+ (numKnownAncestors #false) 1)` ≡ `1` schliessen. Auf die gleiche Weise können wir schliessen: `(numUnknownAncestors #false)` ≡ `1`. Damit haben wir den Basisfall gezeigt.

Für den Induktionsschritt müssen wir die Äquivalenz für `p` = `(make-person n p1 p2)` zeigen und dürfen hierbei verwenden, dass die Aussage für `p1` und `p2` gilt, also `(+ (numKnownAncestors p1) 1)` ≡ `(numUnknownAncestors p1)` und `(+ (numKnownAncestors p2) 1)` ≡ `(numUnknownAncestors p2)`.

Wir können nun wie folgt schliessen:

```
(+ (numKnownAncestors (make-person n p1 p2)) 1)
```

≡ (gemäß *EFUN* und *EKONG*)

```
(+ (cond [(person? (make-person n p1 p2))
          (+ 1
            (numKnownAncestors (person-father (make-
person n p1 p2)))
            (numKnownAncestors (person-mother (make-
person n p1 p2))))]
         [else 0])
   1)
```

≡ (gemäß *STRUCT-predtrue* und *EKONG*)

```
(+ (cond [#true
          (+ 1
             (numKnownAncestors (person-father (make-
person n p1 p2)))
             (numKnownAncestors (person-mother (make-
person n p1 p2))))]
         [else 0])
   1)
```

≡ (gemäß *COND-True* und *EKONG*)

```
(+ (+ 1
     (numKnownAncestors (person-father (make-person n p1 p2)))
     (numKnownAncestors (person-mother (make-
person n p1 p2))))
   1)
```

≡ (gemäß *STRUCT-select* und *EKONG*)

```
(+ (+ 1
     (numKnownAncestors p1)
     (numKnownAncestors p2))
   1)
```

≡ (gemäß *EPRIM*)

```
(+
 (+ (numKnownAncestors p1) 1)
 (+ (numKnownAncestors p2) 1))
```

≡ (gemäß Induktionsannahme und *EKONG*)

```
(+
 (numUnknownAncestors p1)
 (numUnknownAncestors p2))
```

≡ (gemäß *STRUCT-select* und *EKOMM* und *EKONG*)

```
(+
 (numUnknownAncestors (person-father (make-person n p1 p2)))
 (numUnknownAncestors (person-mother (make-person n p1 p2))))
```

≡ (gemäß *EFUN* und *EKOMM*)

```
(numUnknownAncestors (make-person n p1 p2))
```

Damit haben wir (unter Nutzung von *ETRANS*) die Äquivalenz bewiesen. Dieser Beweis ist sehr ausführlich und kleinschrittig. Wenn Sie geübter im Nutzen von Programmäquivalenzen sind, werden ihre Beweise großschrittiger und damit auch kompakter.

Die gleiche Beweismethodik läßt sich für alle rekursiven Datentypen anwenden. Insbesondere läßt sie sich auch für Listen anwenden.

10 Pattern Matching

Viele Funktionen konsumieren Daten, die einen Summentyp oder einen algebraischen Datentyp (also eine Mischung aus Summen- und Produkttypen (§7 “Datendefinition durch Alternativen und Zerlegung: Algebraische Datentypen”) mit einer Summe "ganz oben" haben.

Häufig (und gemäß unseres Entwurfsrezepts) sehen solche Funktionen so aus, dass zunächst einmal unterschieden wird, welche Alternative gerade vorliegt, und dann wird (ggf. in Hilfsfunktionen) auf die Komponenten des in der Alternative vorliegenden Produkttypen zugegriffen.

Beispielsweise haben Funktionen, die Listen verarbeiten, in der Regel diese Struktur:

```
(define (f l)
  (cond [(cons? l) (... (first l) ... (f (rest l))...)]
        [(empty? l) ...]))
```

Mit *Pattern Matching* können solche Funktionen mit deutlich reduziertem Aufwand definiert werden. Pattern Matching hat zwei Facetten: 1) Es definiert implizit eine Bedingung, analog zu den Bedingungen in den `cond` Klauseln oben. 2) Es definiert Namen, die statt der Projektionen (`(first l)` und `(rest l)` im Beispiel) verwendet werden können.

Pattern Matching kann auf allen Arten von Summentypen verwendet werden. Insbesondere ist es nicht auf rekursive Typen wie Listen oder Bäume beschränkt.

10.1 Pattern Matching am Beispiel

Um Unterstützung für Pattern Matching zu bekommen, verwenden wir das Teachpack `2htdp/abstraction`. Fügen Sie an Anfang ihres Programms daher diese Anweisung ein:

```
(require 2htdp/abstraction)
```

Das folgende Beispiel zeigt, wie das `match` Konstrukt verwendet werden kann.

```
(define (f x)
    (match x
      [7 8]
      ["hey" "joe"]
      [(list 1 y 3) y]
      [(cons a (list 5 6)) (add1 a)]
      [(posn 5 5) 42]
      [(posn y y) y]
      [(posn y z) (+ y z)]
      [(cons (posn 1 z) y) z]
      [(? cons?) "nicht-leere Liste"]))
```

Hier sind einige Beispiele, die das Verhalten des `match` Konstrukts illustrieren.

```
> (f 7)
8
> (f "hey")
"joe"
> (f (list 1 2 3))
2
> (f (list 4 5 6))
5
> (f (make-posn 5 5))
42
> (f (make-posn 6 6))
6
> (f (make-posn 5 6))
11
> (f (list (make-posn 1 6) 7))
6
> (f (list 99 88))
"nicht-leere Liste"
> (f 42)
match: no matching clause for 42
```

Jede Klausel in einem `match` Ausdruck beginnt mit einem Pattern. Ein Pattern kann ein Literal sein, wie in den ersten beiden Klauseln (`7` und `"hey"`). In diesem Fall ist das Pattern lediglich eine implizite Bedingung: Wenn der Wert, der gematcht wird (im Beispiel `x`), gleich dem Literal ist, dann ist der Wert des Gesamtausdrucks der der rechten Seite der Klausel (analog zu `cond`).

Interessant wird Pattern Matching dadurch, dass auch auf Listen und andere algebraische Datentypen "gematcht" werden kann. In den Pattern dürfen Namen vorkommen (wie das `y` in `(list 1 y 3)` ; diese Variablen sind im Unterschied zu Strukturnamen oder Literalen keine Bedingungen, sondern sie dienen zur Bindung der Namen an den entsprechenden Teil der Struktur.

Allerdings können Namen zur Bedingung werden, wenn sie mehrfach im Pattern vorkommen. Im Beispiel oben ist dies der Fall im Pattern `(posn y y)`. Dieses Pattern matcht nur dann, wenn `x` eine `posn` ist und beide Komponenten den gleichen Wert haben.

Falls mehrere Pattern gleichzeitig matchen, so "gewinnt" stets das erste Pattern, welches passt (analog dazu wie auch bei `cond` stets die erste Klausel, deren Kondition `true` ergibt, "gewinnt". Daher ergibt beispielsweise `(f (make-posn 5 5))` im Beispiel das Ergebnis `42` und nicht etwa `5` oder `10`.

Das vorletzte Pattern, `(cons (posn 1 z) y)`, illustriert, dass Patterns beliebig tief verschachtelt werden können.

Im letzten Pattern, `(? list?)`, sehen wir, dass auch Prädikatsfunktionen von Listen und Strukturen verwendet werden können, um zu überprüfen, was für eine Art von Wert wir gerade haben. Diese Art von Pattern bietet sich an, wenn man nur wissen möchte, ob der Wert, auf dem wir matchen, zum Beispiel eine Liste oder eine `posn` ist.

Pattern Matching ist in vielen Fällen eine sinnvolle Alternative zum Einsatz von

`cond` Ausdrücken. Beispielsweise können wir mittels Pattern Matching die Funktion

```
(define (person-has-ancestor p a)
  (cond [(person? p)
         (or
          (string=? (person-name p) a)
          (person-has-ancestor (person-father p) a)
          (person-has-ancestor (person-mother p) a))]
        [else #false]))
```

aus §9.1 “Rekursive Datentypen” umschreiben zu:

```
(define (person-has-ancestor p a)
  (match p
    [(person name father mother)
         (or
          (string=? name a)
          (person-has-ancestor father a)
          (person-has-ancestor mother a))]
    [else #false]))
```

10.2 Pattern Matching allgemein

Wir betrachten die Syntax, Bedeutung und Reduktion von Pattern Matching.

10.2.1 Syntax von Pattern Matching

Um die syntaktische Struktur der `match` Ausdrücke zu definieren, erweitern wir die Grammatik für Ausdrücke aus §8.3 “Syntax von BSL” wie folgt:

⟨*e*⟩ ::= ...
| `(` `match` ⟨*e*⟩ {`[` ⟨*pattern*⟩ ⟨*e*⟩ `]`}$^+$ `)`

⟨*pattern*⟩ ::= ⟨*literal-constant*⟩
| ⟨*name*⟩
| `(` ⟨*name*⟩ ⟨*pattern*⟩* `)`
| `(` `?` ⟨*name*⟩`?` `)`

⟨*literal-constant*⟩ ::= ⟨*number*⟩
| ⟨*boolean*⟩
| ⟨*string*⟩

10.2.2 Bedeutung von Pattern Matching

Falls man einen Ausdruck der Form `(match v [p-1 e-1] ... [p-n e-n])` hat, so kann man Pattern Matching verstehen als die Aufgabe, ein minimales `i` zu finden, so dass `p-i` auf `v` "matcht". Aber was bedeutet das genau?

Wir können Matching als eine Funktion definieren, die ein Pattern und einen Wert als Eingabe erhält und entweder "no match" oder eine *Substitution* zurückgibt. Eine Substitution ist ein Mapping $[x_1 := v_1, \ldots, x_n := v_n]$ von Namen auf Werte. Eine

Substitution kann auf einen Ausdruck angewendet werden. Dies bedeutet, dass alle Namen in dem Ausdruck, die in der Substitution auf einen Wert abgebildet werden, durch diesen Wert ersetzt werden. Wenn e ein Ausdruck ist und σ eine Substitution, so schreiben wir $e\sigma$ für die Anwendung von σ auf e. Beispielsweise für $\sigma = [x := 1, y := 2]$ und $e = (+\ \texttt{x}\ \texttt{y}\ \texttt{z})$, ist

$$e\sigma = (+\ \texttt{x}\ \texttt{y}\ \texttt{z})\,[x := 1, y := 2] = (+\ \texttt{1}\ \texttt{2}\ \texttt{z})$$

Das Matching eines Werts auf ein Pattern ist nun wie folgt definiert:

$$\begin{aligned}
match(v, v) &= [] \\
match((name\ p_1 \ldots p_n), <\texttt{make}-name\ v_1 \ldots v_n>) &= match(p_1, v_1) \oplus \ldots \oplus match(p_n, v_n) \\
match((\texttt{cons}\ p_1\ p_2), <\texttt{cons}\ v_1\ v_2>) &= match(p_1, v_1) \oplus match(p_2, v_2) \\
match((?\ \texttt{name}?), <\texttt{make}-name \ldots >) &= [] \\
match(x, v) &= [x := v] \\
match(\ldots, \ldots) &= no\ match \text{ in allen anderen Fällen}
\end{aligned}$$

Hierbei ist $\oplus$ ein Operator, der Substitutionen kombiniert. Das Ergebnis von $\sigma_1 \oplus \sigma_2$ ist "no match", falls σ_1 oder σ_2 "no match" sind oder σ_1 und σ_2 beide ein Mapping für den gleichen Namen definieren aber diese auf unterschiedliche Werte abgebildet werden.

$$\begin{aligned}
[x_1 := v_1, \ldots, x_k := v_k] \oplus [x_{k+1} := v_{k+1}, \ldots, x_n := v_n] &= [x_1 := v_1, \ldots, x_n := v_n] \\
&\text{falls für alle } i, j \text{ gilt:} x_i = x_j \Rightarrow v_i = v_j.
\end{aligned}$$

Beispiele:

$$match((\texttt{make} - \texttt{posn}\ \texttt{x}\ \texttt{y}), <\texttt{make} - \texttt{posn}\ \texttt{3}\ \texttt{4}>) = [x := 3, y := 4]$$

$$match((\texttt{make} - \texttt{posn}\ \texttt{3}\ \texttt{y}), <\texttt{make} - \texttt{posn}\ \texttt{3}\ \texttt{4}>) = [y := 4]$$

$$match(\texttt{x}, <\texttt{make} - \texttt{posn}\ \texttt{3}\ \texttt{4}>) = [x := <\texttt{make} - \texttt{posn}\ \texttt{3}\ \texttt{4}>]$$

$$match((\texttt{cons}\ (\texttt{make} - \texttt{posn}\ \texttt{x}\ \texttt{3})\ \texttt{y}), <\texttt{cons}\ <\texttt{make} - \texttt{posn}\ \texttt{3}\ \texttt{3}>\ \texttt{empty}>) = [x := \texttt{3}, y := empty]$$

$$match((\texttt{cons}\ (\texttt{make} - \texttt{posn}\ \texttt{x}\ \texttt{x})\ \texttt{y}), <\texttt{cons}\ <\texttt{make} - \texttt{posn}\ \texttt{3}\ \texttt{4}>\ \texttt{empty}>) = no\ match$$

10.2.3 Reduktion von Pattern Matching

Wir erweitern die Grammatik des Auswertungskontextes so, dass der Ausdruck, auf dem gemacht wird, zu einem Wert reduziert werden kann. Alle anderen Unterausdrücke des `match` Ausdrucks werden nicht ausgewertet.

⟨*E*⟩ ::= ...
| (match ⟨*E*⟩ {[⟨*pattern*⟩ ⟨*e*⟩]}$^+$)

In der Reduktionsrelation verwenden wir nun die *match* Funktion von oben um zu entscheiden, ob ein Pattern matcht und um ggf. die durch das Pattern gebundenen Namen in dem dazugehörigen Ausdruck durch die entsprechenden Werte zu ersetzen.

(MATCH-YES): Falls in einem Ausdruck `(match v [p-1 e-1] ... [p-n e-n])` gilt: match(`p-1`,`v`) = σ und `e-1` σ = `e`, dann `(match v [p-1 e-1] ... [p-n e-n])` $\rightarrow$ `e`.

(MATCH-NO): Falls hingegen match(`p-1`,`v`) = "no match", so gilt: `(match v [p-1 e-1] ... [p-n e-n])` $\rightarrow$ `(match v [p-2 e-2] ... [p-n e-n])`.

Sind keine Patterns zum Matchen mehr übrig, so wird die Auswertung mit einem Laufzeitfehler abgebrochen, wie oben an dem `(f 42)` Beispiel illustriert. Dies wird in der Reduktionssemantik dadurch modelliert, dass die Auswertung "steckenbleibt", also nicht mehr weiter reduziert werden kann obwohl der Ausdruck noch kein Wert ist.

10.2.4 Pattern Matching Fallstricke

Es gibt einige Besonderheiten bgzl. des Verhaltens von Pattern Matching bei der Verwendung von Literalen, die möglicherweise zu Verwirrungen führen können.

Hier einige Beispiele:

```
> (match true [false false] [else 42])
#true
> (match true [#false false] [else 42])
42
> (match (list 1 2 3) [empty empty] [else 42])
'(1 2 3)
> (match (list 1 2 3) [(list) empty] [else 42])
42
```

Die ersten beiden Beispiele illustrieren, dass es wichtig ist, die boolschen Konstanten als `#true` und `#false` zu schreiben, wenn sie in Pattern vorkommen. Wenn man stattdessen `false` oder `true` schreibt, so werden diese als Namen interpretiert, die durch das Pattern Matching gebunden werden.

Die letzten beiden Beispiele zeigen, dass das gleiche Phänomen bei Listenliteralen auftritt. Schreiben Sie `(list)` und nicht `empty` wenn Sie auf die leere Liste matchen wollen.

11 Quote und Unquote

Listen spielen in funktionalen Sprachen eine wichtige Rolle, insbesondere in der Familie von Sprachen, die von LISP abstammen (wie Racket und BSL).

Wenn man viel mit Listen arbeitet, ist es wichtig, eine effiziente Notation dafür zu haben. Sie haben bereits die `list` Funktion kennengelernt, mit der man einfache Listen kompakt notieren kann.

Allerdings gibt es in BSL/ISL (und vielen anderen Sprachen) einen noch viel mächtigeren Mechanismus, nämlich `quote` und `unquote`. Diesen Mechanismus gibt es seit den 1950er Jahren in LISP, und noch heute eifern beispielsweise Template Sprachen wie Java Server Pages oder PHP diesem Vorbild nach.

Um mit `quote` und `unquote` zu arbeiten, ändern Sie bitte den Sprachlevel auf "Anfänger mit Listenabkürzungen" beziehungsweise "Beginning Student with List Abbreviations".

11.1 Quote

Das `quote` Konstrukt dient als kompakte Notation für große und verschachtelte Listen. Beispielsweise können wir mit der Notation `(quote (1 2 3))` die Liste `(cons 1 (cons 2 (cons 3 empty)))` erzeugen. Dies ist noch nicht besonders eindrucksvoll, denn der Effekt ist der gleiche wie

```
> (list 1 2 3)
'(1 2 3)
```

Zunächst mal gibt es eine Abkürzung für das Schlüsselwort `quote`, nämlich das Hochkomma, '.

```
> '(1 2 3)
'(1 2 3)
```

```
> '("a" "b" "c")
'("a" "b" "c")
```

```
> '(5 "xx")
'(5 "xx")
```

Bis jetzt sieht `quote` damit wie eine minimale Verbesserung der `list` Funktion aus. Dies ändert sich, wenn wir damit verschachtelte Listen, also Bäume, erzeugen.

```
> '(("a" 1) ("b" 2) ("c" 3))
'(("a" 1) ("b" 2) ("c" 3))
```

Wir können also mit `quote` auch sehr einfach verschachtelte Listen erzeugen, und zwar mit minimalem syntaktischem Aufwand.

Die Bedeutung von `quote` ist über eine rekursive syntaktische Transformation definiert.

- `'(e-1 ... e-n)` wird transformiert zu `(list 'e-1 ... 'e-n)`. Die Transformation wird rekursiv auf die erzeugten Unterausdrücke `'e-1` usw. angewendet.
- Wenn `l` ein Literal (eine Zahl, ein String, ein Wahrheitswert[4], oder ein Bild) ist, dann wird `'l` transformiert zu `l`.
- Wenn `n` ein Name/Bezeichner ist, dann wird `'n` transformiert zum *Symbol* `'n`.

Ignorieren Sie eine Sekunde die dritte Regel und betrachten wir den Ausdruck `'(1 (2 3))`. Gemäß der ersten Regel wird dieser Ausdruck im ersten Schritt transformiert zu `(list '1 '(2 3))`. Gemäß der zweiten Regel wird der Unterausdruck `'1` zu `1` und gemäß der Anwendung der ersten Regel wird aus dem Unterausdruck `'(2 3)` im nächsten Schritt `(list '2 '3)`. Gemäß der zweiten Regel wird dieser Ausdruck wiederum transformiert zu `(list 2 3)`. Insgesamt erhalten wir also das Ergebnis `(list 1 (list 2 3))`.

Sie sehen, dass man mit `quote` sehr effizient verschachtelte Listen (eine Form von Bäumen) erzeugen kann. Vielleicht fragen Sie sich, wieso wir nicht gleich von Anfang an `quote` verwendet haben. Der Grund dafür ist, dass diese bequemen Wege, Listen zu erzeugen, verbergen, welche Struktur Listen haben. Insbesondere sollten Sie beim Entwurf von Programmen (und der Anwendung des Entwurfsrezepts) stehts vor Augen haben, dass Listen aus `cons` und `empty` zusammengesetzt sind.

11.2 Symbole

Symbole sind eine Art von Werten die Sie bisher noch nicht kennen. Symbole dienen zur Repräsentation symbolischer Daten. Symbole sind verwandt mit Strings; statt durch Anführungszeichen vorne und hinten wie bei einem String, `"Dies ist ein String"`, werden Symbole durch ein einfaches Hochkomma gekennzeichnet: `'dies-ist-ein-Symbol`. Symbole haben die gleiche Syntax wie Namen/Bezeichner, daher sind beispielsweise Leerzeichen nicht erlaubt.

Im Unterschied zu Strings sind Symbole nicht dazu gedacht, Texte zu repräsentieren. Man kann beispielsweise nicht (direkt) Symbole konkatenieren. Es gibt nur eine wichtige Operation für Symbole, nämlich der Vergleich von Symbolen mittels `symbol=?`.

```
> (symbol=? 'x 'x)
#true


> (symbol=? 'x 'y)
#false
```

Symbole sind dafür gedacht, "symbolische Daten" zu repräsentieren. Das sind Daten, die "in der Realität" eine wichtige Bedeutung haben, aber die wir in Programm nur mit einem Symbol darstellen wollen. Ein Beispiel dafür sind Farben: `'red`,

[4]Wenn Sie boolsche Literale quoten wollen, müssen Sie die Syntax `#true` und `#false` für die Wahrheitswerte verwenden; bei `true` und `false` erhalten Sie ein Symbol.

`'green`, `'blue`. Es macht keinen Sinn, die Namen von Farben als Text zu betrachten. Wir wollen lediglich ein Symbol für jede Farbe und vergleichen können, ob eine Farbe beispielsweise `'red` ist (mit Hilfe von `symbol=?`).

11.3 Quasiquote und Unquote

Der `quote` Mechanismus birgt noch eine weitere Überraschung. Betrachten Sie das folgende Programm:

```
(define x 3)
(define y '(1 2 x 4))
```

Welchen Wert hat `y` nach Auswertung dieses Programms? Wenn Sie die Regeln oben anwenden, sehen Sie, dass nicht etwa `(list 1 2 3 4)` sondern `(list 1 2 'x 4)` herauskommt. Aus dem Bezeichner `x` wird also das *Symbol* `'x`.

Betrachten wir noch ein weiteres Beispiel:

```
> '(1 2 (+ 3 4))
'(1 2 (+ 3 4))
```

Wer das Ergebnis `(list 1 2 7)` erwartet hat, wird enttäuscht. Die Anwendung der Transformationsregeln ergibt das Ergebnis: `(list 1 2 (list '+ 3 4))`. Aus dem Bezeichner `+` wird das Symbol `'+`. Das Symbol `'+` hat keine direkte Beziehung zur Additionsfunktion, genau wie das Symbol `'x` in dem Beispiel oben keine direkte Beziehung zum Konstantennamen `x` hat.

Was ist aber, wenn Sie Teile der (verschachtelten) Liste doch berechnen wollen?

Betrachten wir als Beispiel die folgende Funktion:

```
; Number -> (List-of Number)
; given n, generates the list ((1 2) (m 4)) where m is n+1
(check-expect (some-list 2) (list (list 1 2) (list 3 4)))
(check-expect (some-list 11) (list (list 1 2) (list 12 4)))
(define (some-list n) ...)
```

Eine naive Implementation wäre:

```
(define (some-list n) '((1 2) ((+ n 1) 4)))
```

Aber natürlich funktioniert diese Funktion nicht wie gewünscht:

```
> (some-list 2)
'((1 2) ((+ n 1) 4))
```

Für solche Fälle bietet sich `quasiquote` an. Das `quasiquote` Konstrukt verhält sich zunächst mal wie `quote`,ausser dass es statt mit einem geraden Hochkomma mit einem schrägen Hochkomma abgekürzt wird:

```
> `(1 2 3)
'(1 2 3)
```

```
> `(a ("b" 5) 77)
'(a ("b" 5) 77)
```

Das besondere an `quasiquote` ist, dass man damit innerhalb eines gequoteten Bereichs zurückspringen kann in die Programmiersprache. Diese Möglichkeit nennt sich "unquote" und wird durch das `unquote` Konstrukt unterstützt. Auch `unquote` hat eine Abkürzung, nämlich das Komma-Zeichen.

```
> `(1 2 ,(+ 3 4))
'(1 2 7)
```

Mit Hilfe von `quasiquote` können wir nun auch unser Beispiel von oben korrekt implementieren.

```
(define (some-list-v2 n) `((1 2) (,(+ n 1) 4)))


> (some-list-v2 2)
'((1 2) (3 4))
```

Die Regeln zur Transformation von `quasiquote` sind genau wie die von `quote` mit einem zusätzlichen Fall: Wenn `quasiquote` auf ein `unquote` trifft, neutralisieren sich beide. Ein Ausdruck wie `` `,e `` wird also transformiert zu `e`.

11.4 S-Expressions

Betrachten Sie die `person-has-ancestor` Funktion aus §9.2 “Programmieren mit rekursiven Datentypen”. Eine ähnliche Funktion läßt sich auch für viele andere baumartig organisierte Datentypen definieren, beispielsweise solche zur Repräsentation von Ordnerhierarchien in Dateisystemen oder zur Repräsentation der Hierarchie innerhalb einer Firma.

Natürlich könnten wir neben `person-has-ancestor` nun auch noch `file-has-enclosing-directory` und `employee-has-manager` implementieren, aber diese hätten eine sehr ähnliche Struktur wie `person-has-ancestor`. Wir würden also gegen das DRY-Prinzip verstossen.

Es gibt eine ganze Reihe von Funktionen, die sich auf vielen baumartigen Datentypen definieren liessen: Die Tiefe eines Baumes berechnen, nach Vorkommen eines Strings suchen, alle "Knoten" des Baums finden, die ein Prädikat erfüllen, und so weiter.

Um solche Funktionen generisch (also einmal für alle Datentypen) definieren zu können, brauchen wir die Möglichkeit, über die genaue Struktur von Datentypen abstrahieren zu können. Dies funktioniert mit den "getypten" Datentypen, die wir bisher betrachtet haben, nicht.

Eine der großen Innovationen der Programmiersprache LISP war die Idee eines universellen Datenformats: Ein Format, mit dem beliebige strukturierte Daten repräsentiert werden können, und zwar in solch einer Weise, dass das Datenformat Teil der Daten ist und dementsprechend darüber abstrahiert werden können. Diese Idee wird

typischerweise alle paar Jahre wieder einmal neu erfunden; zur Zeit sind beispielsweise XML und JSON beliebte universelle Datenformate.

Der Mechanismus, den es dazu in LISP seit Ende der 1950er Jahre gibt, heißt *S-Expressions*. Was sind S-Expressions? Hier ist eine Datendefinition, die dies genau beschreibt:

```
; An S-Expression is one of:
; - a Number
; - a String
; - a Symbol
; - a Boolean
; - an Image
; - empty
; - a (list-of S-Expression)
```

Beispiele für S-Expressions sind: `(list 1 (list 'two 'three) "four")`, `"Hi"`. Dies sind keine S-Expressions: `(make-posn 1 2)`, `(list (make-student "a" "b" 1))`.

S-Expressions können als universelles Datenformat verwendet werden, indem die Strukturierung der Daten zum Teil der Daten gemacht wird. Statt `(make-posn 1 2)` kann man auch die S-Expression `'(posn 1 2)` oder `'(posn (x 1) (y 2))` verwenden; statt `(make-person "Heinz" (make-person "Horst" false false) (make-person "Hilde" false false))` kann man auch die S-Expression `'(person "Heinz" (person "Horst" #false #false) (person "Hilde" #false #false))` oder `'(person "Heinz" (father (person "Horst" (father #false) (mother #false)) (mother (person "Hilde" (father #false) (mother #f)))))` verwenden.

Der Vorteil der zweiten Variante ist, dass man beliebige strukturierte Daten auf diese Weise uniform ausdrücken kann und die Struktur selber Teil der Daten ist. Damit wird es möglich, sehr generische Funktionen zu definieren, die auf beliebigen strukturierten Daten funktionieren. Der Nachteil ist der, dass man Sicherheit und Typisierung verliert. Es ist schwierig, zu sagen, dass eine Funktion beispielsweise nur S-Expressions als Eingabe verarbeiten kann, die einen Stammbaum repräsentieren.

Der Quote-Operator hat die Eigenschaft, dass er stets S-Expressions erzeugt. Sie können sogar beliebige Definitionen oder Ausdrücke in BSL nehmen, einen Quote-Operator drumherumschreiben, und Sie erhalten eine S-Expression, die dieses Programm repräsentiert.

```
> (first '(define-struct student (firstname lastname matnr)))
'define-struct
```

Diese Eigenschaft, die manchmal *Homoikonizität* genannt, macht es besonders leicht, Programme als Daten zu repräsentieren und Programme zu schreiben, die die Repräsentation eines Programms als Eingabe bekommen oder als Ausgabe produzieren. In Scheme und (vollem) Racket gibt es sogar eine Funktion `eval`, die eine Repräsentation eines Ausdrucks als S-Expression als Eingabe bekommt und die diesen Ausdruck dann interpretiert und das Ergebnis zurückliefert. Beispielsweise

würde `(eval '(+ 1 1))` Ergebnis `2` liefern. Damit wird es möglich, Programme zur Laufzeit zu berechnen und dann direkt auszuführen - eine sehr mächtige aber auch sehr gefährliche Möglichkeit.

11.5 Anwendungsbeispiel: Dynamische Webseiten

Da S-Expressions ein universelles Datenformat sind, ist es einfach, andere Datenformate darin zu kodieren, zum Beispiel HTML (die Sprache in der die meisten Webseiten definiert werden).

Zusammen mit Quasiquote und Antiquote können S-Expressions dadurch leicht zur Erstellung von dynamischen Webseiten, bei denen die festen Teile als Template definiert werden, genutzt werden. Beispielsweise könnte eine einfache Funktion zur Erzeugung einer dynamischen Webseite wie folgt aussehen:

```
; String String -> S-Expression
; produce a (representation of) a web page with given author
and title
(define (my-first-web-page author title)
  `(html
     (head
       (title ,title)
       (meta ((http-equiv "content-type")
              (content "text-html"))))
     (body
       (h1 ,title)
       (p "I, " ,author ", made this page."))))
```

Die Funktion erzeugt die Repräsentation einer HTML-Seite, bei der die übergebenen Parameter an der gewünschten Stelle eingebaut werden. S-Expressions und Quasi/Antiquote führen zu einer besseren Lesbarkeit im Vergleich zur Variante der Funktion, die die Datenstruktur mit `cons` und `empty` oder `list` zusammenbaut. Die erzeugte S-Expression ist zwar noch kein HTML, aber sie kann leicht zu HTML umgewandelt werden. In Racket gibt es zu diesem Zweck beispielsweise die `xexpr->string` und `xexpr->xml` Funktion der XML Bibliothek.

```
> (require xml)


> (xexpr->string (my-first-web-page "Klaus Ostermann" "Meine
Homepage"))
"<html><head><title>Meine Homepage</title><meta
http-equiv=\"content-type\" content=\"text-
html\"/></head><body><h1>Meine Homepage</h1><p>I, Klaus Os-
termann, made this page.</p></body></html>"
```

Der durch `xexpr->string` erzeugte String ist gültiges HTML und könnte nun an einen Browser geschickt und dargestellt werden.

12 Sprachunterstützung für Datendefinitionen und Signaturen

In einem gewissen Sinne ist es egal, welche Programmiersprache man verwendet: Jede Berechnung und jeder Algorithmus lassen sich in jeder Programmiersprache ausdrücken. Man kann jede Programmiersprache und jedes Programmiersprachenkonzept in jeder anderen Programmiersprache simulieren.

Dies gilt zumindest für alle sogenannten "Turing-vollständigen" Sprachen. Fast alle gängigen Programmiersprachen sind Turing-vollständig.

Die Entwurfstechniken, die wir Ihnen beibringen, lassen sich in allen Programmiersprachen verwenden. Allerdings unterscheiden sich Programmiersprachen darin, wie gut die Entwurfstechniken durch die Sprache unterstützt werden. Wenn Sie beispielsweise in Assembler-Programmiersprachen programmieren, gibt es nichts, was unserem `define-struct` entspricht; ein Assembler Programmier muss sich daher selber überlegen, wie er mehrere Werte zusammenfasst und in einem linearen Speicher anordnet.

Auf der anderen Seite gibt es auch Programmiersprachen, in denen unsere Entwurfstechniken besser unterstützt werden als in BSL.

In diesem Kapitel wollen wir darüber reden, wie Sprachen eines unserer Kernkonzepte unterstützen können, nämlich das der Datendefinitionen und Signaturen. Datendefinitionen dienen zur Strukturierung und Klassifikation der Daten in unserem Programm. Signaturen dienen der Beschreibung der Erwartungen einer Funktion an ihre Argumente und der Beschreibung der Garantien bezüglich des Ergebnisses.

Um Sprachen und Sprachfeatures bezüglich ihrer Unterstützung für Datendefinitionen und Signaturen zu bewerten, benötigen wir Kriterien, die qualitativ diese Unterstützung messen. Wir wollen die folgenden Kriterien betrachten:

- Ist sichergestellt, dass alle Werte und Funktionen in einer Weise verwendet werden, die zur Information/Berechnung, die dieser Wert/Funktion repräsentiert, passt?
- Zu welchem Zeitpunkt werden Fehler gefunden? Im Allgemeinen möchte man, dass Fehler möglichst früh auftreten; am besten schon bevor das Programm startet, aber zumindest zu dem Zeitpunkt, an dem der Programmteil, der für den Fehler verantwortlich ist, ausgeführt wird. Am schlechtesten ist, wenn das Programm selber niemals einen Fehler meldet und stattdessen möglicherweise unbemerkt falsche Ergebnisse produziert.
- Wie modular sind die Fehlermeldungen? Wenn ein Fehler auftritt, so möchte man wissen, welcher Programmteil daran "schuld" ist. Dieses sogenannte "Blame Assignment" ist äußerst wichtig, um Fehler effektiv lokalisieren und beheben zu können. Bei Fehlern sollte es stets einen klar benennbaren Programmteil geben, der der Verursacher dieses Fehlers war.
- Wie ausdrucksstark ist die Signatursprache? Kann man die Bedingungen an Eingaben und Ausgaben präzise darin ausdrücken?
- Gibt es sinnvolle Programme, die nicht mehr ausgeführt werden können?

- Wird die Aufrechterhaltung der Konsistenz zwischen Datendefinitionen/Signaturen und dem Programmverhalten unterstützt?

Im folgenden wollen wir die wichtigsten Klassen von Programmiersprachen beschreiben, die sich in wichtigen Punkten bezüglich der oben angeführten Kriterien unterscheiden.

12.1 Ungetypte Sprachen

Ungetypte Sprachen zeichnen sich dadurch aus, dass jede Operation auf alle Arten von Daten angewendet werden kann, unabhängig davon ob es einen Sinn ergibt oder nicht. Beispielsweise ist es nicht sinnvoll, einen String und eine Zahl miteinander zu addieren.

Assembler-Sprachen sind typischerweise ungetypt. alle Arten von Daten als (32 oder 64 bit) Zahlen repräsentiert. Auch Strings, boolsche Werte, und alle anderen Daten werden durch solche Zahlenwerte repräsentiert. Addiert man nun zwei Werte, so werden die Zahlenwerte addiert, egal ob das aus Sicht dessen, was diese Werte repräsentieren, einen Sinn ergibt.

Bezüglich des ersten Punkts aus der Liste oben bieten ungetypte Sprachen daher keinerlei Unterstützung; es liegt vollständig in der Verantwortung des Programmierers, diese Eigenschaft sicherzustellen.

Fehler, also Verstöße gegen diese Eigenschaft, werden in Assembler-Programmen dementsprechend sehr spät gefunden, denn das Programm läuft ja einfach immer weiter, auch wenn die Daten, die gerade berechnet wurden, völlig unsinnig sind.

Da es keine von der Sprache unterstützten Signaturen oder Datendefinitionen gibt, gibt es auch keine Einschränkungen bezüglich der Ausdrucksstärke der Signaturen/Datendefinitionen und es gibt keine Einschränkungen der Art, dass bestimmte Programme nicht ausgeführt werden können. Allerdings gibt es auch keinerlei Unterstützung für die Aufrechterhaltung der Konsistenz; dies liegt allein in der Verantwortlichkeit des Programmierers.

12.2 Dynamisch getypte Sprachen

In dynamisch getypten Sprachen wird jedem Wert ein Typ zugeordnet und das Laufzeitsystem repräsentiert Werte so, dass der Typ eines Wertes während der Ausführung jederzeit abgefragt werden kann. Ein Typ ist daher eine Art Markierung für Werte, die darüber Auskunft gibt, was für eine Art von Wert es ist. Typischerweise gibt es (je nach Sprache unterschiedliche) fest eingebaute ("primitive") Typen sowie vom Benutzer zu definierende Typen. Die Beginning Student Language, in der wir bisher programmiert haben, ist eine dynamisch getypte Sprache. Fest eingebaute Typen sind beispielsweise Boolean (`boolean?`), Number (`number?`), String (`string?`) und Symbole (`symbol?`). Neue Typen können mittels `define-struct` definiert werden.

Die dynamischen Typen werden verwendet, um sicherzustellen, dass nur solche primitiven Operationen auf die Werte angewendet werden, die auch für diese Werte definiert sind. Wenn wir beispielsweise `(+ x y)` auswerten, so prüft das Laufzeitsystem, dass `x` und `y` auch tatsächlich Zahlen sind. Wenn `x` hingegen beispielsweise ein

boolscher Wert ist, so wird dieser, anders als bei ungetypten Sprachen, nicht einfach irgendwie als Zahl interpretiert.

Allerdings gilt diese Eigenschaft nicht für vom Programmierer selber definierte Funktionen. Es ist äußerst sinnvoll, jede Funktionsdefinition mit einer Signatur zu versehen, so wie wir es ja auch gemacht haben, doch es wird nicht geprüft, ob die Signatur auch von der Funktion und den Aufrufern der Funktion eingehalten wird.

Da die dynamischen Typen jedoch nicht alle Informationen umfassen, die wir in Datendefinitionen festhalten, kann es dennoch zu Fehlbenutzungen von Werten kommen. Beispielsweise macht es keinen Sinn, eine Temperatur und eine Länge zu addieren. Falls beide jedoch durch den Typ Number repräsentiert werden, kann das Laufzeitsystem diesen Fehler nicht feststellen.

Schauen wir uns mal an einigen Beispielen an, wie und wann in dynamische Typsystemen Typfehler auftauchen. Betrachten Sie folgende Funktion:

```
; Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or
empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Die Funktion gibt den Rest der Liste nach dem ersten Vorkommen des Elements `x` zurück (und `empty` falls das Element nicht vorkommt). Hier zwei Beispiele dazu:

```
> (rest-after 5 (list 1 2 3 4))
'()
> (rest-after 2 (list 1 2 3 4))
'(3 4)
```

Was passiert jedoch, wenn wir die Signatur verletzten?

```
> (rest-after 2 (list "eins" "zwei" "drei"))
=: expects a number as 2nd argument, given "eins"
```

Wir sehen, dass wir in diesem Fall einen Laufzeitfehler erhalten. Allerdings tritt nicht bei jeder Verletzung der Signatur (sofort) ein Laufzeitfehler auf:

```
> (rest-after 2 (list 1 2 "drei" "vier"))
'("drei" "vier")
```

In diesem Beispiel wird eine Liste übergeben, die nicht nur Zahlen enthält, aber da die hinteren Elemente der Liste nicht verwendet werden gibt es auch keinen Laufzeitfehler.

Betrachten wir jetzt einmal den Fall, dass nicht der Aufrufer der Funktion sondern die Funktion selber die Signatur verletzt. In diesem Beispiel gibt die Funktion einen String statt einer leeren Liste zurück, falls `x` in der Liste nicht vorkommt.

```
; Number (list-of Number) -> (list-of Number)
; returns the remainder of xs after first occurence of x, or
empty otherwise
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Diese Beispiele illustrieren, dass es keinen Fehler ergibt, wenn wir die Funktion mit einem `x` aufrufen, welches in der Liste enthalten ist.

```
> (rest-after 2 (list 1 2 3 4))
'(3 4)
```

Selbst wenn wir ein `x` auswählen, welches nicht in der Liste enthalten ist, ergibt die Ausführung keinen Laufzeitfehler.

```
> (rest-after 5 (list 1 2 3 4))
"not a list"
```

Erst wenn wir das Ergebnis verwenden und damit rechnen kommt es zu einem Laufzeitfehler.

```
> (cons 6 (rest-after 5 (list 1 2 3 4)))
cons: second argument must be a list, but received 6 and "not a list"
```

Allerdings ist es im Allgemeinen sehr schwer, herauszufinden, wer denn "Schuld" an diesem Fehler ist, denn die Stelle an der der Fehler auftritt ist möglicherweise weit von der Stelle entfernt, die den Fehler verursacht. Wenn Sie sich die Fehlermeldung anschauen, sehen Sie auch nichts, das darauf hindeutet, dass die Ursache des Fehlers in der Implementierung der `rest-after` Funktion zu finden ist. Daher sind Fehlermeldungen in dynamisch getypten Sprachen nicht sehr modular.

12.3 Dynamisch überprüfte Signaturen und Contracts

Um Fehler früher zu finden und Fehlermeldungen modularer zu machen, können Signaturen und Datendefinitionen auch als Programme definiert werden. Diese Programme können dann verwendet werden, um Signaturen zu überprüfen während das Programm läuft.

Eine Datendefinition wie:

```
; a list-of-numbers is either:
;  - empty
;  - (cons Number list-of numbers)
```

kann beispielsweise durch folgendes Programm repräsentiert werden:

```
; [X] (list-of X) -> Boolean
; checks whether xs contains only numbers
(define (list-of-numbers? xs)
  (if (empty? xs)
      #true
      (and (number? (first xs))
           (list-of-numbers? (rest xs)))))
```

Diese "ausführbaren" Datendefinitionen können dann, zusammen mit den vordefinierten Prädikaten wie `number?` verwendet werden, um eine dynamisch geprüfte Variante der `rest-after` Funktion zu definieren:

```
; Number (list-of Number) -> (list-of Number)
; dynamically checked version of rest-after
(define (rest-after/checked x xs)
  (if (number? x)
      (if (and (list? xs)
               (list-of-numbers? xs))
          (if (list-of-numbers? (rest-after x xs))
              (rest-after x xs)
              (error "function must return list-of-numbers"))
          (error "second arg must be list-of-numbers"))
      (error "first arg must be a number")))
```

Diese Funktion verhält sich genau wie `rest-after` sofern sich die Funktion und ihre Aufrufer an die Signatur halten:

```
> (rest-after/checked 2 (list 1 2 3 4))
'(3 4)
```

Im Fehlerfall gibt es jedoch viel früher eine Fehlermeldung und diese Fehlermeldung ist modular (sie tritt an der Stelle auf, die auch die Ursache des Fehlers ist).

```
> (rest-after/checked "x" (list 1 2 3 4))
first arg must be a number
```

Allerdings werden nun auch Programme mit einer Fehlermeldung abgebrochen, die, wie wir oben gesehen haben, vorher ohne Fehler durchgelaufen sind:

```
> (rest-after/checked 2 (list 1 2 "drei" 4))
second arg must be list-of-numbers
```

Dennoch ist es sinnvoll, diese Programme mit einem Fehler abzubrechen, denn im Allgemeinen wird früher oder später doch noch ein (dann nicht mehr modularer) Fehler auftreten. In jedem Fall ist ein Verstoss gegen die Signatur ein Hinweis auf einen Programmierfehler, unabhängig davon ob er tatsächlich letzten Endes zu einem Fehler führen würde.

Wie wir sehen, ist es allerdings aus Programmierersicht relativ mühselig und fehleranfällig, auf diese Art Signaturen und Datendefinitionen zu überprüfen. Deshalb gibt es einige Sprachen, die die dynamische Prüfung von Signaturen und Datendefinitionen direkt und komfortabel unterstützen.

Dies trifft beispielsweise auf die Sprache Racket zu, die auch von der DrRacket Umgebung unterstützt wird und die, bis auf kleine Abweichungen, die Beginning Student Language als Teilsprache unterstützt. In Racket können dynamische Prüfungen von Signaturen – in dem Kontext auch *Contracts* genannt – an der Grenze zwischen Modulen definiert werden. Module sind abgeschlossene Programmeinheiten, die in Racket meistens mit Dateien assoziiert sind, also jede Datei ist ein Modul.

Hier sehen Sie die Definition eines Moduls welches die `rest-after` Funktion von oben implementiert. In der `provide` Klausel des Moduls wird dieser Funktion ein *Contract*, also eine ausführbare Signatur, zugeordnet. Wir speichern das Modul in einer Datei "heinz.rkt", um zu illustrieren, dass vielleicht der Entwickler Heinz dieses Modul programmiert hat. Wie Sie sehen, hat Heinz den gleichen Fehler in die Implementierung eingebaut, den wir schon oben betrachtet haben.

`"heinz.rkt"`

```
#lang racket

(provide
 (contract-out
      [rest-after (-> number? (listof number?) (listof number?))]))
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Betrachten Sie nun ein Modul vom Entwickler Elke, welche diese Funktion benutzen möchte und daher das Modul von Heinz über eine `require` Klausel "importiert". Über diese Klausel wird deutlich gemacht, dass das Modul von Elke von dem Heinz Modul abhängt und dessen Funktionen verwenden möchte.

`"elke.rkt"`

```
#lang racket

(require "heinz.rkt")

(rest-after "x" (list 1 2 3 4))
```

Elke hat allerdings im Aufruf der Funktion gegen den Contract verstossen und hat als erstes Argument einen String übergeben. Wenn wir versuchen, dieses Programm auszuführen, so erhalten wir folgende Fehlermeldung:

```
rest-after: contract violation
 expected: number?
```

```
 given: "x"
 in: the 1st argument of
     (->
      number?
      (listof number?)
      (listof number?))
 contract from: /Users/klaus/heinz.rkt
 blaming: /Users/klaus/elke.rkt
  (assuming the contract is correct)
 at: /Users/klaus/heinz.rkt:3.24
```

Sie sehen, dass nicht nur der Aufruf der Funktion direkt als fehlerhaft erkannt wurde. Die Fehlermeldung sagt auch klar, wer an diesem Fehler Schuld ist, nämlich Elke.

Elke korrigiert also ihren Fehler. Nun kommt jedoch der Fehler, den Heinz in die Funktion eingebaut hat, zum Tragen. Dieser Fehler wird jedoch sofort gefunden und es wird korrekt Heinz die Schuld daran zugewiesen.

`"elke.rkt"`

```
 #lang racket

 (require "heinz.rkt")

 (rest-after 5 (list 1 2 3 4))
```

```
rest-after: broke its contract
  promised: "list?"
  produced: "not a list"
  in: the range of
      (->
       number?
       (listof number?)
       (listof number?))
  contract from: /Users/klaus/heinz.rkt
  blaming: /Users/klaus/heinz.rkt
   (assuming the contract is correct)
  at: /Users/klaus/heinz.rkt:3.24
```

Wie diese Beispiele illustrieren, ist der Hauptvorteil von dynamisch überprüften Signaturen und Contracts, das Fehler früher gefunden werden und die Fehlermeldungen modular sind und es bei Verletzungen einen klar benennbaren "Schuldigen" gibt. Wenngleich Fehler hierdurch früher gefunden werden, so werden die Fehler dennoch erst während der Programmausführung gefunden. Da es im Allgemeinen unendlich viele verschiedene Programmausführungen für ein Programm gibt, kann man sich nie sicher sein, dass nicht doch noch Contract-Verletzungen zur Laufzeit auftreten könnnen.

Ein wichtiger Nachteil von Contracts ist, dass man nur solche Contracts ausdrücken kann, die auch tatsächlich berechnet werden können. Eine Signatur wie

```
; [X] (list-of X) -> (list-of X)
```

erfordert es beispielsweise, dass man zur Überprüfung dieses Contracts ein Prädikat benötigt, welches überprüft, ob ein Listenelement ein X ist. Dieses Prädikat muss gegebenenfalls im Programm mit übergeben und gegebenenfalls über große "Entfernungen" durch das Programm "durchgeschleift" werden.

Außerdem können Contracts offensichtlich nur Restriktionen überprüfen, für die die relevanten Informationen auch als Daten zur Verfügung stehen. Eine Datendefinition wie

```
; A temperature is a number that is larger than -273.15.
; interp. temperature in degrees Celsius
```

läßt sich nicht überprüfen, weil wir eine Zahl nicht ansehen können, ob sie eine Temperatur repräsentiert. Allerdings können wir durch eine Strukturdefinition ein passendes Tag dazu definieren, welches dann auch zur Laufzeit überprüfbar ist:

```
(define-struct temperature (d))
; A temperature is: (make-temperature Number) where the number
is larger than -273.15
; interp. a temperature in degrees celsius
```

Ein pragmatischer Nachteil von dynamischen Überprüfungen ist, dass diese die Laufzeit eines Programms stark negativ beeinflussen können. Deshalb gibt es in einigen Sprachen die Möglichkeit, die dynamische Überprüfung abzustellen.

12.4 Statisch getypte Sprachen

Die letzte Variante, um Signaturen und Datendefinitionen durch die Sprache zu unterstützen, ist die Idee eines statischen Typsystems. In einem statischen Typsystem wird jedem Programmteil *vor der Ausführung* ein Typ zugeordnet, und zwar so, dass der Typ eines zusammengesetzten Programmteils nur von den Typen seiner Komponenten abhängt (sogenannte *Kompositionalität*).

Beispielsweise kann dem Ausdruck `(+ e-1 e-2)` der Typ `Number` zugeordnet werden unter der Voraussetzung, dass `e-1` und `e-2` ebenfalls diesen Typ haben.

Statische Typsysteme zeichnen sich durch zwei wichtige Eigenschaften aus: 1) Falls ein Programm den Typchecker durchläuft ("wohlgetypt" ist), so wird in allen (i.A. unendlich vielen) möglichen Programmausführungen kein Typfehler auftreten. In dem Sinne sind statische Typen viel mächtiger als Tests, weil diese immer nur eine kleine Zahl unterschiedlicher Programmausführungen überprüfen können. 2) Es gibt stets Programme, die vom Typchecker abgelehnt werden, obwohl sie eigentlich ausgeführt werden könnten, ohne das ein Typfehler auftritt. Diese Eigenschaft ist eine direkte Konsequenz des sogenannten "Theorem von Rice", welches aussagt, dass nichttriviale Eigenschaften des Verhaltens von Programmen nicht entscheidbar sind.

DrRacket unterstützt eine getypte Variante von Racket, Typed Racket. Hier ist unser Beispiel von oben in Typed Racket:

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

Wie wir sehen, gibt es in Typed Racket eine formale Syntax für die Signatur von Funktionen. Das Typsystem von Typed Racket ist so gestaltet, dass die Konsistenz der Funktion zur angegebenen Signatur überprüft werden kann, ohne das Programm bzw. einen Test auszuführen. Es kann also einmal, "once and for all", überprüft werden, dass `rest-after` die angegebene Signatur einhalten wird, und zwar für alle Parameter die den angegebenen Typen genügen.

Diese Funktion kann nun wie in der Beginning Student Language aufgerufen werden:

```
> (rest-after 2 (list 1 2 3 4))
- : (Listof Integer)
'(3 4)
```

Allerdings gibt es einen wichtigen Unterschied: Der Funktionsaufruf wird ebenfalls vor dem Auruf auf Konsistenz mit der Funktionssignatur überprüft:

```
> (rest-after "x" (list 1 2 3 4))
eval:5:0: Type Checker: type mismatch
  expected: Integer
  given: String
  in: 4


> (rest-after 2 (list 1 2 "drei" 4))
eval:6:0: Type Checker: type mismatch
  expected: (Listof Integer)
  given: (List One Positive-Byte String Positive-Byte)
  in: 4
```

Dass diese Überprüfung schon vor dem Aufruf stattfindet, erkennt man daran, dass die Typprüfung eines Aufrufs auch dann gelingt, wenn der tatsächliche Aufruf einen Laufzeitfehler generieren würde.

```
> (:print-type (rest-after (/ 1 0) (list 1 2 3 4)))
(Listof Integer)
```

Auch die Prüfung der Funktion selber findet statt ohne die Funktion auszuführen. Ein Verstoss gegen die angegebene Signatur wird sofort angezeigt.

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))
```

```
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
eval:9:0: Type Checker: type mismatch
  expected: (Listof Integer)
  given: String
  in: xs
```

Der grosse Vorteil statischer Typprüfung ist, dass diese schon vor der Programmausführung (beispielsweise beim Entwickler und nicht beim Kunden) gefunden werden und ein wohlgetyptes Programm niemals Typfehler generieren wird. In der Theorie formalisiert man diese Eigenschaft häufig so, dass die Reduktionssemantik für die getypte Sprache stets die Wohlgetyptheit erhält, also wenn ein Programm vor der Reduktion wohlgetypt hat ist es das auch nach der Reduktion (sogenanntes "Preservation" oder "Subject Reduction" Theorem) und wohlgetypte Programme, die keine Werte sind, können stets reduziert werden (sogenanntes "Progress" Theorem).

Der größte Nachteil statischer Typprüfung ist, dass es stets Programme gibt, die vom Typchecker abgelehnt werden, obwohl ihre Ausführung keinen Fehler ergeben würde.

Hier ein kleines Programm, welches in BSL ohne Typfehler ausgeführt wird:

```
> (+ 1 (if (> 5 2) 1 "a"))
2
```

Das gleiche Programm wird in Typed Racket abgelehnt:

```
> (+ 1 (if (> 5 2) 1 "a"))
eval:10:0: Type Checker: type mismatch
  expected: Number
  given: (U One String)
  in: "a"
```

Der Entwurf von Typsystemen, mit denen möglichst viele Programme überprüft werden können, ist ein sehr aktiver Forschungszweig in der Informatik.

13 Sprachunterstützung für Algebraische Datentypen

Wie wir in §7 “Datendefinition durch Alternativen und Zerlegung: Algebraische Datentypen” und §9.1 “Rekursive Datentypen” gesehen haben, sind algebraische Datentypen essentiell zur Strukturierung von komplexen Daten. Ähnlich dazu wie Signaturen und Datendefinitionen unterschiedlich gut durch Sprachmittel unterstützt werden können (§12 “Sprachunterstützung für Datendefinitionen und Signaturen”), gibt es auch bei algebraischen Datentypen Unterschiede darin, wie gut diese von der Programmiersprache unterstützt werden.

Wir werden uns vier verschiedene Arten anschauen, wie man algebraische Datentypen ausdrücken kann und diese im Anschluss bewerten.

Zu diesem Zweck betrachten wir folgende Datendefinitionen für arithmetische Ausdrücke:

Beispiel:

```
; An Expression is one of:
; - (make-literal Number)
; - (make-addition Expression Expression)
; interp. abstract syntax of arithmetic expressions
```

Als "Interface" für den Datentyp wollen wir die folgende Menge von Konstruktoren, Destruktoren und Prädikaten betrachten:

```
; Number -> Expression
; constructs a literal exression
(define (make-literal value) ...)

; Expression -> Number
; returns the number of a literal
; throws an error if lit is not a literal
(define (literal-value lit) ...)

; [X] X -> Bool
; returns #true iff x is a literal
(define (literal? x) ...)

; Expression Expression -> Expression
; constructs an addition expression
(define (make-addition lhs rhs) ...)

; [X] X -> Bool
; returns #true iff x is an addition expression
(define (addition? x) ...)

; Expression -> Expression
; returns left hand side of an addition expression
; throws an error if e is not an addition expression
```

```
(define (addition-lhs e) ...)

; Expression -> Expression
; returns right hand side of an addition expression
; throws an error if e is not an addition expression
(define (addition-rhs e) ...)
```

Wir werden nun unterschiedliche Arten betrachten, wie wir diesen Datentyp repräsentieren können.

13.1 ADTs mit Listen und S-Expressions

Wie in §11.4 “S-Expressions” diskutiert, können verschachtelte Listen mit Zahlen, Strings etc. – also S-Expressions – als universelle Datenstruktur verwendet werden. Hier ist eine Realisierung der Funktionen von oben auf Basis von S-Expressions:

```
(define (make-literal n)
  (list 'literal n))

(define (literal-value l)
  (if (literal? l)
      (second l)
      (error 'not-a-literal)))

(define (literal? l)
  (and
   (cons? l)
   (symbol? (first l))
   (symbol=? (first l) 'literal)))

(define (make-addition e1 e2)
  (list 'addition e1 e2))

(define (addition? e)
  (and
   (cons? e)
   (symbol? (first e))
   (symbol=? (first e) 'addition)))

(define (addition-lhs e)
  (if (addition? e)
      (second e)
      (error 'not-an-addition)))

(define (addition-rhs e)
  (if (addition? e)
```

```
      (third e)
      (error 'not-an-addition)))
```

Auf Basis dieses Interfaces können nun Funktionen definiert werden, wie zum Beispiel ein Interpreter für die Ausdrücke:

```
(define (calc e)
  (cond [(addition? e) (+ (calc (addition-lhs e))
                          (calc (addition-rhs e)))]
        [(literal? e) (literal-value e)]))


> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
'(addition (addition (literal 0) (literal 1)) (literal 2))


> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
3
```

Beachten Sie, dass der Code von `calc` in keiner Weise von der Repräsentation der Ausdrücke abhängt sondern lediglich auf Basis des Interfaces definiert wurde.

13.2 ADTs mit Strukturdefinitionen

In dieser Variante verwenden wir Strukturdefinitionen, um das obige Interface zu implementieren. Wir haben die Namen so gewählt, dass sie mit denen, die durch `define-struct` gebunden werden, übereinstimmen, deshalb können wir in zwei Zeilen das gesamte Interface implementieren:

```
(define-struct literal (value))
(define-struct addition (lhs rhs))
```

Auch in dieser Variante können wir nun wieder Funktionen auf Basis des Interfaces implementieren. Die `calc` Funktion aus dem vorherigen Abschnitt funktioniert unverändert mit der `define-struct` Repräsentation von algebraischen Datentypen:

```
(define (calc e)
  (cond [(addition? e) (+ (calc (addition-lhs e))
                          (calc (addition-rhs e)))]
        [(literal? e) (literal-value e)]))
```

Allerdings haben wir durch `define-struct` nun eine neue, komfortablere Möglichkeit, um algebraische Datentypen zu verarbeiten, nämlich Pattern Matching (§10 “Pattern Matching”):

```
(define (calc e)
  (match e
    [(addition e1 e2) (+ (calc e1) (calc e2))]
    [(literal x) x]))


> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
(addition (addition (literal 0) (literal 1)) (literal 2))


> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
3
```

13.3 ADTs mit `define-type`

Wechseln Sie zum Nutzen von `define-type` auf die "Intermediate Student Language"

Eine neue Möglichkeit, um algebraische Datentypen zu repräsentieren, bietet das `2htdp/abstraction` Teachpack mit dem `define-type` Konstrukt. Im Unterschied zu `define-struct` bietet `define-type` direkte Unterstützung für Summentypen, daher kann der Summentyp `Expression` mit seinen unterschiedlichen Alternativen direkt definiert werden. Ein weiterer wichtiger Unterschied zu `define-type` ist, dass zu jedem Feld einer Alternative eine Prädikatsfunktion angegeben wird, die definiert, welche Werte für dieses Feld zulässig sind. Diese Prädikatsfunktionen sind eine Form von dynamisch überprüften Contracts (siehe §12.3 “Dynamisch überprüfte Signaturen und Contracts”).

```
(define-type Expression
  (literal (value number?))
  (addition (left Expression?) (right Expression?)))
```

Analog zu `define-struct` (§6.2 “Strukturdefinitionen”) definiert `define-type` für jede Alternative Konstruktor-, Selektor- und Prädikatsfunktionen, so dass wir Werte dieses Typs auf die gleiche Weise definieren können:

```
> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
(addition (addition (literal 0) (literal 1)) (literal 2))
```

Allerdings wird bei Aufruf der Konstruktoren überprüft, ob die Felder die dazugehörige Prädikatsfunktion erfüllen. Der folgende Aufruf, der bei der `define-struct` Variante ausgeführt werden könnte, schlägt nun zur Laufzeit fehl:

```
> (make-addition 0 1)
```

```
make-addition: expects an undefined or a Expression, given 0
  in: the 1st argument of
      (->
       (or/c undefined? Expression?)
       (or/c undefined? Expression?)
       addition?)
  contract from: make-addition
  blaming: use
   (assuming the contract is correct)
  at: eval:13.0
```

Passend zu `define-type` gibt es auch eine Erweiterung von `match`, nämlich `type-case`. Mit Hilfe von `type-case` kann die `calc` Funktion nun wie folgt definiert werden:

```
(define (calc e)
  (type-case Expression e
    [literal (value) value]
    [addition (e1 e2) (+ (calc e1) (calc e2))]))


> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
3
```

Der wichtigste Unterschied zwischen `match` und `type-case` ist, dass der Typ `Expression`, auf dem wir eine Fallunterscheidung machen möchten, explizit mit angegeben wird. Dies ermöglicht es der DrRacket Umgebung, bereits *vor* der Programmausführung zu überprüfen, ob alle Fälle abgedeckt wurden. Beispielsweise wird die folgende Definition bereits vor der Ausführung mit einer entsprechenden Fehlermeldung zurückgewiesen.

```
> (define (calc2 e)
    (type-case Expression e
      [addition (e1 e2) (- (calc2 e1) (calc2 e2))]))
type-case: syntax error; probable cause: you did not include a case for the literal
variant, or no else-branch was present
```

Der Preis für diese Vollständigkeitsüberprüfung ist, dass `type-case` nur ein sehr eingeschränktes Pattern Matching erlaubt. Beispielsweise ist es nicht erlaubt, Literale, verschachtelte Pattern, oder nicht-lineares Pattern Matching zu verwenden. Im Allgemeinen haben die Klauseln von `type-case` stets die Form `((variant (name-1 ... name-n) body-expression))`, wobei in `body-expression` die Namen `name-1` bis `name-n` verwendet werden können.

13.4 Ausblick: ADTs mit Zahlen

Ist es auch möglich, algebraische Datentypen zu repräsentieren, wenn man weder Listen/S-expressions noch `define-struct` oder `define-type` zur Verfügung hat, sondern der einzige eingebaute Datentyp (natürliche) Zahlen sind? Diese Frage ist von großer Bedeutung in der theoretischen Informatik, denn dort möchte man häufig möglichst minimale und einfache Rechenmodelle definieren, die dennoch prinzipiell mächtig genug wären, um beliebige Programme darin auszudrücken. Es spielt hierbei in der Regel keine Rolle, ob diese Techniken praxistauglich oder effizient sind. Wichtig ist nur, ob eine Kodierung prinzipiell möglich ist. Der Grund, wieso die Rechenmodelle möglichst minimal sein sollen, ist der, dass dann leichter wichtige Eigenschaften dieser Rechenmodelle analysiert und bewiesen werden können.

Beispielsweise ist es in vielen Rechenmodellen wichtig, ob es eine Art universelles Programm gibt, das die Repräsentation eines Programms als Eingabe bekommt und dann dieses Programm quasi simuliert. In der BSL wäre dies beispielsweise ein Programm, welches eine Repräsentation eines BSL Programms als Eingabe bekommt und dann die Ausführung dieses Programms simuliert. Ein solches Programm nennt man auch *Selbstinterpreter*. In der theoretischen Informatik kommen solche Konstruktionen häufig vor, beispielsweise bei der sogenannten Universellen Turing-Maschine, im Normalformtheorem in der Theorie rekursiver Funktionen, beim sogenannten "Halteproblem", und im Beweis der Unvollständigkeitstheoreme von Gödel. Als Anerkennung der Beiträge von Kurt Gödel werden solche Repräsentationstechniken in der Berechenbarkeitstheorie auch "Gödelisierung" genannt. All diese Rechenmodelle haben gemein, dass Zahlen quasi der einzige eingebaute Datentyp sind.

Bevor wir algebraische Datentypen durch Zahlen repräsentieren, überlegen wir uns erstmal einen Spezialfall: Wie können wir ein Paar von zwei Zahlen, beispielsweise 17 und 12, durch eine einzelne Zahl repräsentieren? Eine Idee wäre, die Zahlen in Dezimalrepräsentation hintereinander zu schreiben: 1712. Aber woher wissen wir dann, dass 17 und 12 gemeint waren und nicht etwa 1 und 712 oder 171 und 2? Eine Lösung hierzu wäre, als Trennzeichen bestimmte Zahlenfolgen zu wählen, die in den Zahlen selbst nicht vorkommen dürfen. Wir werden eine andere Technik wählen, die von Kurt Gödel vorgeschlagen wurde und die auf der Eindeutigkeit der Primfaktorzerlegung beruht.

Die Eindeutigkeit der Primfaktorzerlegung sagt aus, dass jede Zahl eindeutig in ihre Primfaktoren zerlegt werden kann. Wenn die Primzahlen durch $p_1, p_2, \ldots$ bezeichnet werden, so bedeutet dies, dass es für jede natürliche Zahl n eine Zahl k sowie Zahlen $n_1, \ldots, n_k$ gibt, so dass $n = p_1^{n_1} \cdot p_2^{n_2} \cdot \ldots \cdot p_k^{n_k}$. Beispielsweise kann die Zahl 18 zerlegt werden in $18 = 2^1 \cdot 3^2$.

Aufgrund der Eindeutigkeit der Primfaktoren könnten wir beispielsweise Paare mit Hilfe der ersten beidem Primzahlen, 2 und 3, repräsentieren. Beispielsweise könnten die Zahlen 17 und 12 von oben durch die Zahl $2^{17} \cdot 3^{12} = 69657034752$ repräsentiert werden. Um die beiden Zahlen wieder zu extrahieren, berechnen wir, wie häufig sich die Zahl ohne Rest durch 2 bzw. 3 teilen lässt.

Diese Idee können wir durch folgendes Programm ausdrücken:

```
; Nat Nat -> Nat
```

```
; computes how often n can be divided by m
(define (how-often-dividable-by n m)
  (if (zero? (modulo n m))
      (add1 (how-often-dividable-by (/ n m) m))
      0))


> (how-often-dividable-by 69657034752 2)
17
> (how-often-dividable-by 69657034752 3)
12
```

Offensichtlich können durch Nutzen von mehr Primzahlen mit der gleichen Technik beliebig lange Listen von (natürlichen) Zahlen durch eine einzelne Zahl repräsentiert werden. Verschachtelte Listen, also quasi S-Expressions, können durch Verschachtelung der gleichen Kodierungstechnik repräsentiert werden. Wollen wir beispielsweise ein Paar von zwei Paaren repräsentieren und n_1 ist die Kodierung des ersten Paars und n_2 die Kodierung des zweiten Paars, so kann das Paar der beiden Paare durch $2^{n_1} \cdot 3^{n_2}$ repräsentiert werden.

Bleibt noch die Frage, wie wir Summentypen repräsentieren. Hierzu können wir einfach für jede Variante disjunkte Mengen von Primzahlen nehmen.

Auf Basis dieser Idee können wir nun das Interface für den Expression Datentyp implementieren:

```
(define prime-1 2)
(define prime-2 3)
(define prime-3 5)

(define (make-literal value) (expt prime-1 value))
(define (literal-value l) (how-often-dividable-by l prime-1))

(define (make-addition e-1 e-2)
  (* (expt prime-2 e-1)
     (expt prime-3 e-2)))

(define (addition-lhs e)
  (how-often-dividable-by e prime-2))

(define (addition-rhs e)
  (how-often-dividable-by e prime-3))

(define (addition? e)
  (zero? (modulo e prime-2)))

(define (literal? e)
  (or (= e 1) (zero? (modulo e prime-1))))
```

Diese Kodierung ergibt sehr schnell sehr große Zahlen, daher ist sie nicht für praktische Zwecke brauchbar, aber sie funktioniert wie gewünscht:

```
> (define e-1 (make-addition (make-addition (make-
literal 0) (make-literal 1)) (make-literal 2)))
> e-1
380166742320848568199802495386788316875
> (literal-value (addition-lhs (addition-lhs e-1)))
0
> (literal-value (addition-rhs (addition-lhs e-1)))
1
> (literal-value (addition-rhs e-1))
2
```

Auch Funktionen wie die `calc` Funktionen funktionieren weiterhin auch auf Basis dieser ADT Repräsentation:

```
(define (calc e)
  (cond [(addition? e) (+ (calc (addition-lhs e))
                          (calc (addition-rhs e)))]
        [(literal? e) (literal-value e)]))


> (calc e-1)
3
```

13.5 Diskussion

Die unterschiedlichen Möglichkeiten, algebraische Datentypen (ADTs) zu unterstützen, unterscheiden sich in wichtigen Eigenschaften. Im folgenden bezeichnen wir die Variante ADTs mit Listen und S-Expressions als a), ADTs mit Strukturdefinitionen als b), ADTs mit `define-type` als c), und ADTs mit Zahlen als d).

- Boilerplate: Gibt es in der Art, wie ADTs kodiert werden, Verstöße gegen das DRY-Prinzip? Solchen Code nennt man *boilerplate code*. Bei a) und d) haben wir solchen boilerplate code, denn die Konstruktoren, Destruktoren und Prädikate müssen manuell auf eine mechanische Art und Weise implementiert werden.

- Unterscheidbarkeit der Alternativen: Können die Alternativen bei Summentypen stets unterschieden werden? Bei b) und c) bekommen wir durch die Sprache eindeutige Prädikate geliefert, mit denen wir die Alternativen unterscheiden können. Bei a) und d) muss der Programmierer diszipliniert genug arbeiten um diese Eigenschaft sicherzustellen.

- Erweiterbarkeit der Summen: Können Summen leicht um weitere Alternativen erweitert werden? Ein Beispiel wäre, dass der Typ für arithmetische Ausdrücke

von oben noch durch eine Alternative zur Multiplikation von zwei arithmetischen Ausdrücken erweitert werden soll. Mit "erweitert" ist gemeint, dass hierzu keine Modifiktion von bestehendem Code erforderlich ist. Bei den Varianten a), b) und d) können wir leicht weitere Alternativen zu der jeweiligen Repräsentation des ADTs hinzufügen, beispielsweise in der Variante b) durch Definition einer weiteren Struktur. Allerdings werden bestehende Programme, die solche ADTs als Eingabe bekommen, in der Regel nicht korrekt auf diesen erweiterten ADTs arbeiten, denn sie berücksichtigen den hinzugekommenen Fall nicht. Es ist auch nicht ohne weiteres möglich, ohne Modifikation dieser Programme die Funktionalität für den hinzugekommenen Fall hinzuzufügen. [5] Die Variante c) hingegen ist eine "geschlossene" Kodierung eines ADTs: Alle Alternativen werden an einer Stelle definiert, daher ist eine Erweiterung um weitere Alternativen ohne Modifikation von Code nicht ohne weiteres möglich.

- Erweiterbarkeit der Produkte: Können Produkte leicht um weitere Komponenten erweitert werden? Ein Beispiel wäre, dass die Alternative für Literale noch um eine weitere Zahl erweitert werden soll, welches aussagt, dass die Zahl nur eine Annäherung an die genaue Zahl ist und die zweite Zahl gibt an, wie präzise die Annäherung ist. In der Variante a) können wir Produkte leicht dadurch erweitern, dass wir die Listen länger machen. Auf eine ähnliche Weise können wir auch in der Variante d) die Produkte erweitern. In den Varianten b) und c) hingegen ist die Anzahl der Komponenten (die "Arität") der Konstruktoren in den Struktur- bzw. Typdefinitionen festgelegt und diese läßt sich nicht ohne Modifikation des bestehenden Codes erweitern.

- Typsicherheit: Ist sichergestellt, dass die Datendefinition vom Programm eingehalten wird? Bei den Varianten a) und d) gibt es keinerlei Unterstützung für Typsicherheit; falls Daten konstruiert werden, die gegen die Datendefinition verstossen, so wird dies weder vor der Programmausführung noch während der Konstruktion sondern erst möglicherweise im weiteren Verlauf der Programmausführung festgestellt. Bei b) wird zumindest sichergestellt, dass den Konstruktoren die korrekte Anzahl von Parametern übergeben wird (Arität). Diese Prüfung findet, je nach Sprachvariante, schon vor der Programmausführung (BSL) bzw. während der Ausführung des Konstruktors (ISL) statt. Bei der Variante c) hingegen wird eine Variante von Contracts verwendet (siehe §12.3 “Dynamisch überprüfte Signaturen und Contracts”), um während der Konstruktion der Daten zu überprüfen, ob die Komponenten zur Datendefinition passen.

- Vollständigkeit von Fallunterscheidungen: Wird sichergestellt, dass Fallunterscheidungen bei Summentypen alle Möglichkeiten berücksichtigen? Die einzige Variante, die die Vollständigkeit vor der Programmausführung überprüfen kann, ist c). Dies ist ein wichtiger Vorteil, der daraus resultiert, dass ADTs in dieser Variante "geschlossen" sind.

[5] Dieses und ähnliche Erweiterungsprobleme werden unter dem Namen "Expression Problem" in vielen wissenschaftlichen Artikeln diskutiert.

- Zeit- und Platzeffizienz: Ist die Kodierung von ADTs effizient bezüglich des Platzbedarfs zur Repräsentation der Daten im Speicher sowie bezüglich des Zeitbedarfs zur Konstruktion und Destruktion (Analyse) von ADTs? Effizienz ist kein wichtiges Thema in dieser Lehrveranstaltung, aber die Variante d) fällt aus diesen Grund für praktische Anwendungen aus.

- Datentyp-generische Programme: Ist es möglich, Programme zu schreiben, die mit jedem algebraischen Datentyp funktionieren, wie beispielsweise eine Funktion, die zu einem Wert, der einen beliebigen algebraischen Datentyp hat, die Tiefe des Baums berechnet? Bei universellen Datenformaten wie S-Expressions ist dies leicht, während es nicht offensichtlich ist, wie man dies bei mittels `define-struct` oder `define-type` definierten ADTs hinbekommen könnte. Datentyp-generische Programmierung, insbesondere für statisch getypte Sprachen, ist ein sehr aktiver Zweig in der Programmiersprachenforschung.

13.6 Ausblick: ADTs mit Dictionaries

Eine wichtige Datenstruktur, die es in fast allen Programmiersprachen gibt, sind *dictionaries*, manchmal auch bezeichnet als *associative array*, *map*, oder *hash table*. Dictionaries repräsentieren partielle Funktionen mit einer endlichen Definitionsmenge (den sog. *keys*); jedem key wird also maximal ein Wert zugeordnet. Eine gängige Notation für Dictionaries ist JSON, die JavaScript Object Notation. Eine Person mit Namen und Vornamen könnte in JSON beispielsweise so aufgeschrieben werden:
`{ name: 'Ostermann', vorname: 'Klaus'}`
In diesem Fall sind `name` und `vorname` die Keys und `'Ostermann'` und `'Klaus'` die jeweils zugeordneten Werte. Dictionaries sind wie Listen, bei denen die Elemente nicht durch ihre Position in der Liste sondern durch einen Key adressiert werden. Da Dictionaries genau wie Listen verschachtelt werden können, können Dictionaries ähnlich wie s-expressions als universelle Datenstruktur verwendet werden.

Beispielsweise könnten die aritmetischen Ausdrücke von oben auch durch Dictionaries repräsentiert werden:
`{ kind: 'add', lhs: { kind: 'lit', val: 7}, rhs: { kind: 'lit', val: 12}};`

Dictionaries spielen in einer Reihe von Programmiersprachen eine zentrale Rolle und werden als universelle Datenstruktur im Sinne von S-Expressions verwendet, beispielsweise in JavaScript, AWK, Lua, Python, Perl und Ruby. Unter dem Schlagwort "NoSQL" gibt es auch eine Reihe von Datenbanktechnologien, in denen Dictionaries eine zentrale Rolle spielen.

Ein Interpreter für arithmetische Ausdrücke, die als Dictionaries repräsentiert werden, kann beispielsweise in JavaScript wie folgt aussehen:

```
var calc = function (e) {
  switch (e.kind) {
    case 'lit': return e.val;
    case 'add': return calc(e.lhs)+calc(e.rhs);
  }
}
```

```
calc({ kind: 'add', lhs: { kind: 'lit', val: 7}, rhs: { kind:
'lit', val: 12}});
```

Bezüglich der Diskussion in §13.5 “Diskussion” verhalten sich dictionary-repräsentierte ADTs ähnlich wie S-Expressions. [6] Ein wichtiger Vorteil gegenüber S-Expressions ist, dass der Zugriff auf die Komponenten keinen "boilerplate" Code erfordert sondern mit Hilfe der "Dot-Notation" (wie `e.lhs`) effizient ausgedrückt werden kann.

[6] Einige Sprachen, wie beispielsweise Python, verwenden statt der Dot-Notation die Notation `e['lhs']`.

14 DRY: Abstraktion überall!

Dieser Teil des Skripts basiert auf [HTDP/2e] Part III

Wir haben bereits in Abschnitt §2.6 “DRY: Don’t Repeat Yourself!” das "Don’t Repeat Yourself" Prinzip kennengelernt: Gute Programme sollten keine Wiederholungen enthalten und nicht redundant sein. Wir haben gelernt, dass wir diese Redundanz durch verschiedene *Abstraktionsmechanismen* beseitigen können.

In diesem Abschnitt werden wir zunächst diese bereits bekannten Abstraktionsmechanismen nochmal Revue passieren lassen. Allerdings sind diese Mechanismen nicht für alle Arten von Redundanz geeignet. Wir werden in diesem Kapitel daher weitere mächtige Abstraktionsmechanismen kennenlernen. Hierzu brauchen wir neue Sprachkonzepte, die Sie bitte durch das Umschalten auf den "Zwischenstufe mit Lambda" Sprachlevel aktivieren.

Unsere Programmiersprache wird hierdurch erstmal komplexer. Am Ende des Kapitels werden wir sehen, dass es allerdings eine so mächtige Art der Abstraktion gibt, dass wir damit viele andere Abstraktionsarten damit subsumieren können. Dies wird die Sprache konzeptuell wieder vereinfachen.

14.1 Abstraktion von Konstanten

Die wohl einfachste Art der Abstraktion ist anwendbar, wenn es in einem Programm an mehreren Stellen Ausdrücke gibt, die konstant sind, also immer zum gleichen Wert auswerten. Der Abstraktionsmechanismus, den wir in diesem Fall verwenden können, ist die Definition von Konstanten. Diese Möglichkeit haben wir bereits ausführlich im Abschnitt §2.6 “DRY: Don’t Repeat Yourself!” besprochen.

14.2 Funktionale Abstraktion

Häufig gibt es in einem Programm die Situation, dass an vielen Stellen Ausdrücke vorkommen, die nicht gleich sind, aber sich nur in einigen Stellen in den verwendeten Werten unterscheide.

Beispiel:

```
; (list-of String) -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (cond
    [(empty? l) false]
    [else
     (or
      (string=? (first l) "dog")
      (contains-dog?
       (rest l)))]))



; (list-of String) -> Boolean
```

```
; does l contain "cat"
(define (contains-cat? l)
  (cond
    [(empty? l) false]
    [else
     (or
      (string=? (first l) "cat")
      (contains-cat?
       (rest l)))]))
```

Die Ausdrücke in den beiden Funktionsbodies sind bis auf den String `"dog"` beziehungsweise `"cat"` gleich.

Gute Programmierer sind zu faul, viele ähnliche Ausdrücke und Funktionen zu schreiben. Für die Elimination dieser Art von Redundanz ist *funktionale Abstraktion* geeignet:

```
; String (list-of String) -> Boolean
; to determine whether l contains the string s
(define (contains? s l)
  (cond
    [(empty? l) false]
    [else (or (string=? (first l) s)
              (contains? s (rest l)))]))
```

Falls gewünscht, können `contains-dog?` und `contains-cat?` auf Basis dieser Funktion wieder hergestellt werden:

```
; (list-of String) -> Boolean
; does l contain "dog"
(define (contains-dog? l)
  (contains? "dog" l))

; (list-of String) -> Boolean
; does l contain "cat"
(define (contains-cat? l)
  (contains? "cat" l))
```

Alternativ dazu können auch die Aufrufer von `contains-dog?` und `contains-cat?` so modifiziert werden, dass stattdessen `contains?` verwendet wird.

Wenn Sie einmal `contains?` definiert haben, wird es nie wieder nötig sein, eine Funktion ähnlich wie die erste Variante von `contains-dog?` zu definieren.

14.3 Funktionen als Funktionsparameter

Betrachten Sie folgenden Code:

```
; (list-of Number) -> Number
```

```
; adds all numbers in l
(define (add-numbers l)
  (cond
    [(empty? l) 0]
    [else
     (+ (first l)
        (add-numbers (rest l)))]))


; (list-of Number) -> Number
; multiplies all numbers in l
(define (mult-numbers l)
  (cond
    [(empty? l) 1]
    [else
     (* (first l)
        (mult-numbers (rest l)))]))
```

In diesem Beispiel haben wir eine ähnliche Situation wie bei `contains-dog?` und `contains-cat?`: Beide Funktionen unterscheiden sich nur an zwei Stellen: a) Im `empty?` Fall wird einmal `0` und einmal `1` zurückgegeben. b) Im anderen Fall wird einmal addiert und das andere mal multipliziert.

Der Fall a) ist völlig analog zu `contains-dog?` und `contains-cat?` lösbar, indem wir diesen Wert als Parameter übergeben.

Eine andere Situation haben wir im Fall b). Hier unterscheiden sich die Funktionen nicht in einem Wert, sondern im Namen einer aufgerufenen Funktion! Hierzu brauchen wir eine neue Art der Abstraktion, nämlich die Möglichkeit, Funktionen als Parameter an andere Funktionen zu übergeben.

Wenn wir mal einen Augenblick lang dieses Problem ignorieren und einfach auf gut Glück das offensichtliche tun:

```
(define (op-numbers op z l)
  (cond
    [(empty? l) z]
    [else
     (op (first l)
         (op-numbers (rest l) z))]))

(define (add-numbers l) (op-numbers + 0 l))
(define (mult-numbers l) (op-numbers * 1 l))
```

... so stellen wir fest, dass wir tatsächlich über Funktionen genau so abstrahieren können wie über Werte! Wir können also Funktionen als Parameter übergeben und diese Parameter an der ersten Position eines Funktionsaufrufs verwenden!

Das entscheidende an `op-numbers` ist nicht, dass `add-numbers` und `mult-numbers` nun Einzeiler sind. Der entscheidende Punkt ist, dass wir nun eine sehr

mächtige abstrakte Funktion geschaffen haben, die wir nun universell auch für viele andere Dinge benutzen können.

Zum einen können wir feststellen, dass `op-numbers` überhaupt keinen Code mehr enthält, der spezifisch für Zahlen ist. Das deutet darauf hin, dass wir die Funktion auch mit Listen anderer Werte verwenden können. Der Name `op-numbers` ist daher irreführend und wir benennen die Funktion um:

```
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
     (op (first l)
         (op-elements op z (rest l)))]))
```

Zum anderen können wir als Parameter für `op` nicht nur primitive, sondern auch beliebige selber definierte Funktionen übergeben.

Schauen wir uns an einigen Beispielen an, was man mit `op-elements` alles machen kann:

Wir können Zahlen aufsummieren:

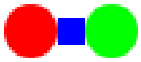
```
> (op-elements + 0 (list 5 8 12))
25
```

Wir können Strings zusammenhängen:

```
> (op-elements string-append "" (list "ab" "cd" "ef"))
"abcdef"
```

Wir können Bilder komponieren:

```
> (op-elements beside empty-image (list (circle 10 "solid" "red")
                                        (rectangle 10 10 "solid" "blue")
                                        (circle 10 "solid" "green")))
```



Wir können eine Liste kopieren, was alleine nicht sehr nützlich ist:

```
> (op-elements cons empty (list 5 8 12 2 9))
'(5 8 12 2 9)
```

...aber zeigt, dass wir mit leichten Variationen davon andere interessante Dinge tun können. Beispielsweise können wir zwei Listen aneinanderhängen:

```
(define (append-list l1 l2)
  (op-elements cons l2 l1))


> (append-list (list 1 2) (list 3 4))
'(1 2 3 4)
```

Wir können aus einer Liste von Listen eine Liste machen:

```
> (op-elements append-list empty (list (list 1 2) (list 3 4) (list 5 6)))
'(1 2 3 4 5 6)
```

Und schliesslich, als etwas fortgeschrittenes Beispiel, können wir mit `op-elements` eine Liste von Zahlen sortieren. Zu diesem Zweck benötigen wir eine Hilfsfunktion, die eine Zahl in eine Liste sortierter Zahlen einfügt:

```
; A (sorted-list-of Number) is a (list-of Number) which is
sorted by "<"

; Number (sorted-list-of Number) -> (sorted-list-of Number)
; inserts x into a sorted list xs
(check-expect (insert 5 (list 1 4 9 12)) (list 1 4 5 9 12))
(define (insert x xs)
  (cond [(empty? xs) (list x)]
        [(cons? xs) (if (< x (first xs))
                        (cons x xs)
                        (cons (first xs) (insert x (rest xs))))]))
```

Und schon können wir aus `insert` und `op-elements` eine Sortierfunktion bauen:

```
> (op-elements insert empty (list 5 2 1 69 3 66 55))
'(1 2 3 5 55 66 69)
```

Diese Sortierfunktion wird in der Algorithmik *insertion sort* genannt.
Die Funktion `op-elements` ist übrigens so nützlich, dass sie auch als primitive Funktion zur Verfügung gestellt wird, und zwar mit dem Namen `foldr`.

Es ist nicht so wichtig, wenn sie dieses letzte Beispiel nicht verstehen. Es soll lediglich demonstrieren, was man gewinnt, wenn man durch Abstraktion wiederverwendbare Funktionen definiert.

Übrigens geht es beim Entwurf von Funktionen wie `op-elements` nicht nur um die Wiederverwendung von Code, sondern auch um die "Wiederverwendung" von Korrektheitsargumenten. Betrachten Sie beispielsweise die Frage, ob ein Programm terminiert oder vielleicht eine Endlosschleife enthält. Dies ist einer der häufigsten Fehler in Programmen und bei Schleifen, wie bei den Beispielen oben, immer ein potentielles Problem. Wenn wir einmal zeigen, dass `op-elements` terminiert, so wissen wir, dass alle Schleifen, die wir mit Hilfe von `op-elements` bilden, auch terminieren (sofern die Funktion, die wir als Argument übergeben, terminiert). Wenn wir nicht `op-elements` benutzen würden, müssten wir diese Überlegung jedes mal aufs neue anstellen.

14.4 Abstraktion in Signaturen, Typen und Datendefinitionen

14.4.1 Abstraktion in Signaturen

Redundanz kann auch in Signaturen auftreten, und zwar in dem Sinne, dass es viele Signaturen für den gleichen Funktionsbody geben kann.

Betrachten wir zum Beispiel die Funktion `second`, die das zweite Element aus einer Liste von Strings berechnet:

```
; (list-of String) -> String
(define (second l) (first (rest l)))
```

Hier ist eine Funktion, die das zweite Element aus einer Liste von Zahlen berechnet:

```
; (list-of Number) -> Number
(define (second l) (first (rest l)))
```

Die Funktionsdefinitionen sind bis auf die Signatur identisch. Wir könnten, um den Code nicht zu duplizieren, mehrere Signaturen zu der gleichen Funktion schreiben:

```
; (list-of String) -> String
; (list-of Number) -> Number
(define (second l) (first (rest l)))
```

Offensichtlich ist dies aber keine sehr gute Idee, weil wir die Liste der Signaturen erweitern müssen, immer wenn die Funktion mit einem neuem Elementtyp verwendet werden soll. Es ist auch nicht möglich, alle Signaturen hinzuschreiben, denn es gibt unendlich viele, z.B. folgende unendliche Sequenz von Signaturen:

```
; (list-of String) -> String
; (list-of (list-of String)) -> (list-of String)
; (list-of (list-of (list-of String))) -> (list-of (list-of
String))
; (list-of (list-of (list-of (list-of String)))) -> (list-of
(list-of (list-of String)))
; ...
```

Wir haben jedoch bereits informell einen Abstraktionsmechanismus kennengelernt, mit dem wir diese Art von Redundanz eliminieren können: Typvariablen. Eine Signatur mit Typvariablen steht implizit für alle möglichen Signaturen, die sich ergeben, wenn man die Typvariablen durch Typen ersetzt. Wir kennzeichnen Namen als Typvariablen, indem wir den Namen der Typvariablen in eckige Klammern vor die Signatur setzen. In der Signatur können wir dann die Typvariable verwenden. Hier ist das Beispiel von oben mit Typvariablen:

```
; [X] (list-of X) -> X
(define (second l) (first (rest l)))
```

Der Typ, durch den eine Typvariable ersetzt wird, darf zwar beliebig sein, aber er muss für alle Vorkommen der Typvariablen der gleiche sein. So ist zum Beispiel diese Signatur nicht äquivalent zur vorherigen, sondern falsch:

```
; [X Y] (list-of X) -> Y
(define (second l) (first (rest l)))
```

Der Grund ist, dass die Signatur auch für konkrete Signaturen wie

```
; (list-of Number) -> String
```

steht, aber dies ist keine gültige Signatur für `second`.

14.4.2 Signaturen für Argumente, die Funktionen sind

Wir haben in diesem Kapitel die Möglichkeit eingeführt, Funktionen als Parameter an andere Funtionen zu übergeben. Wir wissen aber noch nicht, wie wir Signaturen solcher Funktionen beschreiben.

Dieses Problem lösen wir dadurch, dass wir es zulassen, Signaturen als Typen zu verwenden. Betrachten wir beispielsweise eine Funktion, die ein Bild mit einem Kreis kombiniert, aber die Art, wie es kombiniert werden soll, zu einem Parameter der Funktion macht:

```
(define (add-circle f img)
  (f img (circle 10 "solid" "red")))
```

Beispielsweise kann diese Funktion so verwendet werden:

```
> (add-circle beside (rectangle 10 10 "solid" "blue"))
```

oder so:

```
> (add-circle above (rectangle 10 10 "solid" "blue"))
```

Die Signatur von `add-circle` können wir nun so beschreiben:

```
; (Image Image -> Image) Image -> Image
(define (add-circle f img)
  (f img (circle 10 "solid" "red")))
```

Diese Signatur sagt aus, dass der erste Parameter eine Funktion mit der Signatur Image -> Image sein muss. Wir verwenden also nun Signaturen als Typen oder, anders gesagt, wir unterscheiden nicht mehr zwischen Signaturen und Typen.

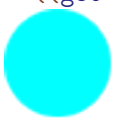
Funktionen können nicht nur Funktionen als Argumente bekommen, sondern auch Funktionen als Ergebnis zurückliefern. Beispiel:

```
; Color -> Image
(define (big-circle color) (circle 20 "solid" color))

; Color -> Image
(define (small-circle color) (circle 10 "solid" color))

; String -> (Color -> Image)
(define (get-circle-maker size)
  (cond [(string=? size "big") big-circle]
        [(string=? size "small") small-circle]))
```

Die Funktion `get-circle-maker` können wir nun aufrufen und sie liefert eine Funktion zurück. Das bedeutet, dass wir einen Aufruf von `get-circle-maker` an der *ersten* Position eines Funktionsaufrufs haben können. Bisher stand an dieser Position immer der Name einer Funktion oder (seit wir Funktionen als Parameter kennen) Namen von Funktionsparametern.

Beispiel:

```
> ((get-circle-maker "big") "cyan")
```



Beachten Sie in dem Beispiel die Klammersetzung: Dieser Aufruf ist ein Funtionsaufruf einer Funktion mit einem Parameter. Die Funktion, die wir aufrufen, ist `(get-circle-maker "big")` und ihr Parameter ist `"cyan"`. Dies ist also etwas völlig anderes als der ähnlich aussehende, in diesem Beispiel aber unsinnige Ausdruck `(get-circle-maker "big" "cyan")`.

14.4.3 Funktionen höherer Ordnung

Funktionstypen können beliebig verschachtelt werden. Nehmen wir beispielsweise an, wir haben noch andere Funktionen mit der gleichen Signatur wie `add-circle`, zum Beispiel `add-rectangle`. Eine Funktion, die wir mit einer dieser Funktionen parametrisieren können, ist zum Beispiel:

```
; ((Image Image -> Image) Image -> Image) Image Image -> Image
(define (add-two-imgs-with f img1 img2)
  (above (f beside img1) (f beside img2)))
```

Diese Funktion können wir jetzt mit Funktionen wie `add-circle` parametrisieren:

```
> (add-two-imgs-with add-circle (circle 10 "solid" "green") (circle 10 "solid" "blue"))
```

Funktionen, in deren Signatur nur ein einziger Pfeil vorkommt (also Funktionen, die keine Funktionen als Parameter haben oder zurückgeben), heißen in diesem Kontext auch *Funktionen erster Ordnung* oder *first-order functions*. Beispiele für Funktionen erster Ordnung sind `circle` oder `cons`. Funktionen, die Funktionen als Parameter bekommen oder als Ergebnis zurückgeben, nennt man auch *Funktionen höherer Ordnung* oder *higher-order functions*. Beispiele für Funktionen höherer Ordnung sind `add-circle` und `add-two-imgs-with`.

Manchmal unterscheidet man bei Funktionen höherer Ordnung noch, was die maximale Schachtelungstiefe der Funktionspfeile in den Argumenten einer Funktion ist.

Beispielsweise nennt man `add-circle` eine *Funktion zweiter Ordnung* und `add-two-imgs-with` eine *Funktion dritter Ordnung*.

In der alltäglichen Programmierung sind Funktionen zweiter Ordnung sehr häufig, aber Funktionen von drittem und höherem Grad werden recht selten gebraucht.

14.4.4 Polymorphe Funktionen höherer Ordnung

Auch polymorphe Funktionen können Funktionen höherer Ordnung sein. Wir haben bereits die Funktion `op-elements` oben kennengelernt. Schauen wir uns an, mit welchen Typen sie in den Beispielen verwendet wird.

Im Beispiel `(op-elements + 0 (list 5 8 12))` benötigen wir für `op-elements` offensichtlich die Signatur

```
; (Number Number -> Number) Number (list-of Number) -> Number
```

während wir für `(op-elements string-append "" (list "ab" "cd" "ef"))` die Signatur

```
; (String String -> String) String (list-of String) -> String
```

erwarten. Diese Beispiele suggerieren folgende Verallgemeinerung durch Typvariablen:

```
; [X] (X X -> X) X (list-of X) -> X
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
     (op (first l)
         (op-elements op z (rest l)))]))
```

Allerdings passt diese Signatur nicht zu allen Beispielen von oben. Beispielsweise benötigen wir für das `(op-elements cons empty (list 5 8 12 2 9))` die Signatur

```
; (Number (list-of Number) -> (list-of Number)) (list-of Num-
ber) (list-of Number) -> (list-of Number)
```

Es gibt keine Ersetzung von X durch einen Typ in der Signatur oben, die diese Signatur erzeugt. Wenn wir uns die Funktionsdefinition etwas genauer hinschauen, können wir allerdings auch noch eine allgemeinere Signatur finden, nämlich diese:

```
; [X Y] (X Y -> Y) Y (list-of X) -> Y
(define (op-elements op z l)
  (cond
    [(empty? l) z]
    [else
     (op (first l)
         (op-elements op z (rest l)))]))
```

Die Signatur ist allgemeiner, weil sie für mehr konkrete Signaturen (ohne Typvariablen) steht und ist ausreichend für alle Beispiele, die wir betrachet haben.

14.4.5 Abstraktion in Datendefinitionen

Bei generischen Datentypen wie solchen für Listen oder Bäume haben wir bereits gesehen, dass man in diesem Fall nicht redundante Datendefinitionen wie folgt schreiben möchte:

```
; a List-of-String is either
; - empty
; - (cons String List-of-String)

; a List-of-Number is either
; - empty
; - (cons Number List-of-Number)
```

Ähnlich wie bei Funktionssignaturen abstrahieren wir über die Unterschiede mit Hilfe von Typvariablen:

```
; a (list-of X) is either
; - empty
; - (cons X (list-of X))
```

Im Unterschied zu Funktionssignaturen verzichten wir bei Datendefinitionen darauf, die verwendeten Typvariablen separat zu kennzeichnen (wie bei Funktionssignaturen mit den eckigen Klammern), weil bei Datendefinitionen durch die Position der Typvariablen in der ersten Zeile einer Datendefinition klar ist, dass es sich um eine Typvariable handelt.

Wir können Datendefinitionen auch nutzen, um viele weitere Eigenschaften der Werte, die durch die Datendefinition beschrieben werden, festzulegen. Zum Beispiel können wir so die Menge der nicht-leeren Listen definieren:

```
; a (nonempty-list-of X) is: (cons X (list-of X))
```

14.4.6 Grammatik der Typen und Signaturen

Wir können die Typen und Signaturen, mit denen wir nun programmieren können, durch eine Grammatik beschreiben. Diese Grammatik spiegelt gut die rekursive Struktur von Typen wieder. Nicht alle Typen, die mit dieser Grammatik gebildet werden können, sind sinnvoll. Beispielsweise ist $\langle X \rangle$ ein Typ, der nicht sinnvoll ist. Eine Bedeutung haben nur die Typen, bei denen alle vorkommenden Typvariablen durch ein im Ableitungsbaum darüber liegende Typabstraktion der Form `[` $\langle X \rangle$ `]` $\langle Typ \rangle$) gebunden wurden.

$\langle Typ \rangle$ ::= $\langle Basistyp \rangle$
| $\langle Datentyp \rangle$
| `(` $\langle TypKonstruktor \rangle$ $\langle Typ \rangle^+$ `)`
| `(` $\langle Typ \rangle^+$ `->` $\langle Typ \rangle$ `)`
| $\langle X \rangle$
| `[` $\langle X \rangle^+$ `]` $\langle Typ \rangle$

$\langle Basistyp \rangle$ ::= `Number`

| | `String`
| | `Boolean`
| | `Image`
⟨*Datentyp*⟩ ::= `Posn`
| | `WorldState`
| | ...
⟨*TypKonstruktor*⟩ ::= `list-of`
| | `tree-of`
| | ...
⟨*X*⟩ ::= `X`
| | `Y`
| | ...

Wir haben bisher nicht gesagt, wie wir Funktionstypen interpretieren sollen. Im Moment können wir es als die Menge aller Funktionen, die diesem Typ genügen, interpretieren. Später werden wir dies noch präzisieren.

14.5 Lokale Abstraktion

Häufig benötigt man Konstanten, Funktionen oder Strukturen nur *lokal*, innerhalb einer Funktion. Für diesen Fall gibt es die Möglichkeit, Definitionen lokal zu definieren, und zwar mit `local` Ausdrücken.

14.5.1 Lokale Funktionen

Betrachten wir als erstes lokale Funktionen. Hier ein Beispiel:

Beachten Sie, dass `foldr` der Name der eingebauten Funktion ist, die unserem `op-elements` entspricht.

```
; (list-of String) -> String
; appends all strings in l with blank space between elements
(check-expect (append-all-strings-with-
space (list "a" "b" "c")) " a b c ")
(define (append-all-strings-with-space l)
  (local (; String String -> String
          ; juxtapoint two strings and prefix with space
          (define (string-append-with-space s t)
            (string-append " " s t)))
    (foldr string-append-with-space
           " "
           l)))
```

) Die Funktion `string-append-with-space` ist eine *lokale* Funktion, die nur innerhalb der Funktion `append-all-strings-with-space` sichtbar ist. Sie kann außerhalb von `append-all-strings-with-space` nicht aufgerufen werden.

Da `local` Ausdrücke Ausdrücke sind, können sie überall stehen, wo ein Ausdruck erwartet wird. Häufig werden als äußerster Ausdruck eines Funktionsbodies verwendet, aber sie können auch an jeder anderen Stelle stehen.

Diese Ausdrücke sind daher äquivalent:

```
> (local [(define (f x) (+ x 1))] (+ (* (f 5) 6) 7))
43

> (+ (local [(define (f x) (+ x 1))] (* (f 5) 6)) 7)
43
```

Ein Extremfall ist der, die lokale Definition so weit nach innen zu ziehen, dass sie direkt an der Stelle steht, wo sie verwendet wird.

```
> (+ (* ( (local [(define (f x) (+ x 1))] f) 5) 6) 7)
43
```

Wir werden später sehen, dass es für diesen Fall noch eine bessere Notation gibt.

14.5.2 Lokale Konstanten

Nicht nur Funktionen können lokal sein, sondern auch Konstanten und Strukturdefinitionen.

Betrachten wir beispielsweise eine Funktion zum Exponentieren von Zahlen mit der Potenz 8:

Alternativ hätten wir im Body die Kurzschreibweise `(* x x x x x x x x)` verwenden können.

```
(define (power8 x)
  (* x (* x (* x (* x (* x (* x (* x (* x x))))))))
```

Zum Beispiel:

```
> (power8 2)
256
```

Diese Funktion benötigt acht Multiplikationen zum Berechnen des Ergebnis. Eine effizientere Methode zum Berechnen der Potenz nutzt die Eigenschaft der Exponentialfunktion, dass $a^{2*b} = a^b * a^b$.

```
(define (power8-fast x)
  (local
    [(define r1 (* x x))
     (define r2 (* r1 r1))
     (define r3 (* r2 r2))]
        r3))
```

Mit Hilfe einer Sequenz lokaler Konstantendeklarationen können wir also die Zwischenergebnisse repräsentieren und mit nur 3 Multiplikationen das Ergebnis berechnen. Diese Konstantendeklarationen könnte man nicht durch (globale) Konstantendeklarationen ersetzen, denn ihr Wert hängt von dem Funktionsparameter ab.

Im Allgemeinen verwendet man lokale Konstanten aus zwei Gründen: 1) Um Redundanz zu vermeiden, oder 2) um Zwischenergebnissen einen Namen zu geben. Auf den ersten Punkt kommen wir gleich nochmal zu sprechen; hier erstmal ein Beispiel für den zweiten Fall:

Im Abschnitt §6.5 “Fallstudie: Ein Ball in Bewegung” haben wir eine Funktion programmiert, die einen Bewegungsvektor zu einer Position addiert:

```
(define (posn+vel p q)
  (make-posn (+ (posn-x p) (vel-delta-x q))
             (+ (posn-y p) (vel-delta-y q))))
```

Mit Hilfe lokaler Konstanten können wir den Zwischenergebnissen für die neue x- und y-Koordinate einen Namen geben und deren Berechnung von der Konstruktion der neuen Position trennen:

```
(define (posn+vel p q)
  (local [(define new-x (+ (posn-x p) (vel-delta-x q)))
          (define new-y (+ (posn-y p) (vel-delta-y q)))]
    (make-posn new-x new-y)))
```

Es gibt sogar Programmiersprachen und Programmierstile, in denen man *jedem* Zwischenergebnis einen Namen geben muss. Wenn Sie dies interessiert, recherchieren Sie was *three address code*, *administrative normal form* oder *continuation-passing style* ist.

Das Einführen von Namen für Zwischenergebnisse kann helfen, Code leichter lesbar zu machen, weil der Name hilft, zu verstehen, was das Zwischenergebnis repräsentiert. Es kann auch dazu dienen, eine sehr tiefe Verschachtelung von Ausdrücken flacher zu machen.

Die andere wichtige Motivation für lokale Konstanten ist die Vermeidung von Redundanz. Tatsächlich können wir mit lokalen Konstanten sogar zwei unterschiedliche Arten von Redundanz vermeiden: Statische Redundanz und dynamische Redundanz.

Mit statischer Redundanz ist unser DRY Prinzip gemeint: Wir verwenden lokale Konstanten, um uns nicht im Programtext wiederholen zu müssen. Dies illustriert unser erstes Beispiel. Illustrieren wir dies, indem wir unsere `power8-fast` Funktion wieder de-optimieren indem wir alle Vorkommen der Konstanten durch ihre Definition ersetzen:

```
(define (power8-fast-slow x)
  (local
    [(define r1 (* x x))
     (define r2 (* (* x x) (* x x)))
     (define r3 (* (* (* x x) (* x x)) (* (* x x) (* x x))))]
        r3))
```

Diese Funktion ist, bis auf die Assoziativität der Multiplikationen, äquivalent zu `power-8` von oben. Offensichtlich ist hier aber das DRY-Prinzip verletzt, weil die Unterausdrücke `(* x x)` und `(* (* x x) (* x x))` mehrfach vorkommen. Diese Redundanz wird durch die Verwendung der lokalen Konstanten in `power8-fast` vermieden.

Die zweite Facette der Redundanz, die wir durch lokale Konstanten vermeiden können, ist die dynamische Redundanz. Damit ist gemeint, dass wir das mehrfache Auswerten eines Ausdrucks vermeiden können. Dies liegt daran, dass der Wert einer lokale Konstante nur einmal bei ihrer Definition berechnet wird und im folgenden nur das bereits berechnete Ergebnis verwendet wird. Im Beispiel `power8-fast` haben wir gesehen, dass wir dadurch die Anzahl der Multiplikationen von 8 auf 3 senken konnten. Im Allgemeinen "lohnt" sich die Definition einer lokalen le aus Sicht der dynamischen Redundanz dann, wenn sie mehr als einmal ausgewertet wird.

Zusammengefasst haben lokale Konstanten also den gleichen Zweck wie nicht-lokale (globale) Konstanten, nämlich der Benennung von Konstanten bzw. Zwischenergebnissen und der Vermeidung von statischer und dynamischer Redundanz; lokale Konstanten sind allerdings dadurch mächtiger, dass sie die aktuellen Funktionsparameter und andere lokale Definitionen verwenden können.

14.5.3 Intermezzo: Successive Squaring

Als fortgeschrittenes Beispiel für die Verwendung lokaler Konstanten können wir die Verallgemeinerung des Potenzierungsbeispiels auf beliebige Exponenten betrachten. Dieser Abschnitt kann übersprungen werden.

Um den Effekt der Vermeidung dynamischer Redundanz noch deutlicher zu illustrieren, betrachten wir die Verallgemeinerung von `power8` auf beliebige (natürlichzahlige) Exponenten. Wenn wir den Exponenten wie in Abschnitt §9.3.6 “Natürliche Zahlen als rekursive Datenstruktur” beschrieben als Instanz eines rekursiven Datentyps auffassen, so ergibt sich folgende Definition:

```
; NaturalNumber Number -> Number
(define (exponential n x)
  (if (zero? n)
      1
      (* x (exponential (sub1 n) x))))
```

Die Redundanz in diesem Programm ist nicht offensichtlich; sie wird erst dann offenbar, wenn man die "Ausfaltung" der Definition für einen festen Exponenten betrachtet; also die Version, die sich ergibt, wenn man die rekursiven Aufrufe für ein festgelegtes `n` expandiert. Beispielsweise ergibt das Ausfalten von `exponential` für den Fall `n` = 8 genau die `power8` Funktion von oben. Die Redundanz wird also erst dann offensichtlich, wenn man, unter Ausnutzung der Assoziativität der Multiplikation, `(* x (* x (* x (* x (* x (* x (* x (* x x)))))))` umformt zu `(* (* (* x x) (* x x)) (* (* x x) (* x x)))`.

Dieses "Ausfalten" ist ein Spezialfall einer allgemeineren Technik, die *partial evaluation* heißt.

Eine kleine Komplikation der Anwendung der Technik von oben auf den allgemeinen Fall ist, dass der Exponent im Allgemeinen keine Potenz von 2 ist. Wir gehen mit diesem Problem so um, dass wir die Fälle unterscheiden, ob der Exponent eine gerade oder eine ungerade Zahl ist. Insgesamt ergibt sich dieser Algorithmus, der in der Literatur auch als Exponentiation durch *successive squaring* bekannt ist.

```
(define (exponential-fast x n)
  (if (zero? n)
      1
      (local
        [(define y (exponential-fast x (quotient n 2)))
         (define z (* y y))]
         (if (odd? n) (* x z) z))))
```

Im `power8` Beispiel oben haben wir die Anzahl der benötigten Multiplikationen von 8 auf 3 reduziert. In dieser allgemeinen Version ist der Unterschied noch viel drastischer:

Überlegen Sie, wieso `exponential-fast` nur etwa $\log_2(n)$ Multiplikationen benötigt.

Die `exponential` Funktion benötigt `n` Multiplikationen während `exponential-fast` nur etwa $\log_2(n)$ Multiplikationen benötigt. Das ist ein riesiger Unterschied. Beispielsweise benötigt auf meinem Rechner die Auswertung von `(exponential 2 100000)` fast eine Sekunde, während `(exponential-fast 2 100000)` weniger als eine Millisekunde benötigt. Probieren Sie es selbst aus. Sie können beispielsweise die `time` Funktion zur Messung der Performance verwenden.

14.5.4 Lokale Strukturen

Es ist möglich, auch Strukturen mit `define-struct` lokal zu definieren. Wir werden diese Möglichkeit bis auf weiteres nicht verwenden.

14.5.5 Scope lokaler Definitionen

Der *Scope* einer Definition eines Namens ist der Bereich im Programm, in dem sich eine Verwendung des Namens auf diese Definition bezieht. Unserer Programmiersprache verwendet *lexikalisches Scoping*, auch bekannt als *statisches Scoping*. Dies bedeutet, dass der Scope einer lokalen Definition die Unterausdrücke des `local` Ausdrucks sind.

In diesem Beispiel sind alle Verwendungen von `f` und `x` im Scope ihrer Definitionen.

```
(local [(define (f n) (if (zero? n)
                          0
                          (+ x (f (sub1 n)))))
        (define x 5)]
  (+ x (f 2)))
```

In diesem Ausdruck hingegen ist die zweite Verwendung von `x` nicht im Scope der Definition:

```
(+ (local [(define x 5)] (+ x 3))
   x)
```

Es kann vorkommen, dass es mehrere Definitionen eines Namens (Konstanten, Funktion, Strukturkonstruktoren/-selektoren) gibt, die einen überlappenden Scope haben. In diesem Fall bezieht sich eine Verwendung des Namens immer auf die syntaktisch näheste Definition. Wenn man also im abstrakten Syntaxbaums eines Programms nach außen geht, so "gewinnt" die erste Definition, die man auf dem Weg von dem Auftreten eines Namens zur Wurzel des Baums trifft.

Beispiel: Kopieren Sie das folgende Programm in den Definitionsbereich von DrRacket und drücken auf "Syntaxprüfung". Gehen Sie nun mit dem Mauszeiger über eine Definition oder Benutzung eines Funktions- oder Konstantennamens. Die Pfeile, die Sie nun angezeigt bekommen, illustrieren, auf welche Definition sich ein Name bezieht.

```
(add1 (local
```

```
[(define (f y)
        (local [(define x 2)
                (define (g y) (+ y x))]
          (g y)))
      (define (add1 x) (sub1 x))]
(f (add1 2))))
```

14.6 Funktionen als Werte: Closures

Betrachten Sie folgende Funktion zur numerischen Berechnung der Ableitung einer Funktion:

```
; (Number -> Number) Number -> Number
(define (derivative f x)
  (/ (- (f (+ x 0.001)) (f x))
     0.001))
```

Diese Funktion gibt uns die (numerische) Ableitung einer Funktion an einem speziellen Punkt.

Aber es wäre eigentlich besser, wenn uns diese Funktion die Ableitung selber als Funktion als Ergebnis liefert. Wenn wir zum Beispiel eine Funktion

```
; (Number -> Number) -> Image
(define (plot-function f) ...)
```

hätten, so könnten wir die Ableitung einer Funktion damit nicht zeichnen. Eigentlich möchten wir, dass `derivative` die Signatur

```
; (Number -> Number) -> (Number -> Number)
(define (derivative f) ...)
```

hat. Aber wie können wir diese Version von `derivative` implementieren? Offensichtlich kann die Funktion, die wir zurückgeben, keine global definierte Funktion sein, denn diese Funktion hängt ja von dem Parameter `f` ab. Wir können jedoch diese Funktion *lokal* definieren (und auch gleich etwas besser strukturieren), und dann diese Funktion zurückgeben:

```
; (Number -> Number) -> (Number -> Number)
(define (derivative f)
  (local
    [(define delta-x 0.001)
     (define (delta-f-x x) (- (f (+ x delta-x)) (f x)))
     (define (g x) (/ (delta-f-x x) delta-x))]
    g))
```

Was für eine Art von Wert ist es, der von `derivative` zurückgegeben wird?

Nehmen wir es an, es wäre sowas wie die Definition von `g`, also `(define (g x) (/ (delta-f-x x) delta-x))`. Wenn dem so wäre, was passiert mit der Konstanten `delta-x` und der Funktion `delta-f-x`? Diese sind ja in `derivative` nur lokal gebunden. Was passiert beispielsweise bei der Auswertung von diesem Ausdruck:

```
(local [(define f (derivative exp))
        (define delta-x 10000)]
  (f 5))
```

Wird die `delta-x` Konstante während der Auswertung von `(f 5)` etwa zu `10000` ausgewertet? Das wäre ein Verstoß gegen lexikalisches Scoping und würde zu unvorhersehbaren Interaktionen zwischen unterschiedlichen Programmteilen führen. Offensichtlich sollte `delta-x` in der zurückgegebenen Funktion sich immer auf die lokale Definition beziehen.

Wenn wir eine Funktion als Wert behandeln, so ist dieser Wert mehr als nur die Definition der Funktion. Es ist nämlich die Definition der Funktion *und* die Menge der lokal definierten Namen (Konstanten, Funktionen, Strukturen, Funktionsparameter). Diese Kombination aus Funktionsdefinition und lokaler *Umgebung* nennt man *Funktionsabschluss* oder *Closure*. Die Auswertung einer Funktion (nicht eines Funktionsaufrufs) ergibt also einen Closure. Wird dieser Closure irgendwann später wieder im Rahmen eines Funktionsaufrufs angewendet, so wird die gespeicherte lokale Umgebung wieder aktiviert. Später werden wir das Konzept des Closure noch präzisieren; im Moment merken wir uns, dass die Auswertung einer Funktion die Funktionsdefinition plus ihre lokale Umgebung ergibt, und diese lokale Umgebung bei Anwendung des Closure verwendet wird, um Namen im Funktionsbody auszuwerten.

14.7 Lambda, die ultimative Abstraktion

Nehmen sie an, sie möchten mit Hilfe der `map` Funktion alle Elemente in einer Liste von Zahlen `lon` verdoppeln. Dies könnten Sie wie folgt anstellen:

```
(local [(define (double x) (* 2 x))]
  (map double lon))
```

Dieser Ausdruck ist komplizierter als es ein müsste. Für eine so einfache Funktion wie `double`, die nur lokal verwendet wird, ist es Verschwendung, einen Namen zu vergeben und eine komplette Extra Zeile Code zu verwenden.

In ISL+ (dem Sprachlevel von HTDP welches wir zurzeit verwenden) gibt es aus diesem Grund die Möglichkeit, *anonyme* Funktionen, also Funktionen ohne Namen, zu definieren. Anonyme Funktionen werden mit dem Schlüsselwort `lambda` gekennzeichnet. Daher nennt man solche Funktionen auch `lambda`-Ausdrücke.

Hier ist das Beispiel von oben, aber so umgeschrieben, dass statt `double` eine anonyme Funktion definiert wird.

```
(map (lambda (x) (* 2 x)) lon)
```

Ein `lambda` Ausdruck hat im Allgemeinen die Form `(lambda (x-1 ... x-n) exp)` für Namen `x-1`,...,`x-n` und einen Ausdruck `exp`. Ihre Bedeutung entspricht in etwa einer lokalen Funktionsdefinition der Form `(local [(define (f x-1 ... x-n) exp)] f)`. Beispielsweise hätten wir den Ausdruck von oben auch so schreiben können:

```
(map (local [(define (double x) (* 2 x))] double) lon)
```

Dies ist kein exaktes "Desugaring". Lokale Funktionen können rekursiv sein; `lambda`-Ausdrücke nicht. Allerdings trägt `lambda` sehr wohl zur Vereinfachung der Sprache bei, denn wir können nun Funktionsdefinitionen "desugaren" zu Konstantendefinitionen:

Man kann allerdings mit Hilfe sogenannter *Fixpunktkombinatoren* Rekursion mit `lambda`-Ausdrücken simulieren.

```
(define (f x-1 ... x-n) exp)
```

entspricht

```
(define f (lambda (x-1 ... x-n) exp))
```

Die `lambda`-Ausdrücke machen also sehr deutlich, dass Funktionen "ganz normale" Werte sind, an die wir Konstanten binden können. Wenn wir also bei obiger Definition von `f` einen Funktionsaufruf `(f e-1 ... e-n)` haben, so ist das `f` an der ersten Position kein Funktionsname, sondern ein Konstantenname, der zu einem λ-Ausdruck ausgewertet wird. Im Allgemeinen hat ein Funktionsaufruf also die Syntax `(exp-0 exp-1 ... exp-n)`, wobei alle `exp-i` beliebige Ausdrücke sind aber `exp-0` bei der Auswertung eine Funktion (genauer: ein Closure) ergeben muss.

Das Wort `lambda`-Ausdruck stammt aus dem λ-Kalkül, welches in den 1930er Jahren von Alonzo Church entwickelt wurde. Das λ-Kalkül ist eine Untersprache von ISL+: Im λ-Kalkül gibt es nur λ-Ausdrücke und Funktionsapplikationen — keine Zahlen, keine Wahrheitswerte, keine Konstantendefinitionen, keine Listen oder Strukturen, und so weiter. Dennoch sind λ-Ausdrücke so mächtig, dass man sich all diese Programmiersprachenfeatures innerhalb des λ-Kalkül nachbauen kann, beispielsweise mit Hilfe sogenannter *Church-Kodierungen*.

Übrigens können Sie statt des ausgeschriebenen Wortes `lambda` auch direkt den griechischen Buchstaben λ im Programmtext verwenden. Sie können also auch schreiben:

Schauen Sie in DrRacket unter "Einfügen", mit welcher Tastenkombination sie den Buchstaben λ in ihr Programm einfügen können.

```
(map (λ (x) (* 2 x)) lon)
```

14.8 Wieso abstrahieren?

Programme sind wie Bücher: Sie werden für Menschen (Programmierer) geschrieben und können halt nebenbei auch noch auf einem Computer ausgeführt werden. Auf jeden Fall sollten Programme, genau wie Bücher, keine unnötigen Wiederholungen enthalten, denn niemand möchte solche Programme lesen.

Die Einhaltung des DRY Prinzips durch die Erschaffung guter Abstraktionen hat viele Vorteile. Bei Programmen mit wiederkehrenden Mustern besteht stehts die Gefahr der Inkonsistenz, da man an jeder Stelle, an der das Muster wieder auftritt, dieses Muster wieder korrekt nachbilden muss. Wir haben auch gesehen, dass es zu nicht unerheblicher Vergrößerung des Codes führen kann, wenn man keine guten Abstraktionen hat und sich oft wiederholt.

Der wichtigste Vorteil guter Abstraktionen ist jedoch folgende: Es gibt für jede kohärente Funktionalität des Programms genau eine Stelle, an der diese implementiert ist. Diese Eigenschaft macht es viel einfacher, ein Programm zu schreiben und zu warten.

Wenn man einen Fehler gemacht hat, ist der Fehler an einer Stelle lokalisiert und nicht vielfach dupliziert. Wenn man die Funktionalität ändern möchte, gibt es eine Stelle, an der man etwas ändern muss und man muss nicht alle Vorkommen eines Musters finden (was schwierig oder praktisch unmöglich sein kann). Wichtige Eigenschaften, wie Terminierung oder Korrektheit, können Sie einmal für die Abstraktion nachweisen und sie gilt dann automatisch für alle Verwendungen davon.

Diese Vorteile treffen auf *alle* Arten der Abstraktion zu, die wir bisher kennengelernt haben: Globale und lokale Funktions- und Konstantendefinitionen, Abstrakte Typsignaturen, Abstrakte Datendefinitionen.

Aus diesem Grund formulieren wir folgende Richtlinie als Präzisierung des DRY-Prinzips:

Definieren Sie eine Abstraktion statt einen Teil eines Programms zu kopieren und dann zu modifizieren.

Diese Richtlinie gilt nicht nur während der ersten Programmierung eines Programms. Auch in der Weiterentwicklung und Wartung von Programmen sollten Sie stets darauf achten, ob es in ihrem Programm Verstöße gegen dieses Prinzip gilt und diese Verstöße durch die Definition geeigneter Abstraktionen eliminieren.

15 Bedeutung von ISL+

In diesem Kapitel werden wir die im Vergleich zur letzten formalen Sprachdefinition aus §8 “Bedeutung von BSL” neu hinzugekommen Sprachfeatures präzise definieren. Unsere Methodik bleibt die gleiche. Wir definieren zunächst die Semantik einer Kern-Sprache, die alle wesentlichen Sprachmerkmale enthält. Dann definieren wir eine Auswertungsregel für Programme und Definitionen sowie eine Reduktionsrelation $\rightarrow$, mit der Programme Schritt für Schritt ausgewertet werden können.

15.1 Syntax von Core-ISL+

Die beiden neu hinzugekommenen Sprachfeatures in ISL sind 1) lokale Definitionen und 2) die Möglichkeit, Funktionen als Werte zu betrachten. Um die Definitionen möglichst einfach zu halten, werden wir zunächst mal Sprachfeatures entfernen, die wir bereits in §8 “Bedeutung von BSL” präzise definiert haben und die durch die Hinzunahme der neuen Features nicht beeinflusst werden: Strukturdefinitionen (`define-struct`), konditionale Ausdrücke (`cond`) und logische Operatoren (`and`).

Desweiteren ergeben sich durch die uniforme Behandlung von Funktionen und Werten noch weitere Vereinfachungsmöglichkeiten. Es ist nicht nötig, Konstantendefinitionen und Funktionsdefinitionen zu unterstützen, denn jede Funktionsdefinition `(define (f x) e)` kann durch eine Konstantendefinition `(define f (lambda (x) e))` ausgedrückt werden. Insgesamt ergibt sich damit folgende Syntax für die Kernsprache:

$\langle$*definition*$\rangle$ ::= `(` `define` $\langle$*name*$\rangle$ $\langle$*e*$\rangle$ `)`
$\langle$*e*$\rangle$ ::= `(` $\langle$*e*$\rangle$ $\langle$*e*$\rangle^+$ `)`
| `(` `local` `[` $\langle$*definition*$\rangle^+$ `]` $\langle$*e*$\rangle$ `)`
| $\langle$*name*$\rangle$
| $\langle$*v*$\rangle$
$\langle$*v*$\rangle$ ::= `(` `lambda` `(` $\langle$*name*$\rangle^+$ `)` $\langle$*e*$\rangle$ `)`
| $\langle$*number*$\rangle$
| $\langle$*boolean*$\rangle$
| $\langle$*string*$\rangle$
| $\langle$*image*$\rangle$
| $\langle$+$\rangle$
| $\langle$*$\rangle$
| $\langle$...$\rangle$

]

15.2 Werte und Umgebungen

Auch in ISL findet die Auswertung von Programmen immer in einer Umgebung statt. Da es in der Kernsprache keine Struktur- und Funktionsdefinitionen mehr gibt, ist eine Umgebung nun nur noch eine Sequenz von Konstantendefinitionen.

$\langle$*env*$\rangle$ ::= $\langle$*env-element*$\rangle$*
$\langle$*env-element*$\rangle$ ::= `(` `define` $\langle$*name*$\rangle$ $\langle$*v*$\rangle$ `)`

15.3 Bedeutung von Programmen

Die *(PROG)* Regel aus §8.8 “Bedeutung von Programmen” bleibt für ISL nahezu unverändert erhalten (ohne die Teile für die Auswertung von Struktur- und Funktionsdefinitionen):

(PROG): Ein Programm wird von links nach rechts ausgeführt und startet mit der leeren Umgebung. Ist das nächste Programmelement ein Ausdruck, so wird dieser gemäß der unten stehenden Regeln in der aktuellen Umgebung zu einem Wert ausgewert. Ist das nächste Programmelement eine Konstantendefinition `(define x e)`, so wird, sofern `e` nicht bereits ein Wert ist, in der aktuellen Umgebung zunächst `e` zu einem Wert `v` ausgewertet und dann `(define x v)` zur aktuellen Umgebung hinzugefügt (indem es an das Ende der Umgebung angehängt wird).

Allerdings gibt es einen wichtigen Unterschied, nämlich den, dass sich durch die unten stehende *(LOCAL)* Regel der Rest des noch auszuwertenden Programms während der Auswertung ändern kann.

15.4 Auswertungspositionen und die Kongruenzregel

Der Auswertungskontext legt fest, dass Funktionsaufrufe von links nach rechts ausgewertet werden, wobei im Unterschied zu BSL nun auch die aufzurufende Funktion ein Ausdruck ist, der ausgewertet werden muss.

⟨*E*⟩ `::=` `[]`
`|` `(` ⟨*v*⟩* ⟨*E*⟩ ⟨*e*⟩* `)`

Die Standard Kongruenzregel gilt auch in ISL.

(KONG): Falls *e-1* → *e-2*, dann *E[e-1]* → *E[e-2]*.

15.5 Bedeutung von Ausdrücken

15.5.1 Bedeutung von Funktionsaufrufen

Die Auswertung von Funktionsaufrufen ändert sich dadurch, dass sich die Funktion, die aufgerufen wird, sich im Allgemeinen erst während der Funktionsauswertung ergibt.

(APP): `(` `(` `lambda` `(` *name-1 ... name-n* `)` *e* `)` *v-1 ... v-n* `)` → *e*[*name-1* := *v-1 ... name-n* := *v-n*]

Die Ersetzung der formalen Argumente durch die tatsächlichen Argumente in dieser Regel ist komplexer als es zunächst den Anschein hat. Insbesondere darf ein Name wie *name-1* nicht durch *v-1* ersetzt werden, wenn *name-1* in einem Kontext vorkommt, in dem es eine lexikalisch nähere andere Bindung (durch `lambda` oder `local`) von *name-1* gibt. Beispiel: `((lambda (x) (+ x 1)) x)`[x := 7] = `((lambda (x) (+ x 1)) 7)` und nicht `((lambda (x) (+ 7 1)) 7)`. Wir werden hierauf in unserer Diskussion zu Scope in §15.6 “Scope” zurückkommen.

(PRIM): Falls *v* eine primitive Funktion *f* ist und *f(v-1,...,v-n)=v*,
dann `(` *v v-1 ... v-n* `)` → *v*.

Auch primitive Funktionen können das Resultat von Berechnungen sein; beispielsweise hängt im Ausdruck `((if cond + *) 3 4)` die Funktion, die verwendet wird, vom Wahrheitswert des Ausdrucks `cond` ab.

15.5.2 Bedeutung von lokalen Definitionen

Die komplexeste Regel ist die zur Bedeutung von lokalen Definitionen. Sie verwendet einen wie oben definierten Auswertungskontext E um aus lokalen Definitionen globale Definitionen zu machen. Um Namenskollisionen zu vermeiden, werden lokale Definitionen ggf. umbenannt. Abhängigkeiten von lokalen Definitionen vom lokalen Kontext (zum Beispiel einem Funktionsargument) wurden ggf. durch vorhergehende Substitutionen beseitigt.

(LOCAL): E[`(` `local` `[` `(` `define` *name-1 e-1* `)` ... `(` `define` *name-n e-n* `)` `]` *e* `)`] → `(` `define` *name-1′ e-1′* `)` ... `(` `define` *name-n′ e-n′* `)` E[*e′*] wobei *name-1′*,...,*name-n′* "frische" Namen sind die sonst nirgendwo im Programm vorkommen und *e′*,*e-1′*,...,*e-n′* Kopien von *e*, *e-1*,...,*e-n′* sind, in denen alle Vorkommen von *name-1*,...,*name-n* durch *name-1′*,...,*name-n′* ersetzt werden.

Die grammatikalisch nicht ganz korrekte Notation `(` `define` *name-1′ e-1′* `)` ... `(` `define` *name-n′ e-n′* `)` E[*e′*] soll hierbei bedeuten, dass E[`(` `local` ... *e* `)`] ersetzt wird durch E[*e*] und gleichzeitig die Definitionen `(` `define` *name-1′ e-1′* `)` ... `(` `define` *name-n′ e-n′* `)` als nächste mittels *(PROG)* auszuwertende Definition in das Programm aufgenommen werden.

Beispiel:

```
(define f (lambda (x)
            (+ 2
               (local
                 [(define y (+ x 1))]
                 (* y 2)))))
(f 2)
```

Dann

```
(f 2)
```

→

```
(+ 2
   (local
     [(define y (+ 2 1))]
     (* y 2)))
```

→

```
(define y_0 (+ 2 1))
(+ 2 (* y_0 2))
```

In diesem zweiten Schritt wurde die *(LOCAL)* verwendet, um aus der lokalen Definition eine globale Definition zu machen. Die Abhängkeit vom lokalen Kontext (nämlich dem Funktionsargument `x`) wurde zuvor im ersten Schritt durch eine Verwendung der *(APP)* Regel beseitigt. Die Auswertung setzt sich nun durch Verwendung der *(PROG)* fort, also wir werten durch `(+ 2 1)` → `3` die Konstantendefinition aus, fügen `(define y_0 3)` zur Umgebung hinzu, und werten nun in dieser Umgebung `(+ 2 (* y_0 2))` zum Ergebnis `8` aus.

15.5.3 Bedeutung von Konstanten

Die Definition von Konstanten ist gegenüber BSL unverändert.

(CONST): *name* → *v*, falls `(` `define` *name* *v* `)` die letzte Definition von *name* in *env* ist.

15.6 Scope

Wir wollen unter der Berücksichtigung der lokalen Definitionen nochmal die Diskussion über Scope aus §14.5.5 “Scope lokaler Definitionen” aufgreifen und diskutieren, wie die formale Definition lexikalisches Scoping garantiert. Lexikalisches Scoping ist in zwei der Regeln oben sichtbar: *(APP)* und *(LOCAL)*.

In *(LOCAL)* findet eine Umbenennung statt: Der Name von lokalen Konstanten wird umbenannt und alle Verwendungen des Namens in den Unterausdrücken des `local` Ausdrucks werden ebenfalls umbenannt. Dadurch, dass diese Umbenennung genau in den Unterausdrücken vollzogen wird, wird lexikalisches Scoping sichergestellt. Dadurch, dass ein "frischer" Name verwendet wird, kann keine Benutzung des Namens ausserhalb dieser Unterausdrücke an die umbenannte Definition gebunden werden.

Das gleiche Verhalten findet sich in *(APP)*: Dadurch, dass die formalen Parameter nur im Body der Funktion durch die aktuellen Argumente ersetzt werden, wird lexikalisches Scoping sichergestellt. Gleichzeitig wird hierdurch definiert, wie Closures repräsentiert werden, nämlich als Funktionsdefinitionen, in denen die "weiter aussen" gebundenen Namen bereits durch Werte ersetzt wurden.

Beispiel: Der Ausdruck `(f 3)` in diesem Programm

```
(define (f x)
  (lambda (y) (+ x y)))
(f 3)
```

wird reduziert zu `(lambda (y) (+ 3 y))`; der Wert für `x` wird also im Closure mit gespeichert.

Ein wichtiger Aspekt von lexikalischem Scoping ist *Shadowing*. Shadowing ist eine Strategie, mit der Situation umzugehen, dass gleichzeitig mehrere Definitionen eines Namens im Scope sind.

Beispiel:

```
(define x 1)
(define (f x)
  (+ x (local [(define x 2)] (+ x 1))))
```

In diesem Beispiel gibt es drei *bindende* Vorkommen von `x` und zwei *gebundene* Vorkommen von `x`. Das linke Vorkommen von `x` in der letzten Zeile des Beispiels ist im lexikalischen Scope von zwei der drei Definitionen; das rechte Vorkommen von `x` ist sogar im lexikalischen Scope aller drei Definitionen.

Shadowing besagt, dass in solchen Situationen stets die lexikalisch "nächste" Definition "gewinnt". Mit "nächste" ist die Definition gemeint, die man als erstes antrifft,

wenn man in der grammatikalischen Struktur des Programmtextes von dem Namen nach außen geht. Die weiter innen stehenden Definitionen überdecken also die weiter außen stehenden Definitionen: Sie werfen einen Schatten ("shadow"), in dem die aussen stehende Definition nicht sichtbar ist. Daher wird in dem Beispiel oben beispielsweise der Ausdruck `(f 3)` zu `6` ausgewertet.

Shadowing rechtfertigt sich aus einer Modularitätsüberlegung: Die Bedeutung eines Ausdrucks sollte möglichst lokal ablesbar sein. Insbesondere sollte ein Ausdruck, der keine ungebundenen Namen enthält (ein sogenannter "geschlossener" Term), überall die gleiche Bedeutung haben, egal wo man ihn einsetzt. Beispielsweise sollte der Ausdruck `(lambda (x) x)` immer die Identitätsfunktion sein und nicht beispielsweise die konstante `3` Funktion nur weil weiter außen irgendwo `(define x 3)` steht. Dieses erwünschte Verhalten kann nur durch lexikalisches Scoping mit Shadowing garantiert werden.

Programmiersprachen, die es erlauben, lokal Namen zu definieren und lexikalisches Scoping mit Shadowing verwenden, nennt man auch Programmiersprachen mit *Blockstruktur*. Blockstruktur war eine der großen Innovationen in der Programmiersprache ALGOL 60. Die meisten modernen Programmiersprachen heute haben Blockstruktur.

16 Generative Rekursion

Häufig richtet sich die Struktur von Funktionen nach der Struktur der Daten, auf denen die Funktionen arbeiten. Beispielsweise sieht unser Entwurfsrezept für algebraische Datentypen vor, für jede Alternative des Datentyps eine Hilfsfunktion zu definieren. Wenn wir einen rekursiven Datentyp haben, so sieht unser Entwurfsrezept den Einsatz struktureller Rekursion vor.

In manchen Fällen jedoch muss man von dieser Parallelität von Daten und Funktionen abweichen: Die Struktur der Daten passt nicht zu der Art und Weise, wie das Problem in Teilprobleme aufgeteilt werden soll.

16.1 Wo kollidiert der Ball?

Betrachten Sie als Beispiel die Fallstudie zum Ball in Bewegung in §6.5 “Fallstudie: Ein Ball in Bewegung”. Nehmen wir an, wir möchten eine Funktion, die zu einem Ball berechnet, an welcher Position der Ball das erste Mal eine Kollision mit der Wand hat. Wenn wir einen Ball `ball` haben, so können wir durch Aufruf von `(move-ball ball)`, `(move-ball (move-ball ball))` usw. die Bewegung des Balls simulieren. Wie lange wollen wir diese Simulation durchführen? So lange, bis es eine Kolllision gibt, also bis `(collision current-ball)` nicht `"none"` ist.

Dies rechtfertigt die folgende Definition:

```
; Ball -> Posn
; computes the position where the first collision of the ball
occurs
(define (first-collision ball)
  (cond [(string=? (collision (ball-loc ball)) "none")
         (first-collision (move-ball ball))]
        [else (ball-loc ball)]))
```

Wenn wir uns diese Definition anschauen, so stellen wir zwei Besonderheiten fest: 1) Die Fallunterscheidung im Body der Funktion hat nichts mit der Struktur der Eingabe zu tun. 2) Das Argument, welches wir im rekursiven Funktionsaufruf übergeben, ist kein Teil der ursprünglichen Eingabe. Stattdessen generiert `(move-ball ball)` einen völlig neuen Ball, nämlich den Ball der um einen Schritt vorgerückt ist. Offensichtlich ist es nicht möglich, mit unserem bisherigen Entwurfsrezept eine Funktion dieser Art zu generieren.

16.2 Schnelles Sortieren

Betrachten Sie das Problem, eine Liste von Zahlen zu sortieren. Eine Verwendung unseres Entwurfsrezepts ergibt folgendes Template:

```
; (listof number)  ->  (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (sort l)
```

```
(match l
  [(list) ...]
  [(cons x xs) ... x ... (sort xs) ...]))
```

Im Basisfall ist die Vervollständigung des Templates trivial. Im rekursiven Fall müssen wir offensichtlich `x` in die bereits sortierte Liste `(sort xs)` einfügen. Hierzu könne wir die bereits in §14.3 “Funktionen als Funktionsparameter” definierte Funktion `insert` verwenden. Insgesamt ergibt sich damit die folgende Definition:

```
; (listof number)  ->  (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (sort l)
  (match l
    [(list) (list)]
    [(cons x xs) (insert x (sort xs))]))
```

Dieser Algorithmus, den man auch *insertion sort* nennt, ergibt sich zwangsläufig, wenn man mittels struktureller Rekursion eine Liste sortieren möchte. Allerdings ist dieser Algorithmus nicht sehr effizient. Wenn wir eine Liste `(list x-1 ... x-n)` haben, so ergibt die Expansion von `(sort (list x-1 ... x-n))` den Ausdruck `(insert x-1 (insert x-2 ... (insert x-n empty) ...))`. Im schlechtesten Fall (beispielsweise einer rückwärts sortierten Liste) benötigt `insert` so viele Berechungsschritte, wie die Liste lang ist. Da n + (n-1) + ... + 1 = n*(n+1)/2, ergibt sich, dass die Anzahl der Berechnungsschritte des Sortieralgorithmus im schlechtesten Fall quadratisch mit der Länge der Eingabe wächst.

Ein besserer Algorithmus ergibt sich, wenn wir das Problem geschickter in Teilprobleme zerlegen, als die Struktur der Daten dies suggeriert. Ein gängiger Algorithmus ist *quick sort*. Bei diesem Algorithmus wählen wir in jedem Schritt ein Element aus, beispielsweise das erste Listenelement. Dieses Element wird Pivot-Element genannt. Dann unterteilen wir den Rest der Liste in Listenelemente die kleiner (oder gleich) und solche die größer als das Pivot-Element sind. Wenn wir diese neu generierten Listen rekursiv sortieren, so können wir die Gesamtliste sortieren, indem wir die beiden sortierten Listen mit dem Pivot-Element in der Mitte aneinanderhängen.

Insgesamt ergibt sich damit folgende Definition:

```
; (listof number)  ->  (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (qsort l)
  (match l
    [(list) (list)]
    [(cons x xs)
     (append
      (qsort (smaller-or-equal-than x xs))
      (list x)
      (qsort (greater-than x xs)))]))
```

```
; Number (listof Number) -> (listof Number)
; computes a list of all elements of xs that are
; smaller or equal than x
(define (smaller-or-equal-than x xs)
  (filter (lambda (y) (<= y x)) xs))

; Number (listof Number) -> (listof Number)
; computes a list of all elements of xs that are
; greater than x
(define (greater-than x xs)
  (filter (lambda (y) (> y x)) xs))
```

Die Rekursionsstruktur in diesem Algorithmus unterscheidet sich ebenfalls deutlich von dem bekannten Muster der strukturellen Rekursion. Die Eingabe des rekursiven Aufrufs ist zwar in gewissem Sinne ein Teil der Eingabe (in dem Sinne dass die Listenelemente in den rekursiven Aufrufen eine Teilmenge der Listenelemente der Originaleingabe bilden), aber sie sind kein Teil der Eingabe im Sinne der Struktur der Eingabe. Die Fallunterscheidung in diesem Beispiel ist die gleiche, die wir auch bei struktureller Rekursion haben, aber statt einem rekursiven Aufruf wie bei struktureller Rekursion auf Listen haben wir zwei rekursive Aufrufe.

Es ist nicht ganz einfach, zu sehen, dass quick sort in der Regel viel schneller ist als insertion sort (und auch nicht Thema dieser Lehrveranstaltung), aber Sie können sehen, dass für den Fall, dass die beiden Listen `(smaller-or-equal-than x xs)` und `(greater-than x xs)` stets etwa gleich große Listen erzeugen, die Rekursionstiefe nur logarithmisch in der Länge der Liste ist. Man kann zeigen, dass die Anzahl der benötigten Rechenschritte zur Sortierung einer Liste der Länge n im Durchschnitt proportional zu n*log(n) ist.

16.3 Entwurf von generativ rekursiven Funktionen

Wir nennen Rekursionsstrukturen die nicht (notwendigerweise) dem Muster der strukturellen Rekursion entsprechen *generative Rekursion*. Eine generativ rekursive Rekursion hat eine Struktur, die etwa wie folgt aussieht:

```
(define (generative-recursive-fun problem)
      (cond
        [(trivially-solvable? problem)
         (determine-solution problem)]
        [else
         (combine-solutions
           ... problem ...
           (generative-recursive-fun (generate-problem-
1 problem))
           ...
           (generative-recursive-fun (generate-problem-
n problem)))]))
```

Dieses Template soll verdeutlichen, dass wir über beim Entwurf einer generativ rekursiven Funktion die folgenden fünf Fragen beantworten müssen:

1. Was ist ein trivial lösbares Problem?
2. Was ist die Lösung für ein trivial lösbares Problem?
3. Wie generieren wir neue Probleme, die leichter lösbar sind als das Originalproblem? Wie viele neue Probleme sollen wir generieren?
4. Wie berechnen wir aus den Lösungen der generierten Probleme die Lösung des Originalproblems? Benötigen wir hierzu nochmal das Originalproblem (oder einen Teil davon)?
5. Wieso terminiert der Algorithmus?

Die letzte Frage werden wir in nächsten Abschnitt separat behandeln. Die Antwort auf die ersten vier Fragen für das erste Beispiel oben lautet: 1) Ein Ball der bereits kollidiert. 2) Die aktuelle Position des Balles. 3) Indem wir einen Bewegungsschritt des Balles vornehmen (und ihn damit näher zur Kollisionsstelle bringen). 4) Die Lösung des generierten Problems ist die Lösung des Originalproblems; keine weitere Berechnung ist nötig.

Die Antwort auf die ersten vier Fragen für das zweite Beispiel lautet: 1) Das Sortieren einer leeren Liste. 2) Die leere Liste. 3) Indem wir ein Pivotelement auswählen und zwei neue Probleme generieren: Das Sortieren der Liste aller Elemente des Originalproblems, die kleiner (oder gleich) als das Pivotelement sind, und das Sortieren der Liste aller Elemente des Originalproblems, die größer als das Pivotelement sind. 4) Indem wir die beiden sortierten Listen mit dem Pivotelement in der Mitte zusammenhängen.

Eine generativ rekursive Funktion sollte in folgenden Situationen erwogen werden: 1) Die Eingabe hat eine rekursive Struktur, aber es ist nicht möglich oder zu kompliziert, mittels struktureller Rekursion das Problem zu lösen (beispielsweise weil das Ergebnis des rekursiven Aufrufs nicht hilft, das Originalproblem zu lösen). 2) Es gibt eine strukturell rekursive Funktion, die das Problem löst, aber sie ist nicht effizient genug. 3) Die Eingabe ist nicht rekursiv, aber die Anzahl der Berechnungsschritte zur Lösung des Problems ist nicht proportional zur Größe der Eingabe.

Wenn Sie ein Problem mittels generativer Rekursion lösen möchten, sollten Sie als erstes die vier Fragen oben beantworten und dann im Template-Schritt des Entwurfsrezepts das oben stehende Template verwenden (angepasst auf die Antworten auf Frage 1 und 3). Mit den Antworten auf die Frage 2 und 4 können Sie dann die Implementierung des Templates vervollständigen. Die Tests für die Funktion sollten auf jeden Fall sowohl Beispiele für triviale Probleme als auch für den generativ rekursiven Fall enthalten.

Ein wichtiger Unterschied zwischen struktureller Rekursion und generativer Rekursion ist, dass der Entwurf generativ rekursiver Funktionen mehr Kreativität erfordert. Inbesondere ist eine besondere gedankliche Einsicht erforderlich, wie aus dem Problem sinnvolle kleinere Teilprobleme generiert werden können. Bei struktureller Rekursion hingegen ergibt sich die Funktionsdefinition oft schon fast zwingend aus dem Template.

16.4 Terminierung

Eine wichtige Eigenschaft von strukturell rekursiven Funktionen ist, dass diese immer terminieren: Da die Eingabedaten eine endliche Größe haben und in jedem rekursiven Aufruf die Eingabe echt kleiner wird, muss irgendwann ein nichtrekursiver Basisfall erreicht werden.

Dies ist bei generativer Rekursion anders: Wir müssen explizit überlegen, warum eine generativ rekursive Funktion terminiert.

Betrachten Sie hierzu eine Variante des `qsort` Algorithmus von oben, in dem wir den Ausdruck `(smaller-or-equal-than x xs)` ersetzen durch `(smaller-or-equal-than x l)`. Statt also nur aus der Restliste (ohne Pivotelement) die kleiner-oder-gleichen Elemente herauszusuchen, suchen wir in der Liste, die auch das Pivotelement noch enthält:

```
(define (qsort l)
  (match l
    [(list) (list)]
    [(cons x xs)
     (append
      (qsort (smaller-or-equal-than x l))
      (list x)
      (qsort (greater-than x xs)))]))
```

Betrachten wir nun einen Aufruf wie `(qsort (list 5))`. Da `(smaller-or-equal-than 5 (list 5))` die Liste `(list 5)` ergibt, wird dieser Aufruf zu einem rekursiven Aufruf der Form `(qsort (list 5))` führen. Wir haben also eine Endlosschleife produziert.

Wie können wir diesen Fehler vermeiden? Die Terminierung einer generativ rekursiven Funktion kann durch zwei Schritte gezeigt werden:

1. Wir definieren eine Abbildung, die den Satz von Funktionsargumenten auf eine natürliche Zahl abbildet. Diese Abbildung misst quasi die Größe der Eingabe, wobei "Größe" nicht notwendigerweise die physikalische Größe der Daten im Speicher beschreibt sondern die Größe des Problems aus Sicht des Algorithmus.

2. Wir zeigen, dass die Größe der Eingabe bzgl. der Abbildung aus dem ersten Schritt in allen rekursiven Funktionsaufrufen strikt kleiner wird.

Falls die Größe der Originaleingabe (bzgl. der definierten Abbildung) also n ist, so ist sichergestellt, dass die maximale Rekursionstiefe ebenfalls n ist.

Im Falle von quick sort können wir als Abbildung im ersten Schritt die Länge der Liste `l` verwenden. Falls die Länge von `(cons x xs)` n ist, so ist die Länge von `xs` n-1 und damit sind auch `(smaller-or-equal-than x xs)` und `(greater-than x xs)` nicht größer als n-1. Daher wird die Größe der Eingabe in allen rekursiven Aufrufen strikt kleiner.

Im Falle von `first-collision` ist es deutlich komplizierter, die Terminierung zu zeigen. Überlegen Sie, wie in diesem Fall die Größe der Eingabe gemessen werden kann, so dass die Bedingung aus dem zweiten Schritt gilt. Hinweis: Tatsächlich

terminiert `first-collision` nicht immer. Verwenden Sie die Suche nach einem Terminierungsbeweis dazu, um diesen Fehler zu finden und zu beheben.

17 Akkumulation von Wissen

Das Ergebnis eines Funktionsaufrufs hängt nur von den Funktionsparametern ab, nicht jedoch vom Kontext in dem eine Funktion aufgerufen wird. Auf der einen Seite macht diese Eigenschaft Funktionen sehr flexibel, denn sie können in jedem Kontext einfach aufgerufen werden. Auf der anderen Seite sind jedoch manche Probleme so, dass die Problemstellung eine gewisse Abhängigkeit vom Kontext erfordert.

Dies ist insbesondere beim Entwurf rekursiver Funktionen wichtig. Hierzu zwei Beispiele.

17.1 Beispiel: Relative Distanz zwischen Punkten

Nehmen Sie an, wir bekommen eine Liste von Distanzen zwischen Punkten, zum Beispiel `(list 0 50 40 70 30 30)`. Der erste Punkt hat also die Entfernung 0 vom Ursprung, der zweite Punkt die Entfernung 50. Der dritte Punkt ist vom zweiten Punkt 40 entfernt, also vom Ursprung 50+40=90 entfernt.

Nehmen wir an, wir möchten diese Liste umformen zu einer Liste mit absoluten Distanzen zum Ursprung. Wir können auf Basis dieser Informationen und des Beispiels mit der Implementierung dieser Funktion starten:

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute
distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon) ...)
```

Gemäß unseres Entwurfsrezepts können wir versuchen, dieses Problem mittels struktureller Rekursion zu lösen:

```
; (list-of Number) -> (list-of Number)
; converts a list of relative distances to a list of absolute
distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon)
  (match alon
    [(list) ...]
    [(cons x xs) ... x ... (relative-2-absolute xs) ...]))
```

Der Fall für die leere Liste ist trivial, aber wie können wir aus dem Ergebnis von `(relative-2-absolute xs)` und `x` das Gesamtergebnis berechnen? Das erste Element des Ergebnisses ist `x`, aber offensichtlich müssen wir zu jedem Element von `(relative-2-absolute xs)` `x` addieren. Mit dieser Erkenntnis können wir die Funktionsdefinition vervollständigen:

```
; (list-of Number) -> (list-of Number)
```

```
; converts a list of relative distances to a list of absolute
distances
(check-expect (relative-2-absolute (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))
(define (relative-2-absolute alon)
  (match alon
    [(list) (list)]
    [(cons x xs) (cons x
                       (map
                        (lambda (y) (+ y x))
                        (relative-2-absolute xs)))]))
```

Obwohl diese Funktion wie gewünscht funktioniert, ist sie problematisch. Wenn man beispielsweise mit Hilfe der `time` Funktion testet, wie lange die Funktion zur Bearbeitung von Listen verschiedener Länge benötigt, so kann man sehen, dass die benötigte Zeit quadratisch mit der Länge der Liste wächst.

Dieses Phänomen kann man auch direkt in der Funktionsdefinition sehen, denn durch den Aufruf von `map` wird in jedem Rekursionsschritt die komplette bisher berechnete Liste nochmal bearbeitet.

Wenn wir diese Berechnung von Hand durchführen, so gehen wir anders vor. Wir gehen die Liste nur einmal von links nach rechts durch und merken uns die Summe der Zahlen, die wir bisher gesehen haben.

Diese Form der Buchführung passt allerdings nicht zu unserem bisherigen Schema für Rekursion. Wenn wir zwei Listen `(cons x1 xs)` und `(cons x2 xs)` haben, so hat der rekursive Aufruf in beiden Fällen die Form `(f xs)`. Was `f` mit `xs` macht, kann also nicht von `x1` oder `x2` abhängen.

Um dieses Problem zu lösen, führen wir einen zusätzlichen Parameter ein: `accu-dist`. Dieser Parameter repräsentiert die akkumulierte Distanz der bisher gesehenen Punkte. Wenn wir mit der Berechnung starten, wird `accu-dist` mit `0` initialisiert. Während der Berechnung können wir die erste relative Distanz in eine absolute Distanz umwandeln, indem wir `accu-dist` hinzuaddieren; für die Berechnung im rekursiven Aufruf müssen wir den Wert des Akkumulators auf den neuen Kontext anpassen.

Insgesamt ergibt sich damit der folgende Code:

```
; (list-of Number) Number -> (list-of Number)
(define (relative-2-absolute-with-acc alon accu-dist)
  (match alon
    [(list) (list)]
    [(cons x xs)
     (local [(define x-absolute (+ accu-dist x))]
       (cons x-absolute (relative-2-absolute-with-acc xs x-
absolute)))]))
```

Diese Definition ist noch nicht ganz äquivalent zu `relative-2-absolute`, aber wir können die ursprüngliche Signatur leicht rekonstruieren:

```
; (list-of Number) -> (list-of Number)
```

```
; converts a list of relative distances to a list of absolute
distances

(check-expect (relative-2-absolute-2 (list 0 50 40 70 30 30))
              (list 0 50 90 160 190 220))

(define (relative-2-absolute-2 alon)
  (local
     ; (list-of Number) Number -> (list-of Number)
    [(define (relative-2-absolute-with-acc alon accu-dist)
       (match alon
         [(list) (list)]
         [(cons x xs)
          (local [(define x-absolute (+ accu-dist x))]
            (cons x-absolute
                  (relative-2-absolute-with-acc xs x-
absolute)))]))]
    (relative-2-absolute-with-acc alon 0)))
```

Einige Experimente mit `time` bestätigen, dass `relative-2-absoluste-2` viel effizienter als `relative-2-absoluste` ist und statt einem quadratischen nur noch ein lineares Wachstum der Laufzeit aufweist.

17.2 Beispiel: Suche in einem Graphen

Als zweites Beispiel wollen wir uns das Problem anschauen, in einem Graphen einen Weg zwischen zwei Knoten zu finden (sofern dieser existiert).

Ein gerichteter Graph ist eine Menge von Knoten und eine Menge von gerichteten Kanten zwischen den Knoten. Es gibt unterschiedliche Möglichkeiten, Graphen zu repräsentieren. Wir entscheiden uns für eine Repräsentation, bei der zu jedem Knoten gespeichert wird, welche Kanten von diesem Knoten ausgehen:

```
; A Node is a symbol

; A Node-with-neighbors is a (list Node (list-of Node))

; A Graph is a (list-of Node-with-neighbors)
```

Hier ist ein Beispiel für einen Graphen, der dieser Datendefinition entspricht:

```
(define graph1
  '((A (B E))
    (B (E F))
    (C (D))
    (D ())
    (E (C F))
    (F (D G))
    (G ())))
```

Wir suchen also eine Funktion, deren Aufgabe wir wie folgt definieren können:

```
; Node Node Graph  ->  (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F G))
(define (find-route origination destination G) ...)
```

Zunächst einmal können wir feststellen, dass wir dieses Problem mit struktureller Rekursion nicht sinnvoll lösen können. Wenn wir zum Beispiel in `graph1` eine Route von B nach E suchen, nützt uns die Information, ob es in `(rest graph1)` eine solche Route gibt, nicht viel, denn es könnte ja sein, dass die Route durch A geht.

Um dieses Problem mit generativer Rekursion zu lösen, müssen wir die in §16.3 "Entwurf von generativ rekursiven Funktionen" beschriebenen fünf Fragen beantworten:

1. Ein trivial lösbares Problem ist die Frage nach einer Route von einem Knoten `n` zu sich selbst.
2. Die Lösung ist im trivialen Fall `(list n)`.
3. Wenn das Problem nicht-trivial ist, können wir für jeden Nachbarn des Knoten das Problem generieren, eine Route von dem Nachbarn zum Ziel zu finden.
4. Wenn eines dieser Probleme erfolgreich gelöst wird, so ist das Gesamtergebnis der Weg zum Nachbarn gefolgt von der gefundenen Route vom Nachbarn aus.
5. Das Terminierungsargument verschieben wir auf später.

Insgesamt ergibt sich damit das folgende Programm:

```
; Node Node Graph  ->  (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F G))
(define (find-route origination destination G)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define possible-route
                    (find-route/list (neighbors origination G) destination G)))
            (cond
              [(boolean? possible-route) false]
              [else (cons origination possible-route)]))]))

; (list-of Node) Node Graph  ->  (list-of Node) or false
; to create a path from some node on lo-Os to D
; if there is no path, the function produces false
(check-member-of (find-route/list '(E F) 'G graph1) '(F G) '(E F G))
```

```
(define (find-route/list lon D G)
  (match lon
    [(list) false]
    [(cons n ns)
     (local ((define possible-route (find-route n D G)))
       (cond
         [(boolean? possible-route) (find-route/list ns D G)]
         [else possible-route]))]))


; Node Graph -> (list-of Node)
; computes the set of neighbors of node n in graph g
(check-expect (neighbors 'B graph1) '(E F))
(define (neighbors n g)
  (match g
    [(cons (list m m-neighbors) rest)
     (if (symbol=? m n)
         m-neighbors
         (neighbors n rest))]
    [(list) (error "node not found")]))
```

Algorithmen wie `find-route` nennt man auch *Backtracking Algorithmen*, weil sie systematisch Alternativen ausprobieren und beim Erschöpfen der lokal sichtbaren Alternativen zurückspringen zum nächsten Punkt, an dem noch nicht alle Alternativen erschöpft sind.

Kommen wir nun zurück zu der Frage, ob der Algorithmus immer terminiert. Woher wissen wir, dass wir wirklich "näher am Ziel" sind, wenn wir von einem Knoten zu seinem Nachbarn gehen? "Näher zum Ziel" heißt in diesem Fall, dass wir entweder näher am Zielknoten sind, oder dass wir die Zahl der Alternativen, die noch ausprobiert werden müssen, verringert haben.

Allerdings kann man relativ leicht sehen, dass dieser Algorithmus tatsächlich nicht immer terminiert, nämlich dann, wenn der Graph Zyklen enthält. Wenn wir beispielsweise den Graph

```
(define graph2
  '((A (B E))
    (B (A E F))
    (C (D))
    (D (C))
    (E (C F))
    (F (D G))
    (G (D))))
```

betrachten und den Ausdruck

```
(find-route 'A 'G graph2)
```

auswerten, so stellen wir fest, dass die Auswertung nicht terminiert, weil wir entlang der Route `'(A B A B A B ...)` im Kreis laufen und keinerlei Fortschritt machen: Ein Aufruf von

```
(find-route 'A 'G graph2)
```

bewirkt einen Aufruf von

```
(find-route/list '(B E) 'G graph2)
```

; dieser wiederum zieht einen Aufruf von

```
(find-route 'A 'G graph2)
```

nach sich.

Das Problem ist, genau wie im `relative-2-absolute` Beispiel oben, dass wir nichts über den Kontext wissen, in dem `find-route` aufgerufen wird. In diesem Beispiel wollen wir wissen, ob `find-route` mit einem Startknoten aufrufen, von dem wir bereits gerade versuchen, eine Route zum Ziel zu finden.

Der Akkumulationsparameter, den wir in diesem Beispiel benötigen, repräsentiert die Liste der Knoten, die wir bereits auf dem Pfad, der konstruiert wird, besucht haben. Ein solcher Parameter läßt sich leicht hinzufügen. Da `find-route` und `find-route/list` wechselseitig rekursiv sind, wird der zusätzliche Parameter durch beide Funktionen durchgereicht. Wenn innerhalb von `find-route/acc` `find-route/list` aufgerufen wird, so wird durch den Ausdruck `(cons origination visited)` die Invariante, dass `visited` stets die Menge der bereits besuchten Knoten repräsentiert, sichergestellt.

```
(define (find-route/acc origination destination G visited)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local ((define possible-route
                    (find-route/list (neighbors origination G)
                                     destination
                                     G
                                     (cons origination visited))))
            (cond
              [(boolean? possible-route) false]
              [else (cons origination possible-route)]))]))

(define (find-route/list lon D G visited)
  (match lon
    [(list) false]
    [(cons n ns)
     (local ((define possible-route (find-
route/acc n D G visited)))
       (cond
```

```
          [(boolean? possible-route)
           (find-route/list ns D G visited)]
          [else possible-route]))]))
```

Das reine Berechnen und Durchreichen des Akkumulationsparameters ändert allerdings nichts grundlegendes am Programmverhalten. Insbesondere terminiert der Algorithmus weiterhin nicht bei zyklischen Graphen.

Aber es ist nun leicht, das Programm so zu ändern, dass bereits besuchte Knoten nicht noch einmal besucht werden, nämlich indem wir in `find-route` von der Liste der Nachbarn `(neighbors origination G)` die Knoten abziehen, die bereits in `visited` enthalten sind.

Hierzu definieren wir eine kleine Hilfsfunktion:

```
; [X] (list-of X) (list-of X) -> (list-of X)
; removes all occurences of members of l1 from l2
(check-expect (removeall '(a b) '(c a d b e a b g))
              '(c d e a b g))
(define (removeall l1 l2)
  (match l1
    [(list) l2]
    [(cons x xs) (removeall xs (remove x l2))]))
```

und können damit `find-route` wie oben beschrieben ändern:

```
; Node Node Graph ->  (list-of Node) or false
; to create a path from origination to destination in G
; if there is no path, the function produces false
(check-member-of (find-route 'A 'G graph1) '(A E F G) '(A B E F G) '(A B F G))
(check-member-of (find-route 'A 'G graph2) '(A E F G) '(A B E F G) '(A B F G))
(define (find-route origination destination G)
  (find-route/acc origination destination G '()))

; Node Node Graph (list-of Node) ->  (list-of Node) or false
; computes paths from origination to destination in G
; keeps track of visited nodes to detect cycles in the graph
(define (find-route/acc origination destination G visited)
  (cond
    [(symbol=? origination destination) (list destination)]
    [else (local
            [(define possible-route
               (find-route/list
                (removeall visited (neighbors origination G))
                destination
                G
                (cons origination visited)))]
            (cond
              [(boolean? possible-route) false]
              [else (cons origination possible-route)]))]))
```

Der Test mit `graph2` suggeriert, dass der Algorithmus nun auch auf Graphen mit Zyklen terminiert. Wie sieht das Argument für den allgemeinen Fall aus?

Gemäß der in §16.4 "Terminierung" beschriebenen Methodik müssen wir eine Abbildung definieren, die die Funktionsargumente von `find-route` auf eine natürliche Zahl abbildet. Sei n die Zahl der Knoten in `G` und m die Zahl der Knoten in `visited`. Dann definieren wir die Größe der Eingabe von `find-route` als n-m, also die Anzahl der Knoten, die noch nicht besucht worden.

Die Differenz n-m ist stets eine natürliche Zahl, weil durch das Entfernen der bereits besuchten Knoten aus den Nachbarn (`(remove-all visited (neighbors origination G))`) im Aufruf `(cons origination visited)` stets nur Knoten aus dem Graph hinzugefügt werden, die vorher noch nicht in `visited` enthalten waren. Daher ist `visited` stets eine Teilmenge der Knoten des Graphs; kein Knoten kommt in `visited` mehrfach vor.

Bezüglich des zweiten Schritts aus §16.4 "Terminierung" können wir festhalten, dass die Größe n-m durch das Hinzufügen eines neuen Knoten in `(cons origination visited)` stets strikt kleiner wird.

Der rekursive Aufruf in `find-route` ist jedoch indirekt via `find-route/list`. Diese Funktion ist strukturell rekursiv und terminiert daher immer. Desweiteren können wir festhalten, dass `find-route/list` sowohl `G` als auch `visited` stets unverändert durchreicht. Wenn `find-route/list` daher `find-route` aufruft, so tut es dies mit unveränderten Werten für `G` und `visited`.

Insgesamt können wir festhalten, dass wir damit gezeigt haben, dass die Rekursionstiefe durch die Anzahl der Knoten des Graphen begrenzt ist und damit der Algorithmus für alle Graphen terminiert.

17.3 Entwurf von Funktionen mit Akkumulatoren

Nachdem wir zwei Beispiele gesehen haben, in denen ein Akkumulator sinnvoll ist, wollen wir nun diskutieren, wann und wie man im Allgemeinen Funktionen mit Akkumulatoren entwerfen sollte.

Zunächst mal sollte man einen Funktionsentwurf mit Akkumulator nur dann in Erwägung ziehen, wenn der Versuch, mit dem Standard-Entwurfsrezept die Funktion zu entwerfen, gescheitert ist, oder zu zu kompliziertem oder zu langsamem Code führt.

Die Schlüsselaktivitäten beim Entwurf einer Funktion mit Akkumulator sind: 1) zu erkennen, dass die Funktion einen Akkumulator benötigt (oder davon profitieren würde), und 2) zu verstehen, was genau der Akkumulator repräsentiert (die *Akkumulator-Invariante*).

17.3.1 Wann braucht eine Funktion einen Akkumulator

Wir haben zwei Gründe gesehen, wieso Funktionen einen Akkumulator benötigen:

1. Wenn eine Funktion strukturell rekursiv ist und das Ergebnis des rekursiven Aufrufs wieder von einer rekursiven Hilfsfunktion verarbeitet wird. Häufig kann durch einen Akkumulator die Funktionalität der rekursiven Hilfsfunktion in die Hauptfunktion mit eingebaut werden und statt verschachtelter Iterationen (die

häufig zu quadratischer Laufzeit führen) ist dann häufig eine einfache Iteration ausreichend.

Hier ein weiteres Standardbeispiel:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
; structural recursion
(define (invert alox)
  (cond
    [(empty? alox) empty]
    [else (make-last-item (first alox) (invert (rest alox)))]))

; make-last-item : X (listof X)  ->  (listof X)
; to add an-x to the end of alox
; structural recursion
(define (make-last-item an-x alox)
  (cond
    [(empty? alox) (list an-x)]
    [else (cons (first alox) (make-last-item an-
x (rest alox)))]))
```

2. Wenn wir eine generativ rekursive Funktion haben, dann kann es schwierig sein, diese Funktion so zu entwerfen, dass sie für alle Eingaben eine korrekte Ausgabe berechnet. Im Beispiel oben haben wir gesehen, dass wir uns merken mussten, welche Knoten wir bereits besucht haben, um Terminierung bei Zyklen im Graph sicherzustellen. Im Allgemeinen können wir mit einem Akkumulator beliebiges Wissen über die aktuelle Iteration akkumulieren und nutzen. Wenn es also Wissen gibt, welches nicht lokal verfügbar ist sondern nur im Verlauf der Iteration angesammelt werden kann, so ist ein Akkumulator das richtige Mittel.

Diese beiden Möglichkeiten sind nicht die einzigen, aber sie sind sehr häufig.

17.3.2 Template für Funktionen mit Akkumulatoren

Wenn wir uns dafür entschieden haben, eine Funktion mit Akkumulator auszustatten, so ist es sinnvoll, ein Template für die Funktionsdefinition zu erstellen. Dieses sieht so aus, dass wir die Funktion mit Akkumulator zu einer mit `local` definierten lokalen Funktion der eigentlich zu definierenden Funktion machen und diese Funktion dann im Body des `local` Ausdrucks aufrufen.

Im Beispiel von oben sieht diese Template so aus:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator ...
          (define (rev alox accumulator)
```

```
            (cond
              [(empty? alox) ...]
              [else
                ... (rev (rest alox) ... ( first alox) ... accumulator)
                ...])))
    (rev alox0 ...)))
```

17.3.3 Die Akkumulator-Invariante

Als nächstes ist es sinnvoll, die Akkumulator-Invariante zu formulieren. Die Akkumulator-Invariante sagt, was der Akkumulator in jedem Iterationsschritt repräsentiert.

Zu diesem Zweck müssen wir uns überlegen, welche Daten wir akkumulieren wollen, so dass der Akkumulator uns bei der Implementation der Funktion hilft.

Im Falle von `invert` würde es uns helfen, wenn der Akkumulator die Listenelemente repräsentiert, die wir bisher gesehen haben (in umgekehrter Reihenfolge), denn dann können wir diese im `empty` Fall zurückgeben statt `make-last-item` aufzurufen.

Diese Akkumulatorinvariante sollten wir in den Code hineinschreiben:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; ;; accumulator is the reversed list of all those
items
          ; on alox0 that precede alox
          (define (rev alox accumulator)
            (cond
              [(empty? alox) ...]
              [else
                ... (rev (rest alox) ... ( first alox) ... accumulator)
                ...])))
    (rev alox0 ...)))
```

17.3.4 Implementation der Akkumulator-Invariante

Der nächste Schritt ist, dafür zu sorgen, dass die Akkumulator-Invariante auch tatsächlich eingehalten wird. Dies bedeutet, dass wir uns den initialen Wert für den Akkumulator überlegen müssen sowie die Berechnung, wie wir im rekursiven Aufruf den neuen Akkumulator erhalten. In unserem Beispiel ist offensichtlich der initiale Akkumulator `empty` und der neue Akkumulator im rekursiven Aufruf `(cons (first alox) accumulator)`:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator is the reversed list of all those
items
```

```
          ; on alox0 that precede alox
          (define (rev alox accumulator)
            (cond
              [(empty? alox) ...]
              [else
                ... (rev (rest alox) (cons (first alox) accumulator))
                ...])))
    (rev alox0 empty)))
```

17.3.5 Nutzung des Akkumulators

Die Akkumulator-Invariante zu implementieren ändert noch nichts am externen Verhalten der Funktion. Der Sinn der ganzen Operation liegt darin, dass wir nun den Akkumulator nutzen können, um die Funktion zu implementieren. Wenn wir die Akkumulator-Invariante präzise formuliert haben, ist dieser Schritt typischerweise einfach:

```
; invert : (listof X)  ->  (listof X)
; to construct the reverse of alox
(define (invert alox0)
  (local (; accumulator is the reversed list of all those
items
          ; on alox0 that precede alox
          (define (rev alox accumulator)
            (cond
              [(empty? alox) accumulator]
              [else
                (rev (rest alox) (cons (first alox) accumulator))])))
    (rev alox0 empty)))
```